

Advanced Programming

223MI, 558EC

[Eric Medvet](#)

A.Y. 2019/2020

Lecturer

Eric Medvet

- Associate Professor of Computer Engineering at [Departmenet of Engineering and Architecture, University of Trieste](#)
- Online at: [medvet.inginf.units.it](#)

Research interests:

- Evolutionary Computation
- Machine Learning applications

Labs:

- [Evolutionary Robotics and Artificial Life lab](#)
- [Machine Learning lab](#)

Materials

Teacher slides:

- available [here](#)
- might be updated during the course

Intended usage:

- slides should contain every concept that has to be taught/learnt
- **but**, slides are designed for consumption during a lecture, they might be suboptimal for self-consumption → **take notes!**

Exam

Two options:

1. project + written test + home assignments
 - grade: weighted average with weights 60%, 30%, 10%
2. project + written test + oral exam
 - grade: weighted average with weights 40%, 30%, 30%

Failed if at least one part is graded <6/10.

Home assignments

Home assignments:

- short exercise assigned **during the course**
- will start in the classroom, under teacher's supervision
- student's output to be submitted within the **deadline**, one for each assignment
- grade, for each assignment:
 - 0/10: not submitted or missed deadline
 - up to 5/10: submitted, but not working
 - up to 8/10: submitted and almost working
 - 10/10: submitted and fully working
- grades communicated during the course

More details at first assignment.

List of exercises and home assignments

Just for reference:

1. [Anagrams](#) (also assignment)
2. [Equivalence](#) (also assignment)
3. [File array](#)
4. [Compressed file array](#) (also assignment)

Project

Project:

- assigned at the end of the course
- student's output: software, tests, brief document
- to be submitted **within the exam date**
- grade:
 - 0/10: not submitted or missed deadline
 - 5/10 to 10/10: submitted, depending on
 - quality of code
 - software structure
 - document (mainly clarity)
 - test coverage
 - degree of working

More details at project assignment.

Written test and oral exam

Written test:

- a few questions with short open answers and short programming exercises

Oral exam:

- questions and short programming exercises

Course content

In brief:

1. **Java** as object-oriented programming language
2. Tools and methods for programming
3. Distributed programming

(See the [syllabus](#))

Exercises

There we'll be several exercises:

- approx. 20h on 72h
- software design and implementation exercises
- bring your own device
- in classroom
 - with the teacher actively investigating about your progresses

Practice is fundamental!

(**Computational thinking is an outcome of the coding practice.** See Nardelli, Enrico.

"[Do we really need computational thinking?](#)" Communications of the ACM 62.2 (2019):
32-35.)

You?

- How many of you already know object-oriented programming?
- How many of you already know Java?

Java: what and why?

Why Java?

Practical motivations:

- write once, run anywhere
- large community support, solid tools, many APIs
- free

Teaching-related motivations:

- favors **abstraction**
- realizes many **key concepts** of programming and languages

Language, platform

Java is both:

- a programming language
- a software platform

We will focus on:

- (mainly) the programming language
- (secondarily) the Java 2 Platform, Standard Edition (J2SE)

Java platform

Mainly composed of:

- an execution engine
- a compiler
- a set of specifications
- a set of libraries, with Application Program Interfaces (**APIs**)

Platforms (also called **editions**) differ mainly in the libraries. Most common editions:

- Java 2 Platform, Standard Edition (J2SE): for general-purpose, desktop applications
- Java 2 Platform, Enterprise Edition (J2EE): J2SE + APIs/specifications for multi-tier client-server enterprise applications

Main tools

What is needed for developing in Java?

- Java Development Kit (**JDK**), absolutely necessary
- API documentation (briefly **javadoc**)
- Integrated Development Environment (**IDE**)
 - many options
 - when used proficiently, makes development more efficient and effective

JDKs

One JDK per (vendor, platform, version, os+architecture).

A few options:

- [Oracle JDK](#)
- [OpenJDK](#)

JDK versions

We are currently at version **13**. History from [Wikipedia](#):

Version	Release date	End of Free Public Updates ^{[6][7]}	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	January 2019 for Oracle (commercial) December 2020 for Oracle (personal use) At least September 2023 for AdoptOpenJDK At least June 2023 ^[8] for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	At least August 2024 ^[8] for Amazon Corretto September 2022 for AdoptOpenJDK	September 2026
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK	N/A
Java SE 16	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	TBA	TBA

Legend: Old version Older version, still maintained Latest version Latest preview version Future release

Java IDEs

Many excellent options.

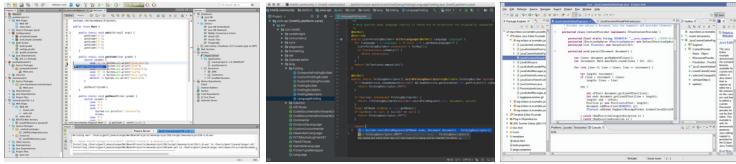
Classic, desktop-based:

- [Apache NetBeans](#)
 - the one I know better
- [Jetbrains IntelliJ IDEA](#)
- [Eclipse IDE](#)

Cloud-based:

- Full: [Codenvy](#), [Eclipse Che + OpenShift](#)
 - steeper learning curve
- Light: [CompileJava.net](#), [rePL.it](#)

IDE: your best friend



Pros:

- makes typing much faster
- greatly helps following conventions
- makes software lifecycle operations faster
- helps finding errors

Cons:

- steep learning curve
- may hide some programming/development concepts

Basic operations

Compiling and executing

.java vs. .class files

- The **source** code is in one or more `.java` files
- The **executable** code is in one or more `.class` files

Compilation: obtaining a `.class` from `.java`

```
javac Greeter.java
```

Class-file

.java contains **exactly one** class definition (common case)

```
public class Greeter {  
    //...  
}
```

Or it contains more than one class definitions, at most one non-inner has the **public** modifier (we will see):

```
public class Greeter {  
    //...  
}  
class Helper {  
    //...  
}
```

```
public class Greeter {  
    //...  
    public class InnerHelper {  
        //...  
    }  
}
```

Compilation results always in one **.class** for each class definition!

Execution

An **application** is a `.class` compiled from a `.java` that has a "special function" `main`

```
public class Greeter {  
    public static void main(String[] args) {  
        //...  
    }  
}
```

An application can be executed:

```
java Greeter
```

(Or `java Greeter.class`, or in other ways)

Class and the virtual machine

- A `.class` file contains executable code in a binary format (**bytecode**)
- The bytecode is **interpreted** by a program (`java`) that simulates a machine: the Java Virtual Machine (**JVM**)

JVM:

- like a real machine, but simulated: it has an instruction set and manages memory
- agnostic with respect to the physical machine
- does not know Java language, knows bytecode

JVM language

An example of the JVM specification, the instruction `iadd`:

Operand Stack

..., value1, value2 → ..., result

Description

Both value1 and value2 must be of type int. The values are popped from the operand stack. The int result is value1 + value2. The result is pushed onto the operand stack.

The result is the 32 low-order bits of the true mathematical result in a sufficiently wide two's-complement format, represented as a value of type int. If overflow occurs, then the sign of the result may not be the same as the sign of the mathematical sum of the two values.

Portability

A `.class` executable can be executed on any machine for which a program exists that simulates the JVM (i.e., a `java`).

"One" `.java` compiled to "one" `.class`, can be executed:

- on Linux x64, with the `java` program for Linux x64
- on Solaris SPARC x64, with the `java` program for Solaris SPARC x64
- Mac OS X, Windows, Android, ...

→ **Write once, run anywhere**

Who executes what

Note:

- the OS executes `java`
- `java` executes the bytecode
 - simulates the JVM which executes the bytecode
- the OS cannot execute the bytecode!

Where is `java`?

`java` is part of the Java Runtime Environment (**JRE**), which includes also necessary libraries. The JRE is part of the JDK.

The JDK contains (mainly):

- the compiler `javac`
- the JRE (including `java`)
- many compiled classes (`.class` files)
- other development tools

It does not contain:

- the classes source code
- the documentation

Inside the JDK

```
eric@cpu:~$ ls /usr/lib/jvm/java-11-openjdk-amd64 -la
totale 48
drwxr-xr-x 2 root root 4096 feb  6 12:08 bin
drwxr-xr-x 4 root root 4096 feb  6 12:08 conf
drwxr-xr-x 3 root root 4096 feb  6 12:08 include
drwxr-xr-x 6 root root 4096 feb  6 12:08 lib
...
...
```

```
eric@cpu:~$ ls /usr/lib/jvm/java-11-openjdk-amd64/bin/ -la
totale 632
-rwxr-xr-x 1 root root 14576 gen 15 16:14 jar
-rwxr-xr-x 1 root root 14576 gen 15 16:14 jarsigner
-rwxr-xr-x 1 root root 14560 gen 15 16:14 java
-rwxr-xr-x 1 root root 14608 gen 15 16:14 javac
-rwxr-xr-x 1 root root 14608 gen 15 16:14 javadoc
...
...
```

Usually the JDK can be **installed** on the OS; but it can be simply uncompressed somewhere.

Documentation and source code

Do you need them?

- documentation: **yes**, in practice
- source code: no, but it can be useful for understanding

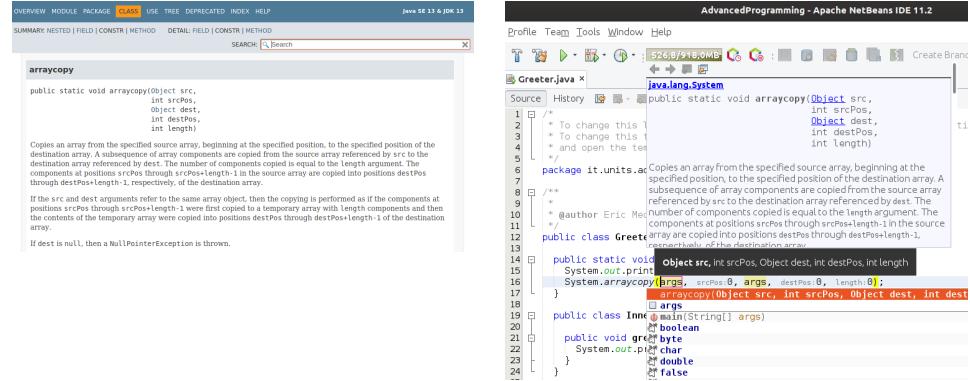
How to access the documentation?

- online:
<https://docs.oracle.com/en/java/javase/13/docs/api/index.html>
- through IDE, often deeply integrated with autocompletion and suggestions

How to access the source code?

- download it
- download it in the IDE, access through the IDE (**ctrl+click**)

Consuming the documentation



The screenshot shows the Apache NetBeans IDE 11.2 interface. On the left, a Javadoc window for the `arraycopy` method is open, displaying its source code and detailed documentation. The documentation explains that it copies a subsequence of array components from the source array to the destination array. It also notes that if the `src` and `dest` arguments refer to the same array object, the copying is performed as if the components at positions `srcPos` through `srcPos+length-1` were first copied to a temporary array with `length` components and then the contents of the temporary array were copied into positions `destPos` through `destPos+length-1` of the destination array. A note states that if `dest` is null, a `NullPointerException` is thrown.

The right side of the interface shows the code editor for a file named `Greeter.java`. The code imports `java.lang.System` and defines a class `Greeter` with a main method. A tooltip is visible over the `arraycopy` call in the code editor, providing a preview of the Javadoc text.

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array. A subsequence of array components are copied from the source array referenced by src to the destination array referenced by dest. The number of components copied is equal to the length argument. The components at positions srcPos through srcPos+length-1 in the source array are copied into positions destPos through destPos+length-1, respectively, of the destination array.

If the src and dest arguments refer to the same array object, then the copying is performed as if the components at positions srcPos through srcPos+length-1 were first copied to a temporary array with length components and then the contents of the temporary array were copied into positions destPos through destPos+length-1 of the destination array.

If dest is null, then a NullPointerException is thrown.

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
    System.arraycopy(args, args, dest, dest, length);
```

```
public class Greeter {
    public static void main(String[] args) {
        boolean b;
        byte by;
        System.out.print("char");
        char c;
        double d;
        b = false;
    }
}
```

Java: interpreted or compiled?

Both!

- `.java` to `.class` through **compilation**
- `.class` is then **interpreted** by `java` for being executed on OS

Actually, things are much more complex:

- on the fly optimization
- recompilation
- compilation to native code
- ...

Intepretation and efficiency

Is intepretation efficient?

- irrelevant question

Relevant question: is my software fast enough?

- it depends: measure!

If not?

1. profile
2. find opportunities for improvement
3. improve

(that is: write good code!)

The problem is rarely in interpretation!

Beyond speed of execution

When choosing a language/platform, speed of execution is just one factor.

Other factors:

- code maintainability
- documentation/support availability
- quality of development tools
- libraries availability
- ...
- previous knowledge
- ...

In general, consider **all costs** and assess options.

Basic concepts

Hello goal!

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Goal: **deep understanding** of this code

- not just what it "does"
- but the role of every part of the code

(Take code highlighting with care!)

Objects

"Things" (entities that can be manipulated in the code) are **objects**

- objects exist
- but **do not have a name**

The code manipulates objects through **references**

- references have a symbolic name, called **identifier**

Both can be created.

object \neq reference

Creating objects and references

```
String s;
```

- **create a reference** to object of type **String** with identifier **s**
(briefly: reference **s**)
 - a reference has always a name!
- no object is created

s

```
String s = new String("Content");
```

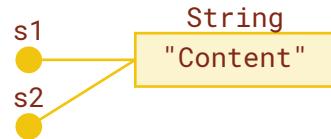
- **create reference **s** of type **String****
- **create object** of type **String** and init it with "**Content**"
- make **s** reference the new object



Many references

```
String s1 = new String("Content");
String s2 = s1;
```

- create reference **s1** and make it reference the new objects
- create reference **s2** and make it reference the object referenced by **s1**



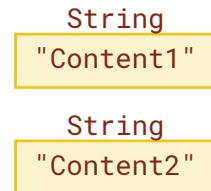
Objects+references diagram: where? when?

- for now: in the memory of the JVM, conceptually.
- at runtime

No references

```
new String("Content1");
new String("Content2");
```

- create object of type **String** and init it with "Content1"
- create object of type **String** and init it with "Content2"



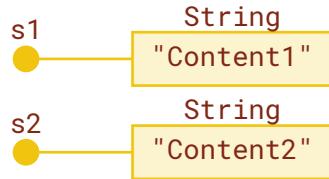
The two objects are not referenced: they cannot be manipulated!

Changing referenced object

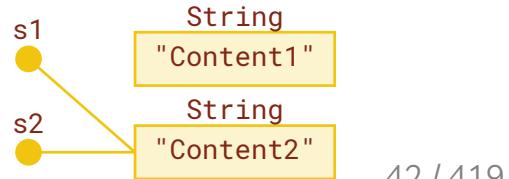
```
String s1 = new String("Content1");
String s2 = new String("Content2");
s1 = s2;
```

- create reference `s1`, create new `String` inited with "Content1", and make `s1` reference it
- create reference `s2`, create new `String` inited with "Content2", and make `s2` reference it
- make `s1` reference the object referenced by `s2`

After 2nd line:



After 3rd line:



Objects+references diagram

```
String dog1 = new String("Simba");
String dog2 = new String("Gass");
new String("Chip");
String squirrel2 = new String("Chop");
dog1 = dog2;
dog2 = dog1;
squirrel2 = squirrel2;
```

Draw the diagram

- after the 3rd line
- after the last line

Manipulating objects

Key questions:

- which operations can be applied on an object?
- how to specify them in the code?
- can operations have parameters?

Manipulating objects

Answers (in brief):

- which operations can be applied on an object?
 - every object is an **instance** of a **type**
 - a type defines operations applicable to instances
- how to specify them in the code?
 - with the **dot notation**: `refName.opName()`
- can operations have input and output parameters?
 - input parameters can be specified between round brackets:
`refName.opName(otherRefName1, otherRefName2)`
 - output can be referenced: `retRefName = refName.opName()`

(We are just scratching the surface; we'll go deeper ahead.)

Classes

A **class** is a type

- every object is an instance of a class
- (in a Java software) there are several classes
 - some are part of the Java language
 - some are part of APIs
- every application includes at least one class, defined by the developer

Classes

class \neq object

Some expressions that are clues of misunderstanding:

- "I wrote the code of an object"
- "X references a class"

Some correct expressions that hide complexity:

- "instantiating a X" (where X is a class)
 - means "creating an object of class X"

Primitive types

Some types are not classes, but **primitive types**:

- `boolean`
- `char`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`

We'll discuss them later.

Using classes

Methods

An operation applicable on a object is a **method** of the corresponding class.

Every method has a **signature**, consisting of:

- name
- sequence of types of input parameters (name not relevant)
- type of output parameter (aka **return type**)
 - possibly **void**, if there is no output

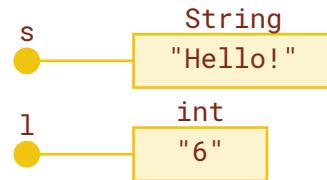
Examples (class String):

- `char charAt(int index)`
- `int indexOf(String str, int fromIndex)`
- `String replace(CharSequence target, CharSequence replacement)`
- `void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`

Method invocation

```
String s = new String("Hello!");  
int l = s.length();
```

- execute (who?) the operation defined by the method `length()` on the object referenced by `s`
 - briefly: invoke `length()` on `s`
- create reference `l` of type `int`
- make `l` reference the object created upon the execution of the operation



Can a method be invoked?

```
String s = new String("Hello! ");
int l = s.length();
```

The compiler (`javac`) verifies (also) that:

- the type of `s` has a method with name `length`
 - briefly: `s` has a method `length`
- the method is used consistently with the signature
 - input and output types are correct

This check:

- is done **at compile time** (by the compiler)
- is much harder than it appears: we'll see
- is hugely **useful** for avoiding errors and designing good software

Colloquially, Java is said to be strongly typed.

52 / 419

Method overloading

A class can have many methods:

- with the same name
- but different input parameters

From another point of view: at most one method with the same (name, input parameters), regardless of the output type.

```
String s = new String("Hello!");
PrintStream ps = new PrintStream(/*...*/);
ps.println(s);
```

Type	Method	Description
	void println()	Terminates the current line by writing the line separator string.
	void println(boolean x)	Prints a boolean and then terminate the line.
	void println(char x)	Prints a character and then terminate the line.
	void println(char[] x)	Prints an array of characters and then terminate the line.
	void println(double x)	Prints a double and then terminate the line.
	void println(float x)	Prints a float and then terminate the line.
	void println(int x)	Prints an integer and then terminate the line.
	void println(long x)	Prints a long and then terminate the line.
	void println(String s)	Prints a String and then terminate the line.

Class [PrintStream](#)

How many methods?

A lot of classes, a lot of methods!

You **have to**:

- use the documentation (briefly: **javadoc**)
- and/or proficiently use the IDE
 - **autocompletion!** (Ctrl+space)

Often, the signature alone is sufficient for understanding what the method does (e.g., `int length()` of `String`).

This is because who wrote the code for `String` correctly chose the signature: in particular the name.

There are **naming conventions**!

Naming conventions

Extremely important!

- code is not just for machines, it's for humans
- with code, a human tells another human about a procedure

Many of them: some just for Java, some broader.

- we'll see part of them, gradually

The degree to which naming conventions are followed concurs in determining the **quality of the code**.

(There are many sources online: e.g., [JavaPoint](#))

Classes and methods

Class:

- a noun, a name of an **actor**: e.g., `Dog`, `Sorter`, `Tree`
- representative of the entity the class represents
- upper camel case (aka dromedary case): `HttpServerResponse`

Method:

- a verb, a name of an action: e.g., `bark`, `sort`, `countNodes`
 - with acceptable exceptions: e.g., `size`, `nodes`
- representative of the operation the method performs
- lower camel case: `removeLastChar`

References

Reference:

- consistent with the corresponding type
- representative of the specific use of the referenced object:
 - not every **String** has to be referenced by **s**
 - specific! e.g., **numOfElements** is better than **n**
- lower camel case

God gave us IDEs, IDEs give us **autocompletion!** Don't spare on chars.

Associative dot operator

```
String s = new String(" shout! ");
s = s.trim().toUpperCase();
```

- invoke `trim()` on object referenced by `s`
- invoke `toUpperCase()` on the object resulting from `trim()` invocation
- make `s` reference the object resulting from `toUpperCase()` invocation

Type	Method	Description
<code>String</code>	<code>toUpperCase()</code>	Converts all of the characters in this <code>String</code> to upper case using the rules of the default locale.
<code>String</code>	<code>trim()</code>	Returns a string whose value is this string, with all leading and trailing space removed, where space is defined as any character whose codepoint is less than or equal to ' <code>U+0020</code> ' (the space character).

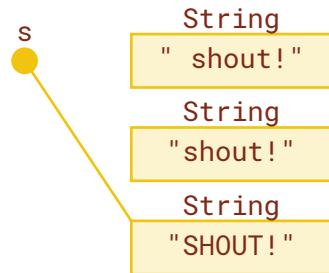
Objects and references diagram

```
String s = new String(" shout! ");
s = s.trim().toUpperCase();
```

After 1st line:



After 2nd line:



Constructor

Every class T has (at least) one special method called **constructor** that, when invoked:

- results in the creation of a new object of class T
- inits the new object

Special syntax for invocation using the **new** keyword:

```
Greeter greeter = new Greeter();
```

There are a few exceptions to this syntax:

- **String** s = "hi!"; same of **String** s = **new** **String**("hi!");

Initialization

What happens with initialization depends on the constructor.

Class Date: The class **Date** represents a specific instant in time, with millisecond precision.

Modifier	Constructor	Description
	Date()	Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

Probably two different initialization outcomes:

```
Date now = new Date();
// some code doing long things
Date laterThanNow = new Date();
```

Multiple constructors

A class C can have more than one constructors:

- at most one with the same input parameters
(the name and the return type are always the same)
 - name: the very same name of the class (e.g., `Date` → `Date()`)
 - return type: C (e.g., `Date` → `Date`)

Class `String`:

Modifier	Constructor	Description
	<code>String()</code>	Initializes a newly created <code>String</code> object so that it represents an empty character sequence.
	<code>String(char[] value)</code>	Allocates a new <code>String</code> so that it represents the sequence of characters currently contained in the character array argument.
	<code>String(String original)</code>	Initializes a newly created <code>String</code> object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

Other example: Socket

Class [Socket](#): This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

Modifier	Constructor	Description
	Socket()	Creates an unconnected Socket.
	Socket(String host, int port)	Creates a stream socket and connects it to the specified port number on the named host. Deprecated. Use DatagramSocket instead for UDP transport.
	Socket(String host, int port, boolean stream)	Creates a socket and connects it to the specified remote host on the specified remote port.
	Socket(String host, int port, InetAddress localAddr, int localPort)	Creates a stream socket and connects it to the specified port number at the specified IP address.
	Socket(InetAddress address, int port)	Deprecated. Use DatagramSocket instead for UDP transport.
	Socket(InetAddress host, int port, boolean stream)	Creates a socket and connects it to the specified remote address on the specified remote port.
	Socket(InetAddress address, int port, InetAddress localAddr, int localPort)	Creates an unconnected socket, specifying the type of proxy, if any, that should be used regardless of any other settings.
protected	Socket(SocketImpl impl)	Creates an unconnected Socket with a user-specified SocketImpl.

Information hiding

The user of a class (that is, a developer possibly different than the one who developed the class):

- can use the class and operates on it
- does not know how it is coded

Class Date:

Type	Method	Description
boolean	after(Date when)	Tests if this date is after the specified date.
boolean	before(Date when)	Tests if this date is before the specified date.
long	getTime()	Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
String	toString()	Converts this Date object to a String of the form:

We can do non-trivial manipulation of dates (meant as entities) through the class **Date** without knowing its code!

Modularity

The user of a class **knows**:

- which operations exist, how to use them, what they do

He/she **does not know**:

- what's inside the object
- how exactly an operation works

The **state** of the object and the **code** of the class might change,
but the user is **not required to be notified of changes!**

→ **Modularity**: everyone takes care of only some part of the software!

Coding classes

Problem: complex numbers

Goal: manipulating complex numbers

By examples (i.e., point of view of the user):

```
Complex c1 = new Complex(7.46, -3.4567);
Complex c2 = new Complex(0.1, 9.81);

Complex c3 = c1.add(c2);
// same for subtract(), multiply(), divide()

double norm = c2.getNorm();
double angle = c2.getAngle();
String s = c2.toString();
double real = c2.getReal();
double imaginary = c2.getImaginary();
```

"Solution": Complex.java

```
public class Complex {
    private double real;
    private double imaginary;
    public Complex(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }
    public double getReal() {
        return real;
    }
    public double getImaginary() { /* ... */ }
    public Complex add(Complex other) {
        return new Complex(
            real + other.real,
            imaginary + other.imaginary
        );
    }
    /* other methods */
}
```

(IDE: code generation, auto formatting.)

What's behind the solution?

Domain knowledge:

- what is a complex number? what are its composing parts?
(entities)
- what are the required operations?

Knowledge of the programming language:

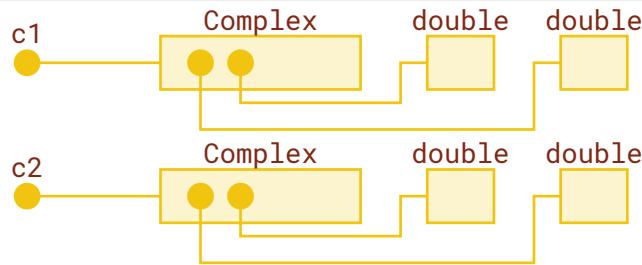
- how to represent the entities in Java?
- how to perform the operations in Java?

You need both!

- you can rely on external help, but you still need to know what you want to do
 - Google/stackoverflow: how to do X in Java?

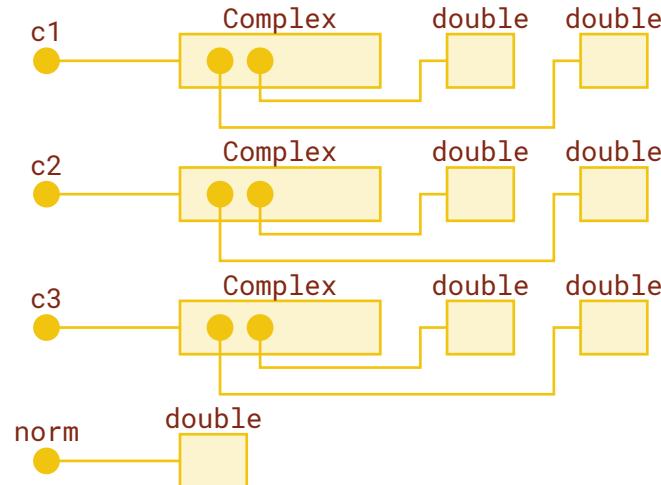
Objects+references diagram

```
Complex c1 = new Complex(7.46, -3.4567);  
Complex c2 = new Complex(0.1, 9.81);
```



Objects+references diagram

```
Complex c1 = new Complex(7.46, -3.4567);
Complex c2 = new Complex(0.1, 9.81);
Complex c3 = c1.add(c2);
double norm = c2.getNorm();
```



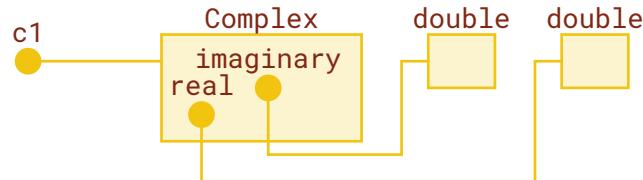
Field

A **field** is an object contained in an object:

```
public class Complex {  
    private double real;  
    private double imaginary;  
    /* ... */  
}  
Complex c1 = new Complex(1.1, 2.2);
```

The fields of an object constitute its **state**.

They are referenced, there is an identifier:



Accessing fields

Can be manipulated using **dot notation**, like methods:

```
public Complex add(Complex other) {  
    return new Complex(  
        real + other.real,  
        imaginary + other.imaginary  
    );  
}
```

Access modifiers: **private**, **public**

Keywords that specify where an identifier is visible, i.e., where it is legal to use it for accessing the field or method (or other, we'll see).

- **private**: visible only within the code of its class
- **public**: visible everywhere

(For brevity, we avoid discussing about syntax: but the Java language specification describes exactly where/how every keyword can be used.)

private fields

File `Complex.java`:

```
public class Complex {  
    private double real;  
    /* here real can be used */  
}  
/* here real can not be used */
```

File `ComplexCalculator.java`:

```
public class ComplexCalculator {  
    public Complex add(Complex c1, Complex c2) {  
        /* here real can not be used */  
    }  
}
```

private or **public**: how to choose?

Ideally, it depends of the **nature of the entity** that the class represents (domain knowledge!)

But (general rule of thumb):

- **fields** should be **private**
 - because they represent the state of the object, and we want to avoid that the state is manipulated from "outside"
- **methods** should be **public**
 - because they are operations that can be performed on the object
 - unless we use a method for describing a "partial" operation that is reused frequently by other public operations: in this case, the method should be **private**

this identifier

It is the identifier of the reference that references the object on which the method, where `this` is, is being executed.

```
public class Complex {  
    public Complex add(Complex other) {  
        return new Complex(  
            this.real + other.real,  
            this.imaginary + other.imaginary  
        );  
    }  
}
```

`this` is a keyword.

Implicit this

Can be omitted:

```
public class Complex {  
    public Complex add(Complex other) {  
        return new Complex(  
            real + other.real,  
            imaginary + other.imaginary  
        );  
    }  
}
```

real and this.real reference the same object.

this for disambiguation

Sometimes it is necessary for disambiguation:

```
public class Complex {  
    private double real;  
    private double imaginary;  
    public Complex(double real, double imaginary) {  
        this.real = real;  
        this.imaginary = imaginary;  
    }  
}
```

Before the line, `real` and `this.real` do not reference the same object.

(This is the typical structure of a constructor: name of input parameters match the name of fields. As usual, the **IDE is your friend** and can write this code automatically!)

Packages

Exported identifiers

A class (i.e., a `.class` file) exports some identifiers of:

- `class` (or classes, but one per `.class` file)
- methods
- fields

Access modifier can be omitted: **default** access modifier:

- `private`: not visible (~ not exported)
- `public`: visible
- default: visible

Package

A **package** is a set of classes with a name

- we'll see how to define it

A package exports the identifiers of its classes:

- **private**: not visible (~ not exported)
- **public**: visible
- default: visible **only within the package**

(There are no many reasons for using the default access modifier; please always specify one between **public** and **private** (or **protected**, we'll see...))

Package name

A sequence of letters separated by dots, e.g.:

- `it.units.erallab`
- `java.util`
- `java.util.logging`

There is no formal hierarchy, nor relationship:

- `java.util.logging` is not a "subpackage" of `java.util`
- they are just different packages

Why and how?

Why package names?

- to avoid name clash that may be likely for classes representing common concepts
 - e.g., `User`, `Point`

How to name packages? (naming conventions)

- usually, (lowercase) reversed institution/company name + product + internal product organization, e.g.,
`it.units.erallab.hmsrobots.objects.immutable`
 - `it.units.erallab`: institution
 - `hmsrobots`: product
 - `objects.immutable`: internal organization

Beyond names:

- packages impact on file organization (we'll see, in directories)

Packages and modules

Since Java 9, a new, higher level abstraction for code entities organization has been introduced: Java Platform Module System (JPMS), briefly **modules**.

- they are interesting and useful, but...
- we'll completely ignore
- you can build rather complex software without knowing them

API documentation

The screenshot shows a web-based API documentation interface. At the top, there is a navigation bar with links: OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar is a search bar labeled "SEARCH:" with a placeholder "Search". The main title of the page is "Java® Platform, Standard Edition & Java Development Kit Version 13 API Specification". A sub-section title "This document is divided into two sections:" is followed by two entries: "Java SE" and "JDK". Each entry has a brief description. Below these sections is a navigation menu with tabs: "All Modules", "Java SE", "JDK", and "Other Modules". Under the "Java SE" tab, there is a table listing a single module: "java.base". The table has two columns: "Module" and "Description". The "Module" column contains "java.base", and the "Description" column contains "Defines the foundational APIs of the Java SE Platform.".

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SEARCH: Search

Java® Platform, Standard Edition & Java Development Kit Version 13 API Specification

This document is divided into two sections:

Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with `java`.

JDK

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with `jdk`.

All Modules **Java SE** **JDK** **Other Modules**

Module	Description
java.base	Defines the foundational APIs of the Java SE Platform.

Fully qualified name

Every class is identified **unequivocally** by its **Fully Qualified Name** (FQN):

FQN = package name + class name

- `java.net.Socket`
- `it.units.erallab.hmsrobots.objects.immutable.Component`

Everywhere in the code, every* class can be identified by its FQN.

```
double r = 10;  
it.units.shapes.Circle circle = new it.units.shapes.Circle(r);  
double area = circle.area();
```

(*: provided it is available; we'll see)

The `import` keyword

Writing always the FQN can make the source code **verbose**, yet it is necessary, otherwise the compiler does not know which class we are referring to.

Solution (shorthand): `import`

```
import it.units.shapes.Circle;

public class ShapeCompound {
    private Circle circle;
    /* ... */
}
```

"`import package.X`" means "dear compiler, whenever I write `X` in this code, I mean `package.X`"

- the compiler internally always uses the FQN

Star import

It might be useful to "import" all the classes in a package:

- it can be done with `import it.units.shapes.*;`

Coding conventions suggest **not** to do it:

- risk: one might import an unneeded class with the same simple name of another class of the same package (e.g.,
`java.awt.Event` and `it.units.timetable.Event`)
- lack of motivation
 - IDEs add `imports` for you **automatically**, and can remove them partially automatically (**autocompletion!**)
 - so don't be lazy and messy

`import` for the developer

`import` is an optimization for the developer:

- recall: code is not just for the machine

`import` does not import code, classes, objects...

Point of view of the compiler

When the compiler processes the source code (`.java`) and finds a class identifier `C`:

- it needs to know its methods and fields (name, signature, modifiers)
- to be able to check if their usage is legit

If without FQN, the compiler looks for `C` definition:

- in the source code (`.java`)
- in the same package (directory)
- in the packages imported with star import

java.lang package

All classes of the `java.lang` packages are available **by default**:

- `import java.lang.*` is implicitly present in every source code

Syntax of FQNs

Package and class name cannot be identified just by looking at the FQN:

E.g., `it.units.UglySw.Point`:

- can be the name of a package
- or can be a FQN where:
 - `Point` is a class
 - or, `UglySw.Point` and `UglySw` are classes (we'll see)
 - or, `units.UglySw.Point`, `units.UglySw`, and `units` are classes, but the first is disrespectful of naming conventions!

Common case and convention: package name are all lowercase!

static field

A **field** of a class C can be defined with the **static** non-access modifier:

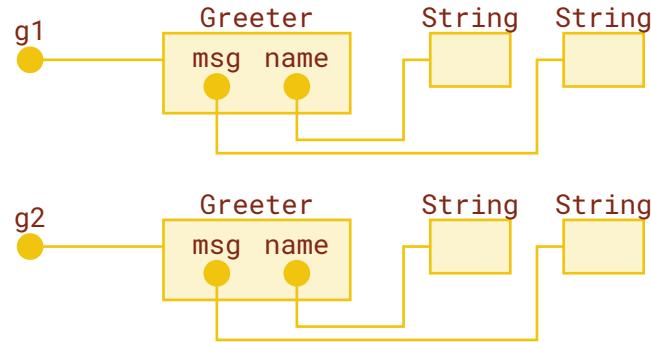
- the corresponding object is **unique**, that is, the same for every instance of C
- it **always** exists, even if no instances of C exist
 - if instantiated

From another point of view:

- the reference to a static field is shared (i.e., the same) among the zero or more instances of the class

static field: diagram

```
public class Greeter {  
    public static String msg;  
    private String name;  
    /* ... */  
}  
  
Greeter g1 = new Greeter();  
Greeter g2 = new Greeter();
```



static method

Also a **method** of a class C can be defined with **static**:

- when invoked, it is not applied on the instance of C
- can be invoked even if no instances of C exist, with a special syntax

The method code cannot access field or use other methods of C that are not **static**:

- they might not exist!
- the compiler performs the check (at compile time)

static method: syntax

```
public class Greeter {  
    private static String msg;  
    private String name;  
    /* ... */  
    public static String sayMessage() {  
        return msg;  
    }  
}  
  
String msg = Greeter.sayMessage(); /* OK! */  
  
Greeter greeter = new Greeter();  
greeter.sayMessage(); /* Syntax is ok, but avoid this form */
```

- `greeter.sayMessage()` is bad because it suggests that the instance `greeter` is somehow involved in this operation, whereas it is indeed not involved!
- only the "class" `Greeter` is involved

When to use `static` for methods?

Conceptually, a method should be `static` if it represents an operation that does not involve an entity represented by an instance of the class:

```
public class Dog {  
    public static Dog getCutest(Dog dog1, Dog dog2) { /* ... */ }  
    public static Dog fromString(String name) { /* ... */ }  
}
```

`static` should be used also for `private` method, when condition above is met, even if they are not visible from outside (but other co-developers see it!)

Hello goal!

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Goal: deep understanding of this code

- not just what it "does"
- but the **role of every part** of the code
 - keywords
 - names

main signature

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

- **public** → **main** has to be invoked "directly" by the JVM upon execution (`java Greeter`): it has to be accessible
- **static** → invokable without having an already existing instance of **Greeter**
- **void** → does not return anything

`public static void main(String[])` is the signature **required** by Java if you want to use the method as an execution **entry point!**

- only the name of the input parameter can be modified

What is `System.out`?

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

- `println` is a method, since it is invoked `()`
- `System.out` might be, in principle:
 1. the FQN of a class (and hence `println` is static in the class `out`)
 2. a field of a class (and hence `out` is static in the class `System`)

There is no package `System` (and `out` would be a class name out of conventions), hence 2 holds: `System` is a class

"Where" is System?

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Where is System?

- more precisely, how can the compiler check that we correctly use `out` field of System?

There is no `import`, hence it can be:

1. there a `System.class` in the same directory of this `.class`,
hence we wrote a `System.java`
2. `System` is in the `java.lang` package, that is always imported by default

2 holds: `java.lang.System` is the FQN of `System`

102 / 419

"What" is **out**?

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Look at System class documentation (fields):

Modifier and type	Field	Description
static PrintStream	err	The "standard" error output stream.
static InputStream	in	The "standard" input stream.
static PrintStream	out	The "standard" output stream.

→ **out** is a field of type **PrintStream** that represents the **standard output**.

(What's the standard output? "Typically this stream corresponds to display output or another output destination specified by the host environment or user.")

What is `println()`? How is it used?

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Look at [PrintStream](#) class documentation (methods):

Type	Field	Description
void	<code>println(int x)</code>	Prints an integer and then terminate the line.
void	<code>println(long x)</code>	Prints a long and then terminate the line.
void	<code>println(Object x)</code>	Prints an Object and then terminate the line.
void	<code>println(String x)</code>	Prints a String and then terminate the line.

"Hello World!" is a string literal and corresponds to `new String("Hello World!")`

- hence `println(String x)` is the used method

Involved classes

```
public class Greeter {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

"Involved" classes:

- defined here: `Greeter`
- defined elsewhere, used here: `System`, `String`, `PrintStream`
 - `System`, `String` in `java.lang`; `PrintStream` in `java.io`

No `import`

No need to import `PrintStream`:

- `import` does not "load code"
- `import` says to the compiler that we will use a simple name for **FQN in the source code**
 - but we do not use `PrintStream` in the source code
 - there is an `import` for `PrintStream` in `System.java` (or FQN)

```
import java.io.PrintStream;
/* ... */

public class System {
    public static PrintStream out;
    /* ... */
}
```

Write your first class
(and some other key concept)

Goal

Write an application that, given a word w and a number n , gives n anagrams of w .

Natural language and specification

"Write an application that, given a word w and a number n , gives n anagrams of w ."

Natural language is ambiguous: this description leaves a lot of choices (and hence **responsability**) to the designer/developer:

- "an application": for which platform? are there technological constraints? (tech)
- "given": how? command line? file? standard input? (tech)
- "a word $w- "a number $n- "anagrams": what is an anagram? (domain)
- " n anagrams": what if there are no enough anagrams? which ones, if more than n ? (domain)$$

It's up to you to take these decisions!

More precise goal

In this particular case:

- a Java application, i.e., a class with a `main()`
- w via standard input, n via command line (customer)
- a word is a non-empty sequence of word characters (regex `[A-Za-z]+`)
- n is natural
- anagram:
 - in general "a word or phrase formed by rearranging the letters of a different word or phrase" (from [Wikipedia](#))
 - here (customer): [permutation of multisets](#) with repetitions
- show up to n , whichever you prefer

Basic building blocks

for achieving the goal

Execution flow control

Usual constructs available:

- `if then else`
- `while`
- `for`*
- `switch`*
- `break`
- `continue`
- `return`

I assume you know them!

(*: with enhanced syntax that we will see later)

Basic I/O

Where:

- input from standard input (aka stdin, typically, "the keyboard")
- output to standard output (aka stdout)

Of what:

- **String**
- primitive types

(We'll see later how to do I/O of other things to/from other places)

Output to standard output

Use `System.out`, static field `out` of class `java.lang.System`: it's of class `java.io.PrintStream`

Type	Field	Description
void	<code>println()</code>	Terminates the current line by writing the line separator string.
void	<code>println(boolean x)</code>	Prints a boolean and then terminate the line.
void	<code>println(char x)</code>	Prints a character and then terminate the line.
void	<code>println(char[] x)</code>	Prints an array of characters and then terminate the line.
void	<code>println(double x)</code>	Prints a double and then terminate the line.
void	<code>println(float x)</code>	Prints a float and then terminate the line.
void	<code>println(int x)</code>	Prints an integer and then terminate the line.
void	<code>println(long x)</code>	Prints a long and then terminate the line.
void	<code>println(Object x)</code>	Prints an Object and then terminate the line.
void	<code>println(String x)</code>	Prints a String and then terminate the line.

The same for `print()`, that does not terminate the line.

(And a very practical `printf()`, that we'll discuss later)

Input from stdin with BufferedReader

```
BufferedReader reader = new BufferedReader(  
    new InputStreamReader(System.in)  
);  
/* ... */  
String line = reader.readLine();
```

- `readLine()` reads one line at once
 - Java takes care of using the proper line termination criterion (`\n` or `\r\n`) depending on the host OS [how does it know?]
- directly reads only strings
- requires adding `throws Exception` after the `()` in the signature of `main`
 - we ignore why, for now

Input from stdin with Scanner

```
Scanner scanner = new Scanner(System.in);
/* ... */
String s = scanner.next();
int n = scanner.nextInt();
double d = scanner.nextDouble();
```

- `next()`, `nextInt()`, ... do three things:
 1. reads one line from the `InputStream` it's built on
 2. splits the line in tokens (`sep = " "`)
 3. converts and returns the first token of proper type
- if the line "has" >1 tokens, they are consumed in subsequent calls of `next*()`

String to primitive type

Here, useful if you use `BufferedReader` for input.

```
String line = reader.readLine();
int n = Integer.parseInt(line);
```

Class [Integer](#)

Modif. and type	Field	Description
static int	parseInt(String s)	Parses the string argument as a signed decimal integer.

Similar for other primitive types:

- `Float.parseFloat(String)`
- `Double.parseDouble(String)`
- ...

Arrays

Array: fixed-length **sequence** of objects of the **same type**

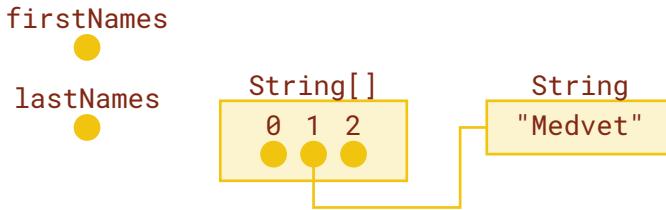
- each object is accessed with the operator [] applied to the reference to the array
- 0-based indexing

```
String[] firstNames;  
String[] lastNames = new String[3];  
lastNames[1] = new String("Medvet");
```

- Creates a reference **firstNames** to **String[]**; does not create the array; does not create the elements
- Creates a reference **lastNames** to **String[]**; create an array of 3 elements; does not create the elements
- Creates a **String** and makes **lastNames[1]** (2nd in the array) reference it

Diagram

```
String[] firstNames;  
String[] lastNames = new String[3];  
lastNames[1] = new String("Medvet");
```



Conventions

Name of arrays (i.e., identifiers of references to arrays):

- plural form of the corresponding reference
 - `Person[] persons, Person[] employees, ...`

Definition:

- `Person persons[]` is the same of `Person[] persons`, but the latter **is much better**:
 - it makes evident that the **type** is an array, rather than the identifier

Array creation

```
String[ ] dogNames = {"Simba", "Gass"};  
//same of new String[]{ "Simba", "Gass"}
```

is the same of

```
String[ ] dogNames = new String[2];  
dogNames[0] = "Simba"; //same of = new String("Simba");  
dogNames[1] = "Gass";
```

Array lenght

The type array has a field `length` of type `int` that contains the array size:

- never changes for the a given array
- cannot be written

```
String[ ] dogNames = {"Simba", "Gass"};
System.out.println(dogNames.length); //prints 2
dogNames = new String[3];
System.out.println(dogNames.length); //prints 3
```

`dogNames.length = 5;` does not compile!

Iterating over array elements

"Traditional" **for** syntax:

```
String[] dogNames = {"Simba", "Gass"};
for (int i = 0; i<dogNames.length; i++) {
    System.out.println("Dog " + i + " name is " + dogNames[i]);
}
```

Enhanced **for** (or for-each) syntax:

```
String[] dogNames = {"Simba", "Gass"};
for (String dogName : dogNames) {
    System.out.println("A dog name is " + dogName);
}
```

- the index is not available inside the loop
- can be applied to other types too (we'll see)

Varargs

In a signature of a method, the **last input parameter**, if of type array, can be specified with the `...` syntax instead of `[]`:

```
public static double max(double... values) { /* 1 ... */}
```

From the inside, exactly the same of `[]`:

```
public static double max(double[] values) { /* 2 ... */}
```

From the outside, i.e., where the method is invoked, `...` enables invocation with variable number of parameters (of the same type):

```
double max = max(4, 3.14, -1.1); //values ≈ double[3]; OK for 1  
max = max(); //values ≈ double[0]; OK for 1  
max = max(new double[2]{1, 2}); //Ok!; OK for 1 and 2
```

(Since Java 5.0. Mathematically speaking, varargs allows to define **variadic functions**.)

About modifiers

```
public static double max(double... values) {  
    double max = values[0];  
    for (int i = 1; i<values.length; i++) {  
        max = (max > values[i]) ? max : values[i];  
    }  
    return max;  
}
```

(condition ? expr1 : expr2 is the ternary operator.)

- [Of which class might be a method?]
- [Why `static`?]
- [What happens with `max()`?]

Why only last input parameter?

```
// does NOT compile!
public static int intersectionSize(String... as, String... bs) {
    /* ... */
}

intersectionSize("hello", "world", "cruel", "world");
```

What is **as** and what is **bs**?

- undecidable!

Java designers could have allowed for some exceptional case that, under some conditions, are not misinterpretable:

- `method(ClassA..., ClassC)`
- `method(ClassA..., ClassB...)`

But they opted for **clarity at the expense of expressiveness**.

Command line arguments

Available as content of `main` only arguments:

```
public class ArgLister {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.println(arg);  
        }  
    }  
}
```

```
eric@cpu:~$ java ArgLister Hello World  
Hello  
World
```

(Possible limitations and syntax variations depending on the host OS)

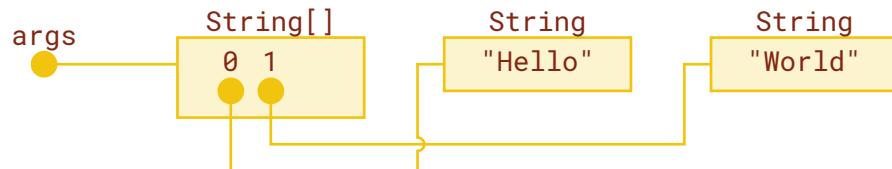
Diagram

```
public class ArgLister {  
    public static void main(String[] args) {  
        for (String arg : args) {  
            System.out.println("Arg: " + arg);  
        }  
    }  
}
```

java prepares args before invoking main()

At the beginning of main() after java ArgLister Hello World

[What inside the 1st and 2nd iteration of the for loop?]



Operating with Strings

Creation (`String` constructors):

- empty: `String s = new String();`
- specified: `String s = new String("hi!");`
 - the same of `String s = "hi!";`
- same of another: `String s = new String(otherString);`
- ...

String methods

Many!

A few examples from the javadoc of [String](#)

Modif. and type	Field	Description
int	compareTo(String anotherString)	Compares two strings lexicographically .
int	CompareToIgnoreCase (String str)	Compares two strings lexicographically, ignoring case differences.
boolean	endsWith(String suffix)	Tests if this string ends with the specified suffix .
int	indexOf(int ch)	Returns the index within this string of the first occurrence of the specified character.
int	indexOf(int ch, int fromIndex)	Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
int	indexOf(String str)	Returns the index within this string of the first occurrence of the specified substring.
int	indexOf(String str, int fromIndex)	Returns the index within this string of the first occurrence of the specified substring , starting at the specified index.
int	length()	Returns the length of this string.
boolean	matches(String regex)	Tells whether or not this string matches the given regular expression.
String	replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement.

String.format()

Modif. and type	Field	Description
	static String format(String format, Object... args)	Returns a formatted string using the specified format string and arguments.

See also [Formatter](#) class and [printf\(\)](#) in [PrintStream](#) class; [here](#) the syntax.

```
String reference = String.format(  
    "FPR=%4.2f\tFNR=%4.2f%n",  
    fp / n,  
    fn / p  
);  
//results in FPR=0.11 FNR=0.10
```

We'll see why [Object...](#)

Strings are immutable

1st sentence of the 2nd para of [String](#) docs: "Strings are **constant**; their values **cannot be changed** after they are created."

Apparently the docs are self-contradictory:

Modif. and type	Field	Description
String	concat(String str)	Concatenates the specified string to the end of this string.
String	toUpperCase()	Converts all of the characters in this String to upper case using the rules of the default locale.

Apparently!

String.concat()

OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Module java.base
Package java.lang

Class String

java.lang.Object
 java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

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

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For

Immutable **String**s: diagram

```
String s1, s2;
s1 = "hamburger";
s2 = s1.substring(3, 7);
System.out.print("s1 = ");
System.out.println(s1);
s1 = s1.replace('a', 'b');
String[] names = { "John", "Fitzgerald", "Kennedy" }
String firstInit, middleInit, lastInit;
firstInit = names[0].substring(0, 1);
middleInit = names[1].substring(0, 1);
lastInit = names[2].substring(0, 1);
firstInit.concat(middleInit).concat(lastInit);
```

Draw the diagram

- after the line starting with **String[]**
- after the last line

Concatenation

Besides `concat()`, `+` operator (alternative syntax):

```
String s1 = "today is ";
String s2 = s1.concat(" Tuesday").concat("!");
```

2nd line is the same of:

```
String s2 = s1 + " Tuesday" + "!" ;
```

`+` is associative on the left:

- `"today is Tuesday"` is obtained before `"today is Tuesday!"`

String + other type

If one of the operands of `+` is of type `String` then, at runtime:

- a `String` representation of the operand is obtained
- the concatenation of the two `Strings` is done

```
int age = 41;  
String statement = "I'm " + age + " years old";
```

[What are the `String` and the non-`String` operands?]

For the case when the non-`String` operand is not a primitive type, we'll see the underlying mechanism in detail.

Primitive types

A few differences with respect to classes:

- they are not created with `new`
- they do not have methods (no constructors) and fields
 - no dot notation

They can be operands of (binary) operators.

```
int n = 512;
double d = 0d;
char a = 'a';
boolean flag = false;
flag = flag || true;
double k = 3 * d - Math.sqrt(4.11);
```

[sqrt() modifiers?]

Initialization of primitive types

If not explicitly initialized:

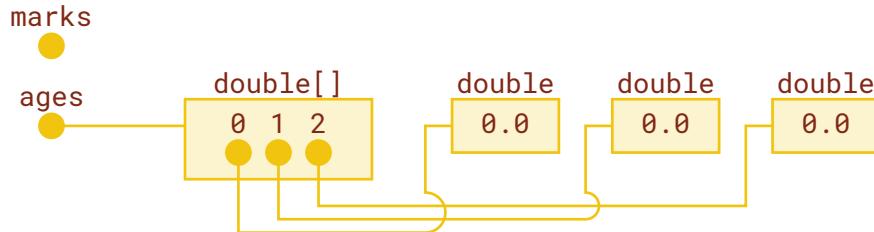
- if field of an object, initialized to default value
 - 0 for numbers, `false` for `boolean`
- if local variable, code does not compile

Initialization of arrays of primitive types

Consistently, for arrays:

- elements of primitive type arrays are initialized to default value

```
int[] marks;  
double[ ] ages = new double[3];
```



Inline field initialization

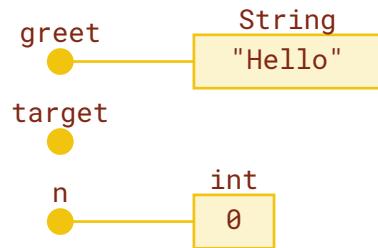
Fields can be initialized inline:

```
public class Greeter {  
    private String greet = "Hello";  
    public void greet() {  
        System.out.println(greet + " World!");  
    }  
}
```

Field initialization is executed before the first statement of any constructor.

Field initialization

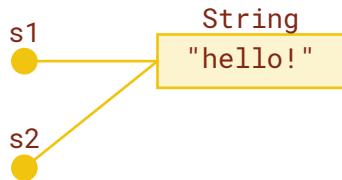
```
public class Greeter {  
    private String greet = "Hello";  
    private String target;  
    private int n;  
    public void greet() {  
        System.out.println(greet + " " + target);  
        System.out.println(n);  
    }  
}
```



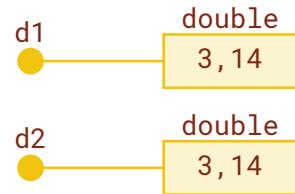
= on primitive types

Assign operator = copies the content, instead of referencing the same object.

```
String s1 = "hello!";  
String s2 = s1;
```



```
double d1 = 3.14;  
double d2 = d1;
```



(Things are a bit more complex, we'll see...)

null

A special "value" that can be referenced by a reference of any non-primitive type.

Fields:

- non-primitive fields are implicitly set to `null`
- recall: primitive fields are set to their default values

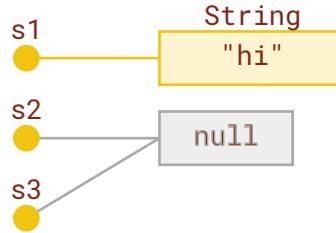
Local variables:

- if not initialized, compilation error!

Referencing null

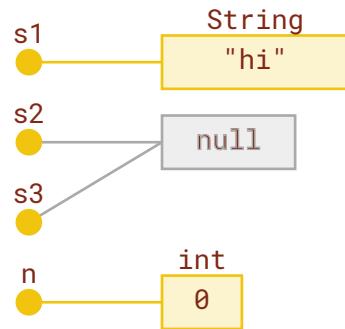
A reference referencing `null` is (approx.) not referencing anything.

```
String s1 = "hi";
String s2 = null;
String s3 = null;
```



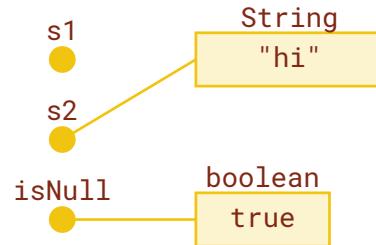
(`null` is not of any particular type; we will omit gray part of the diagram)

```
public class Greeter {
    private String s1 = "hi";
    private String s2 = null;
    private String s3;
    private int n;
}
```



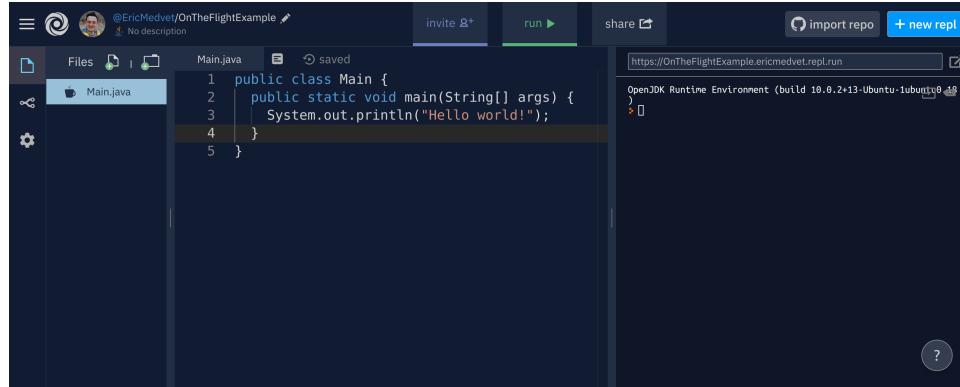
Dereferencing

```
String s1 = "hi";
String s2 = s1;
s1 = null;
boolean isNull = s1 == null;
```



Ready for first exercise!

Online IDE repl.it



The screenshot shows the repl.it web-based IDE. At the top, there's a header with user information (@EricMedvet/OnTheFlightExample), a 'invite' button, a 'run' button, a 'share' button, an 'import repo' button, and a '+ new repl' button. Below the header is a file navigation bar with 'Files' selected, showing a file named 'Main.java'. The code editor contains the following Java code:

```
public class Main {  
    public static void main(String[] args) {  
        System.out.println("Hello world!");  
    }  
}
```

To the right of the code editor is a runtime environment window titled 'OpenJDK Runtime Environment (build 10.0.2+13-Ubuntu-1ubuntu10.0.2)' which displays the output of the executed code: 'Hello world!'. The URL 'https://OnTheFlightExample.ericmedvet.repl.run' is visible at the top of this window.

- register
- enjoy compilation (**javac**) and execution (**java**)
 - note the run button

Code neatly!

Your code is **your representation** of a problem and its solution.

When someone looks at your code, she/he is entering your mind:

- messy code → messy representation → messy mind →
messy you!
- **do you want to look messy?**
 - to your collaborator, your boss, your employee
 - to your teacher

Code review

At least 4 sources of mess (increasingly harder to spot):

- code formatting
- naming
- implementation of algorithm
- algorithm

Code review → the process of checking the code by looking at the **source code**:

- routinely performed within big tech companies
- there is a "IEEE Standard for Software Reviews and Audits" (IEEE 1028-2008)
- not cheap!

Anagrams (~2h, 1st home assignement)

Write an application that, given a word w and a number n , gives n anagrams of w . (Go to [detailed specifications](#), also in next slide)

- multiset of permutations of multiset (i.e., with repetitions)
- with capitalized anagrams

Hints:

- take inspiration from the Internet
 - but please make at least the effort of finding the proper search query
- if you already master "advanced" Java features, try to ignore them, at least initially

If/when done:

1. redo it on a full, desktop IDE
2. profile your code

More precise goal

In this particular case:

- a Java application, i.e., a class with a `main()`
- w via standard input, n via command line (customer)
- a word is a non-empty sequence of word characters (regex `[A-Za-z]+`)
- n is natural
- anagram:
 - in general "a word or phrase formed by rearranging the letters of a different word or phrase" (from [Wikipedia](#))
 - here (customer): [permutation of multisets](#) with repetitions
- show up to n , whichever you prefer

Inheritance and polymorphism

Printing a Date

```
import java.util.Date;  
  
public class Greeter {  
    public static void main(String[] args) {  
        System.out.print("Hello World! Today is: ");  
        System.out.println(new Date());  
    }  
}
```

```
eric@cpu:~$ java Greeter  
Hello World! Today is: Mon Mar 16 10:36:13 UTC 2020
```

It works!

println and Date

PrintStream class:

Type	Field	Description
void	println()	Terminates the current line by writing the line separator string.
void	println(boolean x)	Prints a boolean and then terminate the line.
void	println(char x)	Prints a character and then terminate the line.
void	println(char[] x)	Prints an array of characters and then terminate the line.
void	println(double x)	Prints a double and then terminate the line.
void	println(float x)	Prints a float and then terminate the line.
void	println(int x)	Prints an integer and then terminate the line.
void	println(long x)	Prints a long and then terminate the line.
void	println(Object x)	Prints an Object and then terminate the line.
void	println(String x)	Prints a String and then terminate the line.

No `println` for Date!

- why does the code compile?
- why and how does it execute?
 - what `println` is invokated at runtime?
 - why the printed string makes sense? (`Mon Mar 16 10:36:13 UTC 2020` is actually a date!)

Similarly

```
import java.util.Date;

public class Greeter {
    public static void main(String[] args) {
        System.out.println("Hello World! Today is: " + new Date());
    }
}
```

```
eric@cpu:~$ java Greeter
Hello World! Today is: Mon Mar 16 10:36:13 UTC 2020
```

+ operator on **String** and **Date**:

- why does it compile?
- why and how does it execute?

Inheritance

Inheritance

It is **static** property of the language

- it has effect at compile-time

It allows to **define a new type A starting from existing type B:**

- only new or changed parts (methods/fields) are defined

Syntax

```
public class Derived extends Base {  
    /* ... */  
}
```

We (the developer) mean: "Dear compiler":

- this **Derived** class is identical to the **Base** class
- I'll define **new** fields and methods
 - methods and fields cannot be hidden (or "removed")
- I'll **redefine** existing fields and methods
 - for methods, by re-using the very same signature

We say that **Derived** extends (or **inherits** from, or **derives** from)
Base.

Object

Every class implicitly derives **Object** (when not explicitly deriving from another class):

- every class has the methods and fields of **Object**

```
public class Greeter {  
    /* ... */  
}
```

is the same of:

```
public class Greeter extends Object {  
    /* ... */  
}
```

"Surprisingly", **Object** is a class, not an object...

Inheritance tree

A class can inherit from another class, that inherits from another class, ...

```
public class EnhancedGreeter extends Greeter { /* ... */ }  
public class Greeter { /* ... */ }
```

EnhancedGreeter has methods and fields of Greeter **and** of Object.

Since a class can be derived from many classes, inheritance relationships form a tree.

Inheritance in the documentation

Package `java.io`

Class Reader

`java.lang.Object`
`java.io.Reader`

Package `java.io`

Class BufferedReader

`java.lang.Object`
`java.io.Reader`
`java.io.BufferedReader`

`Reader` has all the fields and methods of `Object`

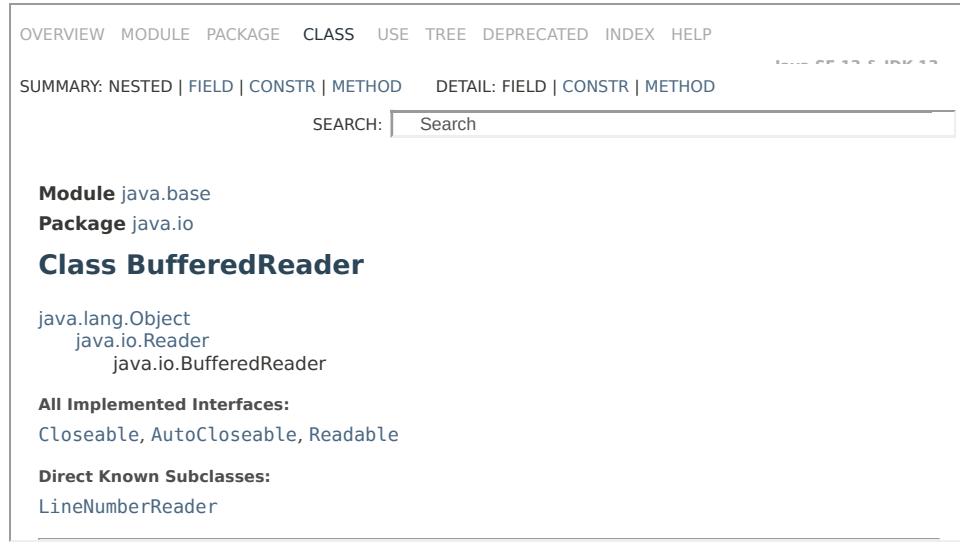
- it may have some more methods and/or fields
- some methods may behave differently

`BufferedReader` has all the fields and methods of `Reader` and `Object`

- it may have some more methods and/or fields
- some methods may behave differently

What methods are inherited?

Specified in the doc ("methods declared in class ..."):



The screenshot shows a JavaDoc page for the `BufferedReader` class. At the top, there's a navigation bar with links for OVERVIEW, MODULE, PACKAGE, CLASS (which is bolded), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, it says "SUMMARY: NESTED | FIELD | CONSTR | METHOD" and "DETAIL: FIELD | CONSTR | METHOD". There's also a search bar labeled "SEARCH: Search".

Module `java.base`
Package `java.io`

Class BufferedReader

Inheritance hierarchy:
java.lang.Object
 java.io.Reader
 java.io.BufferedReader

All Implemented Interfaces:
`Closeable, AutoCloseable, Readable`

Direct Known Subclasses:
`LineNumberReader`

Point of view of Base developer

```
public class Base {  
    /* my stuff! */  
}
```

I **do not need to know** if, how, when, who, where, (and why) **will derive** Base!

But still write clean code!

E.g., **Object** developers did not know we would have been writing some Java code right now

Point of view of Derived developer

```
public class Derived extends Base {  
    /* my incremental stuff! */  
}
```

I do not need to have the source code of **Base**

- to compile, nor to execute
- I need the **.class**, though

E.g., "no one" needs the source code of **Object**!

Constructor and inheritance

```
public class Derived extends Base {  
    public Derived() {  
        /* init things */  
    }  
}
```

- Who (which code) initializes inherited fields (those of **Base**)?
- When?

Note (before the answers) that:

- **Base** fields might be **private**: what code other than that of **Base** can operate on them?
- suppose that a **Derived** method is called within the **Derived()**:
 - it can use **Base** methods (it's legit)
 - those methods can use **Base** fields: who should be the responsible for their initialization?

Constructor derived class

```
public class Derived extends Base {  
    public Derived() {  
        Base();  
        /* init things */  
    }  
}
```

The **compiler** inserts an (implicit, wrt to code) call to derived class constructor **Base()**!

- Who (which code) initializes inherited fields (those of **Base**)?
 - **Base()**
- When?
 - before any statement of **Derived()**

Many constructors/no default constructor

What if `Base` does not have the no-args constructor?

The compiler requires the developer to specify which `Base` constructor to use

- and its parameters

Syntax: `super(...)`

super() constructor

```
public class Greeter {  
    public Greeter(String name) {  
        /* init things */  
    }  
}  
  
public class EnhancedGreeter extends Greeter {  
    public EnhancedGreeter(String firstName, String lastName) {  
        super(firstName + lastName);  
        /* init things */  
    }  
}
```

Ok!

- code compiles

super() constructor: not working

```
public class Base {  
    public Base(int n) { /* ... */ }  
    public Base(double v) { /* ... */ }  
}
```

```
public class Derived extends Base {  
    private int m;  
    public Derived() {  
        m = 0;  
        super(5);  
        /* ... */  
    }  
}
```

Not ok!

- does not compile: `super()` has to be invoked first

super() constructor: not working

```
public class Base {  
    public Base(int n) { /* ... */ }  
    public Base(double v) { /* ... */ }  
}
```

```
public class Derived extends Base {  
    private int m;  
    public Derived() {  
        m = 0;  
        /* ... */  
    }  
}
```

Not ok!

- does not compile: `super()` with no args does not exist

super() constructor: working

```
public class Base {  
    public Base(int n) { /* ... */ }  
    public Base(double v) { /* ... */ }  
    public Base() { /* ... */ }  
}
```

```
public class Derived extends Base {  
    private int m;  
    public Derived() {  
        m = 0;  
        /* ... */  
    }  
}
```

Ok!

- code compiles

[Which constructor of `Base` is invoked?]

Inheritance and inline initialization

```
public static class Base {  
    public int n = 1;  
}
```

```
public static class Derived extends Base {  
    public int m = n + 1;  
}
```

```
Derived derived = new Derived();  
System.out.printf("m=%d%n", derived.m); // -> m=2
```

"Field initialization is executed before the first statement of any constructor." → any of **this** class!

- **super()** is executed before any inline initialization

[What is `derived.n`?]

Polymorphism

Reference type and inheritance

```
public class Derived extends Base { /* ... */ }
```

Any code that works with a **Base** can work also with a **Derived**.

- more formally: a reference to type **Base** can (i.e., the compiler says ok) reference an object of type **Derived**

```
public void use(Base base) { /* ... */ }
```

```
Base base = new Derived(); // OK!!!
```

```
use(new Base()); // OK, sure!  
use(new Derived()); // OK!!!
```

Using Derived

```
public void use(Base base) { /* ... */ }  
use(new Derived()); // OK!!!
```

Why does it compile? (`use()` was written to work with a `Base`)

- `Derived` has all fields and methods of `Base`
 - maybe with different behaviors (methods, but signature is exactly the same)
 - maybe it has also other methods/fields
- → any dot notation valid on a `Base` is also valid on a `Derived!`

("has all fields", but recall visibility!)

Inheritance, philosophycally

Derived has all fields and methods of Base.

Syllogism: anything being a Derived is also a Base

```
public class LivingBeing { /* ... */ }

public class Animal extends LivingBeing { /* ... */ }

public class Mammal extends Animal { /* ... */ }

public class Dog extends Mammal { /* ... */ }

public class Chihuahua extends Dog { /* ... */ }
```

A Chihuahua is a Dog, a Mammal, ..., and an Object.

(But, please, avoid over-using inheritance.)

(There's a [debate](#) about inheritance being **good** or **bad**. Indeed, this is not the most appropriate example of using inheritance...)

Be general

Rule: use the **most general** definition!

```
public void putLeash(Dog dog) { /* ... */ }
```

is better than:

```
public void putLeash(Chihuahua dog) { /* ... */ }
```

even if in your code you will `putLeash()` only on `Chihuahuas` (but provided that you have the inheritance `Dog → Chihuahua`).

... but not too much

Similarly:

```
public Milk milk(Mammal mammal) { /* ... */ }
```

is better than:

```
public Milk milk(Cow cow) { /* ... */ }
```

However:

```
public Milk milk(Animal animal) { /* ... */ }
```

is wrong!

- but (as is here) compiles!
- wrong, because conceptually you cannot milk every animal

Removing methods?

```
public class Derived extends Base { /* ... */ }
```

Why cannot **Derived** remove some methods/fields of **Base**?

- more specifically, why the language does not even have a construct for doing this?

Why cannot **Derived** reduce visibility of methods/fields of **Base**?

- more specifically, why the compiler does not accept it?

Because otherwise "Derived has all fields and methods of Base"
would not be true!

- so we could not write `use(new Derived())`, nor `println(new Date())`, and inheritance would be useless!

Same code, difference behavior

```
public void greet(Person person) {  
    System.out.println("Hi, " + person.getFullName());  
}
```

```
public class Person {  
    /* ... */  
    public String getFullName() {  
        return firstName + " " + lastName;  
    }  
}
```

```
public class Doc extends Person {  
    /* ... */  
    public String getFullName() {  
        return "Dr. " + firstName + " " + lastName;  
    }  
}
```

Outcome is different with the same `greet()` and same fields content!

Polymorphism

It is a **dynamic** consequence of inheritance:

- it has effect at runtime

It results in the same code to have different behaviors depending on the type of the parameter.

Different behavior

```
public class Derived extends Base { /* ... */ }
public void use(Base base) { /* ... */ }
Base base = new Derived();
```

Outcome might be "different" than that with `= new Base()`.

```
use(new Base()); // OK, sure!
use(new Derived()); // OK!!!
```

Outcomes might be "different".

Developer of `use()` did not know that someone would have derived `Base`:

- they knew how to use a `Base`
- who wrote `Derived` **decided** that any `Derived` is a `Base`

Not the opposite!

Obviously...

```
public Base createBase() { /* ... */ }
```

```
Derived derived;  
derived = createBase();
```

1. `Derived derived`: developer → compiler
 - please, take note (and make sure) that with any operation defined in `Derived` can be applied to object referenced by `derived`
2. `derived = createBase()`: compiler → developer
 - no! I cannot meet the requirement ("make sure that") because the object returned by `createBase()` might not be a `Derived`

It might also be a `Derived`, but cannot guarantee...

Might be...

```
public Base createBase() {  
    if (System.currentTimeMillis() % 2 == 0) {  
        return new Base();  
    }  
    return new Derived();  
}
```

(Purely evil! Don't do it at home!)

```
Derived derived;  
derived = createBase();
```

`createBase()` can really return a `Derived`...

What if I (developer) am sure that it is a `Derived`?

Downcasting

```
Derived derived;  
derived = (Derived) createBase();
```

What if I (developer) am sure that it is a **Derived**?

Syntax: **(Derived) base**

- "changes" type of downcast reference
- works only if **Derived** derives (directly or indirectly) from the class of **base**

Why "works only..."?

- because **return** in **createBase()** **must** take a **Base**, so there's no reason to accept something that cannot happen!

Still, at runtime, there is a check at every invocation.

instanceof

Binary operator (keyword) for checking the type of a object at runtime:

- evaluates to `boolean`

```
boolean isString = ref instanceof String;
```

Formally, `a instanceof B` evaluates to `true` if and only if object referenced by `a` might be legitimately referenced by a reference `b` to objects of type `B`.

```
public class Derived extends Base { /* ... */ }

Derived derived = new Derived();
boolean b1 = derived instanceof Object; // -> true
boolean b2 = derived instanceof Base; // -> true
boolean b3 = derived instanceof Derived; // -> true
```

Typical usage

```
Object obj = someOddMethod();
if (obj instanceof String) {
    String string = (String) obj;
    System.out.println(string.toUpperCase());
    // obj.toUpperCase() would not compile!
}
```

Since JDK 14 (March 2020):

```
Object obj = someOddMethod();
if (obj instanceof String string) {
    System.out.println(string.toUpperCase());
}
```

Just a language shorthand! (Called Pattern matching with `instanceof`)

Why?

```
import java.util.Date;

public class Greeter {
    public static void main(String[] args) {
        System.out.print("Hello World! Today is: ");
        System.out.println(new Date());
    }
}
```

- Why does the code compile?
- Why and how does it execute?
 - What `println` is invokated at runtime?
 - Why the printed string makes sense?

Why does it compile?

`PrintStream` class:

Type	Field	Description
void	<code>println(Object x)</code>	Prints an Object and then terminate the line.

`Date` extends `Object` and there is a `PrintStream.println(Object)`!

- a `Date` has everything an `Object` has
- `println()` says it knows how to print on `Object`

Ok, but then?

toString()

Object class:

Type	Field	Description
String	toString()	Returns a string representation of the object.

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Hence:

- every object has a `toString()` method
- every object can be represented as string

Overriding `toString()`

In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The developer of a class may (and should!) redefine (= **override**) `toString()`:

- returns a string that "textually represents" this object
- concise but informative
- easy for a person to read

IDEs have an option for writing a decent `toString()` for you
(**Alt+Ins** in Netbeans)

toString() of Date

```
public String toString()
```

Converts this Date object to a String of the form:

dow mon dd hh:mm:ss zzz yyyy

where:

- dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat).
- mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).
- dd is the day of the month (01 through 31), as two decimal digits.
- hh is the hour of the day (00 through 23), as two decimal digits.
- mm is the minute within the hour (00 through 59), as two decimal digits.
- ss is the second within the minute (00 through 61), as two decimal digits.
- zzz is the time zone (and may reflect daylight saving time). Standard time zone abbreviations include those recognized by the method parse. If time zone information is not available, then zzz is empty - that is, it consists of no characters at all.
- yyyy is the year, as four decimal digits.

Why does it work?

```
import java.util.Date;

public class Greeter {
    public static void main(String[] args) {
        System.out.print("Hello World! Today is: ");
        System.out.println(new Date());
    }
}
```

In PrintStream:

```
public void println(Object x) {
    if ( x == null ) {
        println("null");
    } else {
        println(x.toString());
    }
}
```

Delegation and information hiding

```
public void println(Object x) {  
    if ( x == null ) {  
        println("null");  
    } else {  
        println(x.toString());  
    }  
}
```

The developers of `PrintStream` just took care of how to print an `Object`

- check for `null` inside `println()`
- delegation to `Object toString()` for the rest

No need for knowledge about if, how, when, who, where, (and why) would have derived `Object`!

Why?

```
System.out.println("Now is " + new Date());
```

Why does the code compile?

- because `+` with a `String` as 1st operand "accepts" anything as 2nd operand

Why and how does it execute?

- because the compiler translated `+` `ref` to something like

```
(ref == null) ? "null" : ref.toString()
```

(if `ref` is not a reference to primitive type)

How does polymorphism work?

(briefly)

Own class "field"

At runtime, every object `o` has a description of its class:

- it is an object of class `Class`
- it is "shared" among all instances of the class of `o`
 - approx. like a `static` field

It can be obtained with the `getClass()` method of `Object`:

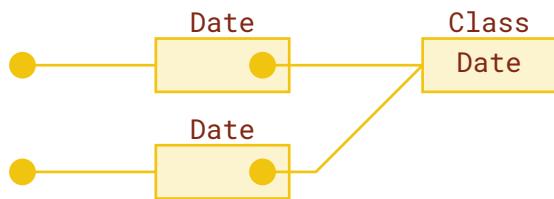
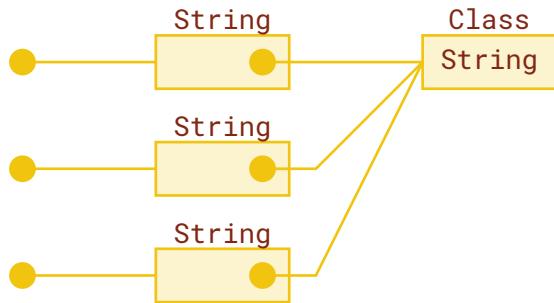
- hence, since it is in `Object`, every `o` has `getClass()`
 - since every object has all the methods declared in `Object`

`Object` class:

Type	Field	Description
<code>Class<?></code>	<code>getClass()</code>	Returns the runtime class of this <code>Object</code> .

(Ignore the `<?>` for now; we'll see it later)

Diagram



What's inside the description?

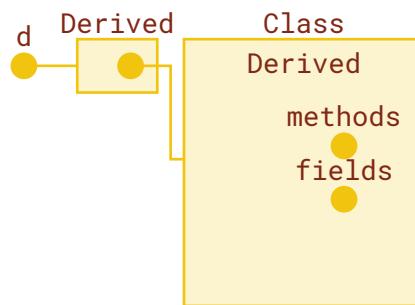
Conceptually, inside the **Class** of a class X:

- methods of X (i.e., **declared in X**)
- fields of X
- ...

(We'll see later more about this!)

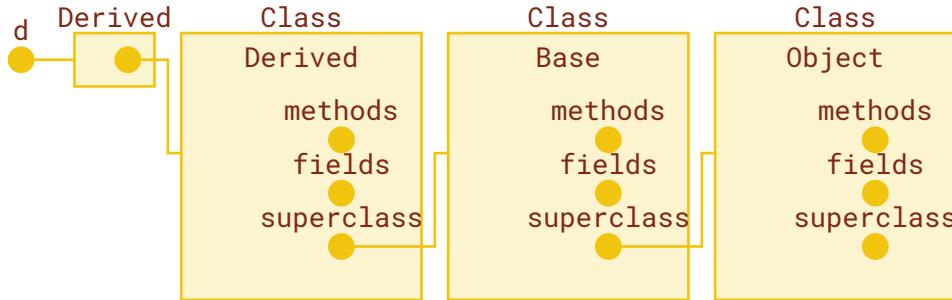
```
public class Base {  
    public void doBase() { /.../ }  
}  
  
public class Derived extends Base {  
    public void doDerived() { /.../ }  
}  
  
Derived d = new Derived();
```

methods contains **doDerived**, not
doBase



Reference to superclass

- and the class of the base type, possibly `Object`



- `methods` of `d.getClass()` contains `doDerived`
- `methods` of `d.getClass().superclass` contains `doBase`
- `methods` of `d.getClass().superclass.superclass` contains `toString, getClass, ...`

(Approximately: types and names of fields/methods are different)

Which method?

```
System.out.println(derived); // -> derived.toString()
```

```
public void println(Object x) {
    if (x == null) {
        println("null");
    } else {
        println(x.toString());
    }
}
```

Conceptually:

1. get `Class c` of `derived`
2. get methods of `c`
3. do methods contains `toString()`
 - yes, use that method
 - no, set `c` to `c.superclass` and go back to 2

In the worst case, `c` is eventually the `Class` describing `Object` 201 / 419

Class and Object

`Class` is a class; `Object` is a class.

- you can use `object` as identifier of a reference to objects of class `Object`
- you **cannot** use `class` as identifier of a reference to objects of class `Class`
 - because `class` is a keyword

(Many developers and examples use `clazz...`)

Methods of Object

Object class:

Type	Field	Description
boolean	equals(Object obj)	Indicates whether some other object is "equal to" this one.
Class<?>	getClass()	Returns the runtime class of this Object.
String	toString()	Returns a string representation of the object.

`equals()`: indicates whether some other object is "equal to" this one.

What's the difference with respect to `==`?

`==` (and `!=`)

`a == b`

- if `a` and `b` of different primitive types**, or of primitive and non-primitive**, or of non compatible* non-primitive types, then code does not compile
- else if `a` and `b` of same primitive type, then evaluates to boolean `true` iff `a` and `b` have the same value, otherwise evaluates to `false`
- else evaluates to boolean `true` if `a` and `b` **reference the same object** or both do not reference any object (i.e., they "are" `null`), otherwise evaluates to `false`

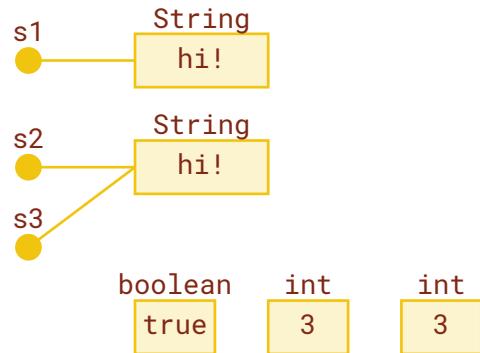
`a != b`

- if it compiles, evaluates to the negation of `a == b`

(*: "compatible" means that types are the same or one derives from the other)

Same object!

```
String s1 = "hi!";
String s2 = "hi!";
String s3 = s2;
boolean b;
b = s1 == s2; // -> false
b = s2 == s3; // -> true
b = s1.length() == s2.length(); // -> true
```



equals()

`Object`: "indicates whether some other object is "equal to" this one".

Actual meaning depend on the type, that possibly overrides
`equals()`

- `String`: "true if and only if [...] represents the same sequence of characters [...]"
- `Date`: "true if and only if [...] represents the same point in time, to the millisecond [...]"
- `PrintStream`: does not override `equals()`
 - the developers of `PrintStream` **decided** that the "equal to" notion of `Object` hold for `PrintStreams`

equals() documentation in Object

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Two things in this documentation:

- what's the general semantics of `equals()`
- how does the specific implementation in `Object` work

equals() in Object

"the most discriminating possible equivalence relation":

```
public class Object {  
    /* ... */  
    public boolean equals(Object other) {  
        if (other == null) {  
            return false;  
        }  
        return this == other;  
    }  
}
```

Note: **this** cannot be **null**!

Overriding `equals`

Semantics level requirements:

- **equivalence** conceptually sound for the specific case
- reflexivity, symmetry, transitivity, consistency, `null`

Syntax level requirement:

- comply with the signature

```
public boolean equals(Object other)
```

 - the argument is of type `Object` also for the deriving type!

The compiler:

- does check fulfillment of syntax requirement
- **cannot check** fulfillment of semantics requirements
 - big responsibility of the developer

Implementing equivalence

It depends on the case, i.e., on the type; it should correspond to equivalence of the represented concepts.

Given type X , should answer the question:

- are x_1 and x_2 the same X ?

E.g., equivalence of persons: are two persons the same?

- in the real world:
 - formed by the same atoms? (probably too strict)
 - same first and last name? (probably too loose)
- in the application:
 - depends of what aspects of real world are modeled

Equivalent persons

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthDate;  
}
```

This class models a person by her/his first name, last name, and birth date:

- it's a **developer's decision**

At most (in strictness), `equals()` can say that "two persons are the same iff they have the same first name, same last name, and same birth date".

- what if two "different" persons has the same name and birth date? (Marco Rossi!)
- what if a person changes name?

Equivalent persons with fiscal code

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthDate;  
    private String fiscalCode;  
}
```

This class models a person with [...] and fiscal code.

By **domain knowledge**, the developer can decide that "two persons are the same iff they have the same fiscal code".

Simple Person.equals()

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthDate;  
  
    public boolean equals(Object other) {  
        if (other == null) {  
            return false;  
        }  
        if (!(other instanceof Person)) {  
            return false;  
        }  
        if (!firstName.equals(((Person) other).firstName)) {  
            return false;  
        }  
        if (!lastName.equals(((Person) other).lastName)) {  
            return false;  
        }  
        if (!birthDate.equals(((Person) other).birthDate)) {  
            return false;  
        }  
        return true;  
    }  
}
```

- (Person) to downcast **other** as Person
- assumes fields are not null

Person.equals() with fiscal code

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthDate;  
    private String fiscalCode;  
  
    public boolean equals(Object other) {  
        if (other == null) {  
            return false;  
        }  
        if (!(other instanceof Person)) {  
            return false;  
        }  
        if (!fiscalCode.equals(((Person) other).fiscalCode)) {  
            return false;  
        }  
        return true;  
    }  
}
```

We (the developers) **decided** that two persons are the same if they have the same fiscal code, **regardless** of name and birth date!

IDEs and `equals()`

As for `toString()`, IDEs have an option for writing `equals()` for you (`Alt+Ins` in Netbeans). E.g., outcome with `fiscalCode`:

```
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Person other = (Person) obj;
    if (!Objects.equals(
        this.fiscalCode,
        other.fiscalCode
    )) {
        return false;
    }
    return true;
}
```

- `this == obj`: a performance optimization
- same `getClass()` different than `instanceof`!
- `Objects.equals()` takes care of checking if field is `null`

(Ignore `final` for now)

Objects

This class consists of static utility methods for operating on objects, or checking certain conditions before operation.

A class with **utility methods**, static by design.

Objects class:

Type	Field	Description
static int	checkIndex(int index, int length)	Checks if the index is within the bounds of the range from 0 (inclusive) to length (exclusive).
static boolean	deepEquals(Object a, Object b)	Returns true if the arguments are deeply equal to each other and false otherwise.
static boolean	equals(Object a, Object b)	Returns true if the arguments are equal to each other and false otherwise.
static boolean	isNull(Object obj)	Returns true if the provided reference is null otherwise returns false.

"deeply equal": equal, but also with arrays.

Many other similar classes in and out the JDK, e.g., Arrays:

- static **fill()**, **sort()**, ...

instanceof vs. getClass()

a instanceof B

- true iff object referenced by a might be legitimately referenced by a reference b to objects of type B

a.getClass() == b.getClass():

- true iff at runtime class of object referenced by a is exactly the same (==) of class of object referenced by b

a.getClass() == B.class

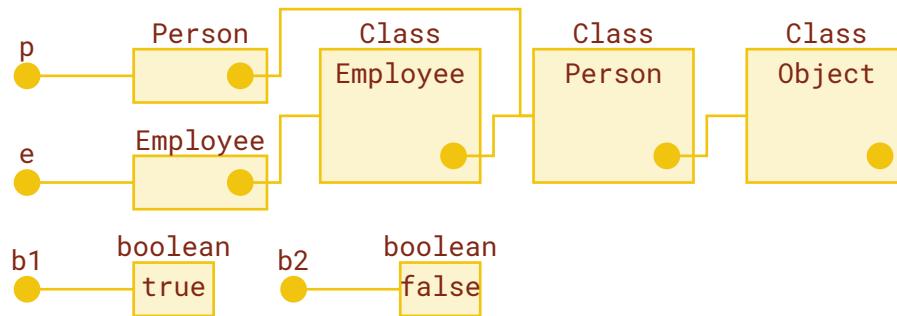
- X.class is (approx.) an implicit static reference to the object representing the class X
- true iff at runtime class of object referenced by a is exactly B

instanceof vs. getClass(): example

```
public class Person { /* ... */ }

public class Employee extends Person { /* ... */ }

Person p = new Person( /* ... */ );
Employee e = new Employee( /* ... */ );
boolean b1 = e instanceof Person; // -> true
boolean b2 = p.getClass() == e.getClass(); // -> false
```



Is an employee equal to the person being the employee? Big question: it depends on how you model the reality.

equals() on complex types

equals() should represent the **equivalence** that is inherently defined by the representation defined by the type.

```
public class BagOfInts {  
    private int[] values;  
}
```

```
public class SequenceOfInts {  
    private int[] values;  
}
```

Models $X = \mathcal{M}(\mathbb{Z}) = \{f : \mathbb{Z} \rightarrow \mathbb{N}\}$. (**bag == multiset**)

Equiv. of x_1, x_2 considers:

- length
- items

Models $X = \bigcup_{i \in \mathbb{N}} \mathbb{Z}^i$.

Equiv. of x_1, x_2 considers:

- length
- items
- order of items

(X "is" the class, $x \in X$ is the object)

($\mathcal{M}(A)$ is my notation for the set of **multisets** built with elements of the set A .)

Everything in the name!

$$X = \mathcal{M}(\mathbb{Z})$$

$$X = \bigcup_{i \in \mathbb{N}} \mathbb{Z}^i$$

```
class BagOfInts {  
    private int[] values;  
}
```

```
class SequenceOfInts {  
    private int[] values;  
}
```

Exactly the **same code, different models!**

Difference in models should be captured by the name of the type!

Multiple inheritance

```
public class Vehicle { /* ... */ }

public class MotorVehicle extends Vehicle {
    public void startEngine() { /* ... */ }
}

public class Motorboat extends MotorVehicle {
    public void startEngine() { System.out.println("bl bl bl"); }
    public void turnPropeller() { /* ... */ }
}

public class Car extends MotorVehicle {
    public void startEngine() { System.out.println("brum brum"); }
    public void turnWheels() { /* ... */ }
}
```

- Every **MotorVehicle** is a **Vehicle**
- Every **Motorboat** is a **MotorVehicle**, thus also a **Vehicle**
- Every **Car** is a **MotorVehicle**, thus also a **Vehicle**.
- In general, a **Car** is not a **Motorboat**!

Here comes the amphibious!



Ohhh, surprise!

There is a **Car** that is **also** a
Motorboat!

(It is a [Fiat 6640](#))

```
public class AmphibiousVehicle extends Motorboat, Car { // NOOOO!
    public void doAmphibiousFancyThings() { /* . . . */ }
}
```

Does not compile! No multiple inheritance!

Why not?

```
public class AmphibiousVehicle extends Motorboat, Car { // NOOOO!
    public void doAmphibiousFancyThings() { /* ... */ }
}
```

```
AmphibiousVehicle fiat6640 = new AmphibiousVehicle();
fiat6640.startEngine();
```

Should it result in `bl bl bl` or in `brum brum`?

- it's certain that there is a `startEngine()`, since both `Motorboat` and `Car` extends `MotorVehicle`: which one should be invoked?
(same for `equals()`, `toString()`, ...)
- same for fields

Java syntax does not allow the developer to specify which one!

- it was a precise decision of Java developers
- some other languages allow [multiple inheritance](#)

Why not?

"it was a precise decision of Java developers"

They might have allowed multiple inheritance from A, B only when A and B intersection of overridden methods was empty:

```
public class A { public void doA() { /*...*/ } }

public class B { public void doB() { /*...*/ } }

public class C extends A, B { public void doC() { /*...*/ } }
```

Looks legit! A c can doA(), doB(), doC(), and do `toString()`!

What if then the developer of A override `toString()`!

- **Disaster!** The same code of C does not compile anymore!

(There is a mechanism in Java for modeling the amphibious; we'll see it!)

Equivalence (~2h, 2nd home assignement)

1. Design and implement a class that represents $X = \mathcal{M}(\mathbb{R})$
 - with a proper `equals()`
 - with a proper `toString()`
 - with a proper constructor that takes $0+$ values $\in \mathbb{R}$
2. Write an application that reads from stdin two comma-separated lists of real numbers, creates $x_1, x_2 \in X$, and outputs a text message saying if x_1, x_2 are the same or not
 - user gives two separated inputs
 - comma separates values within each input

(There are other ways for implementing multisets (and sets, sequences, ...); we'll see them.)

(Do not use the collection framework. Do not use [Guava Multiset](#))

More on
parameters, visibility of identifiers, memory

What happens to parameters?

```
public class Person {  
    private String name;  
    public Person(String name) { this.name = name; }  
    public String getName(){ return name; }  
    public void setName(String name){ this.name = name; }  
}  
  
public void modify(Person p) {  
    p.setName(p.getName().toUpperCase());  
    //no return!  
}  
  
Person eric = new Person("Eric");  
modify(eric);  
System.out.println(eric.getName());
```

What's the **name** here? Are the changes visible to the caller?

Is the parameter passed **by value** or **by reference**?

Parameter passing

By value:

- the called receives a **copy of the object**

By reference:

- the called receives a **copy of the reference**

In Java:

- primitive types are passed by value
- non-primitive types are passed by reference

Passing non-primitive

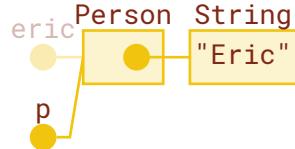
```
public void modify(Person p) { // B  
    p.setName(p.getName().toUpperCase());  
}
```

```
Person eric = new Person("Eric"); // A  
modify(eric); // C
```

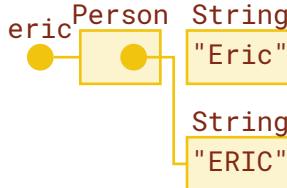
At A:



At B:



At C:



By reference:

p is a copy of eric Change is **visible!**

Passing primitive

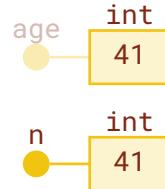
```
public void uselesslyModify(int n) { // B  
    n = 18;  
}
```

```
int age = 41;           // A  
uselesslyModify(age); // C
```

At A:



At B:



At C:



Change is **not visible!**

By value:

41 is a copy of 41

final

A modifier (keyword):

- applicable to classes, methods, fields, variables
- **static**: effects only at **compile-time**

Different effects depending on type of definition:

- class
- method
- fields and variables

final class

Cannot be extended!

- i.e., the compiler raises an error when compiling a class definition that attempts to extend a **final** class

```
public final class Greeter {  
    /* ... */  
}
```

```
public class EnhancedGreeter extends Greeter {  
    /* ... */  
}
```

EnhancedGreeter does not compile!

final method

Cannot be overriden!

- i.e., the compiler raises an error when compiling a class definition, extending a class, that attempts to override a `final` method of the extended class

```
public class Greeter {  
    public final String greet() { return "Hello!" };  
}  
  
public class EnhancedGreeter extends Greeter {  
    public String greet() { /* ... */ }  
}
```

EnhancedGreeter does not compile!

- neither with `public final String greet()`

final fields and variables

Cannot be reassigned!

- i.e., cannot be assigned more than once
 - regardless of being primitive or not
- in practice, they are **constants**

```
public class Greeter {  
    private final String msg = "Hi!";  
    public void update(String msg) {  
        this.msg = msg;  
    } }
```

```
public class Greeter {  
    public void update(final int n) {  
        n = 18;  
    }  
    public void doThings() {  
        final String s = "hi!";  
        s = "hello";  
    } }
```

Does not compile!

Does not compile:

- for **n = 18**
- for **s = "hello"**

Why/when using `final`?

Recall, **compile-time** only effects!

- the developer asks the compiler to check usage of `final` things
 - developer to her/him-self "message"
 - developer to other developer "message"
- no effects at runtime

Use cases:

- constants
 - fields (very common)
- limiting re-usability
 - classes and methods (rare)
- remarking immutability
 - variables (rather rare good practice)

Constants with `final`

Object-wise constant:

- an **immutable** field that can have different values across different instances of the same class
- just `final` (usually `private`)

Class-wise constant:

- a **global** constant, i.e., a placeholder for a hard-coded value
- `final static`

Object-wise constant

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private final Date birthDate;  
    private final String fiscalCode;  
  
    public Person(Date birthDate, String fiscalCode) {  
        this.birthDate = birthDate;  
        this.fiscalCode = fiscalCode;  
    }  
}
```

A developer's **decision** on how to model the "reality":

- for a given **Person**, **birthDate** and **fiscalCode** never change!
 - they are immutable
- for a given **Person**, **firstName** and **lastName** can change

final fields have to be initialized!

237 / 419

Class-wise constant

```
public class Doc extends Person {  
    private final static String TITLE_PREFIX = "Dr.";  
  
    public String getFullName() {  
        return TITLE_PREFIX + " " + firstName + " " + lastName;  
    }  
}
```

"Dr ." is just a costant that we may re-use in many places

- `final static`
- avoid the risk to type it differently
- ease changing the value all-at-once
- `public` if meant to be used outside (e.g., `Math.PI`)

Naming convention is different for `final static` constants:

- all uppercase
- components separated by `_` (underscore)

Remarking immutability

When parameter is a primitive type:

```
public void update(final int n) {  
    /* ... */  
}
```

Reminds me (the developer) that changing **n** would not have effect!

- i.e., dear compiler, tell me if I forget this and attempt to wrongly reassign **n**

In some cases, the IDE suggests to add the **final** modifier.

Also for non-primitive!

```
public void capitalizeName(final Person p) {  
    p.setName(  
        p.getName().substring(0, 1).toUpperCase() +  
        p.getName().substring(1).toLowerCase()  
    );  
}
```

No reason for reassigning **p**!

- if you'd reassign **p**, you were modifying another **Person** object,
not on the passed one (meant to be modified)!

Remarking immutability of variables

```
Person[] teachers = new Person[] {alberto, eric};  
for (final Person teacher : teachers) {  
    capitalizeName(teacher);  
}
```

No reason for reassigning **teacher**!

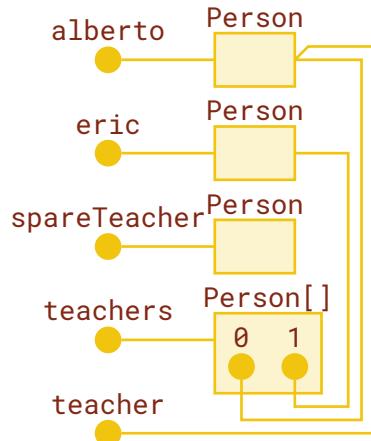
- if you'd reassign it, you were **not** operating on the array element!

```

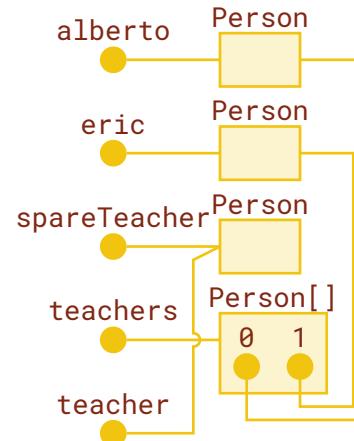
Person spareTeacher = new Person(/* ... */);
Person[] teachers = new Person[] {alberto, eric};
for (Person teacher : teachers) {
    if (teacher.isIll()) { // A
        teacher = spareTeacher;
    } // B
}

```

At A, 1st iteration:



At B, 2nd iter., assume `eric` is ill:



Scope and lifetime

Scope vs. lifetime

Scope of an identifier:

- code portion where the identifier can be legitimately used (i.e., where it **is visible**)
- **static** property

Lifetime of an object or reference:

- time interval when the object or reference **exists**
- **dynamic** property

In general:

- depend on language
- depend on entity type

Scope: reference

Declared in method signature (aka argument variable):

- visible everywhere in the method

Declared in method code (aka local variable):

- visible from declaration to the end of **block**
 - a block is a sequence of statements enclosed by curly brackets { }

Example

```
public String capitalize(final String string) {  
    String capitalized = "";  
    for (final String token : string.split(" ")) {  
        String head = token.substring(0, 1);  
        String remaining = token.substring(1);  
        capitalized = capitalized  
            + head.toUpperCase() + remaining.toLowerCase() + " ";  
    }  
    return capitalized;  
}
```

- Five identifiers: **string, capitalized, token, head, remaining**
- Five different scopes

[What is the scope for each identifier (line of start, line of end)?]

[Do you see other issues in this code? (At least 4..)]

Scope: fields, class, method

Determined by access modifier:

- `private`, `protected`, `default`, `public`

`protected`:

- visible only in a class extending (directly or indirectly) the class where `protected` is used

(Fields are also known as instance variables)

Lifetime

Reference and primitive object:

- exists **only** during the execution of the **block**

Non-primitive object:

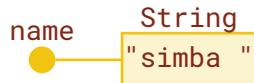
- exists at least **as long as it is referenced**
- when non referenced, might exist or not exist
 - in practice it's the same: it cannot be used!

Example

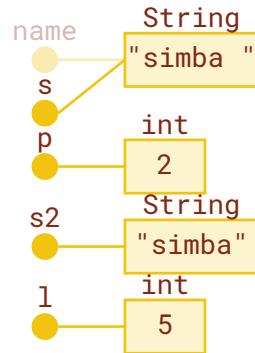
```
public int doThings(String s) {  
    int p = 2;  
    String s2 = s.trim();  
    int l = s2.length(); // [B]  
    return l / p;  
}
```

```
public void run() {  
    String name = "simba "; // [A]  
    int n = doThings(name); // [C]  
}
```

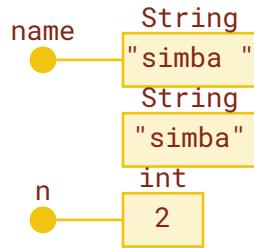
At A:



At B:



At C:



JVM memory

(some more details)

Overview

JVM memory is organized in:

- **heap**
 - long-lived data
 - global
- **stack**
 - short-lived data
 - organized in blocks

Many other differences that we do not need to know.

(Stack has faster access; one stack per thread; stack is much smaller)

Heap vs. stack: storing

Stored in heap:

- every **non-primitive object**
 - everything that is created with **new**
 - with its **fields**, primitive and non-primitive

Stored in stack:

- every **reference**
- every **primitive object not being a field**
- i.e., argument and local variables "created" within methods

Heap vs. stack: freeing

Heap:

- "cleaned" every while (we'll see soon)

Stack (organized in blocks):

- one new block created **at each method invocation**
- block removed (memory becomes free) **just after invocation**

Stack and primitive types

Primitive type objects in the stack: **the identifier identifies ("is") the object**, instead of the reference

This explains the differences:

- lifetime like references
- parameter passing by value
 - "reference by value" is like "object by reference"
- assignment creates copy of value
 - reference assignment creates copy of reference
- `==` compares content
 - content of reference is the "address" of referenced object

And:

- primitive types cannot be `null`

Primitive like references



`String* 0xAAF3` references the `String` stored at `0xAAF3` in the heap.

We'll continue to use the previous notation.

Primitive types has a precise size in memory:

Type	Size	Type	Size	Also references!
byte	1 byte	float	4 bytes	
short	2 bytes	double	8 bytes	4 bytes (heap < 32 GB)
int	4 bytes	char	2 bytes	8 bytes (heap \geq 32 GB)
long	8 bytes	boolean	*	

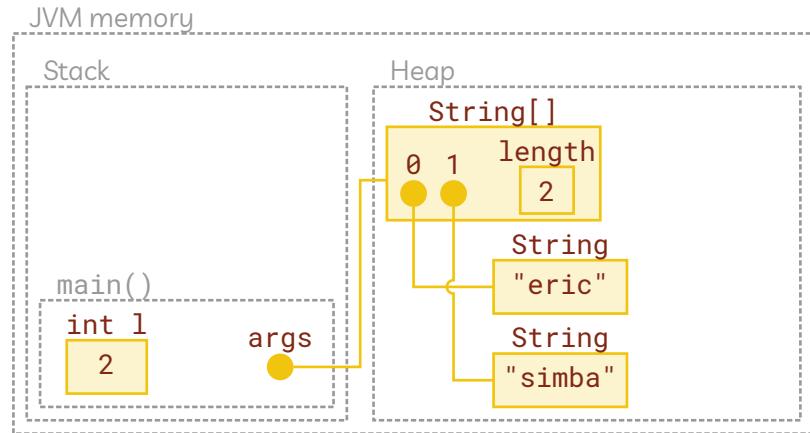
(*: JVM dependent)

→ JVM knows in advance how much stack reserve for a method invocation!

Diagram: updated syntax

```
public static void main(String[] args) {  
    int l = args.length;  
}
```

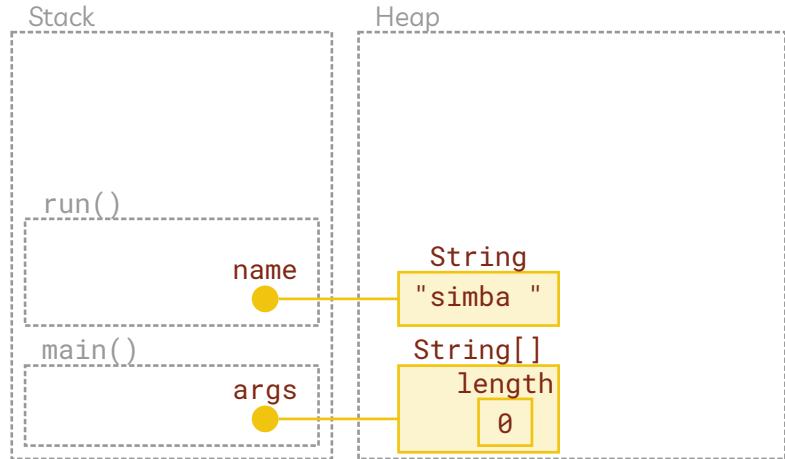
```
eric@cpu:~$ java Greeter eric simba
```



Example updated

```
public int doThings(String s) {  
    int p = 2;  
    String s2 = s.trim();  
    int l = s2.length(); // B  
    return l / p;  
}  
  
eric@cpu:~/java Greeter
```

```
public static void main(String[] args) {  
    (new Main()).run();  
}  
  
public void run() {  
    String name = "simba "; // A  
    int n = doThings(name); // C  
}
```



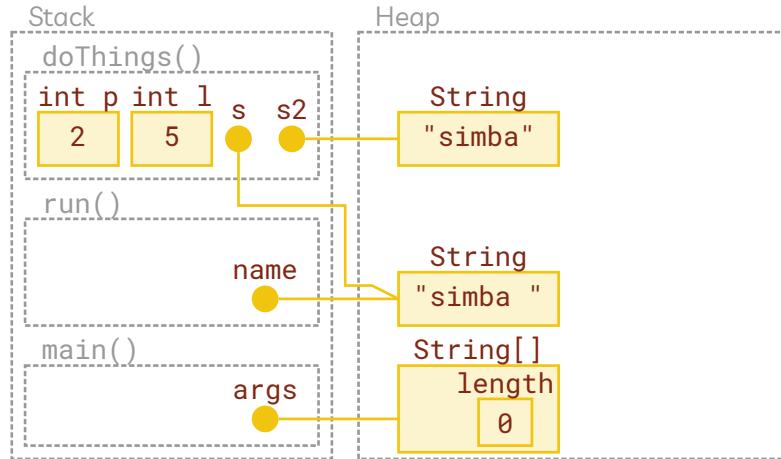
(Also `String` has its fields; we omit for compactness.)

257 / 419

Example updated

```
public int doThings(String s) {  
    int p = 2;  
    String s2 = s.trim();  
    int l = s2.length(); // B  
    return l / p;  
}  
  
eric@cpu:~/java Greeter
```

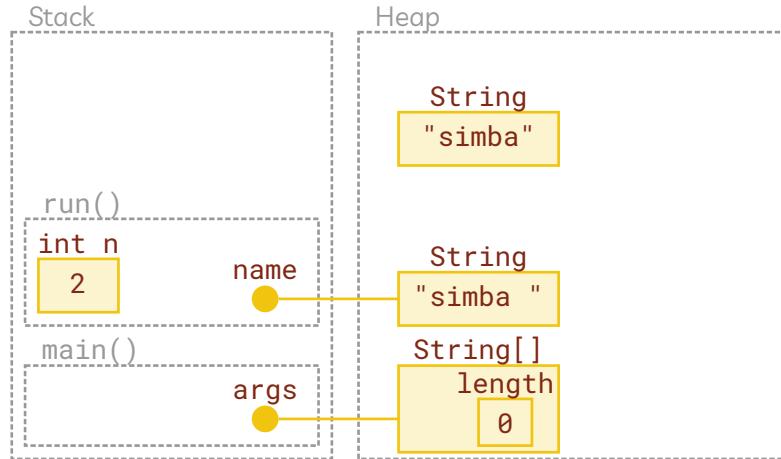
```
public static void main(String[] args) {  
    (new Main()).run();  
}  
  
public void run() {  
    String name = "simba "; // A  
    int n = doThings(name); // C  
}
```



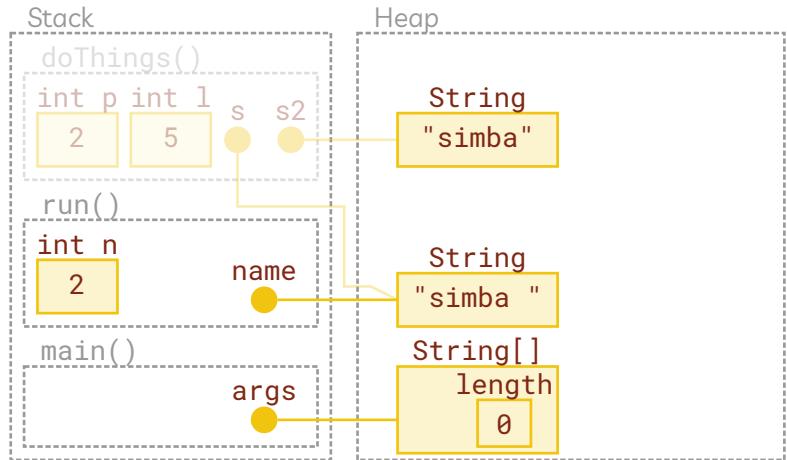
Example updated

```
public int doThings(String s) {  
    int p = 2;  
    String s2 = s.trim();  
    int l = s2.length(); // B  
    return l / p;  
}  
  
eric@cpu:~/java Greeter
```

```
public static void main(String[] args) {  
    (new Main()).run();  
}  
  
public void run() {  
    String name = "simba "; // A  
    int n = doThings(name); // C  
}
```



Freeing the memory

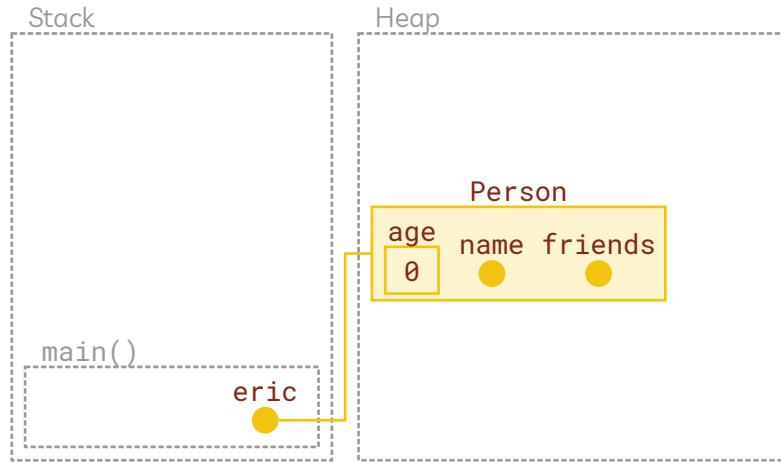


- `doThings()` block in the stack freed just after invocation
- `String "simba"` no more referenced, hence useless → **garbage**
 - who/when/how frees the corresponding heap space?

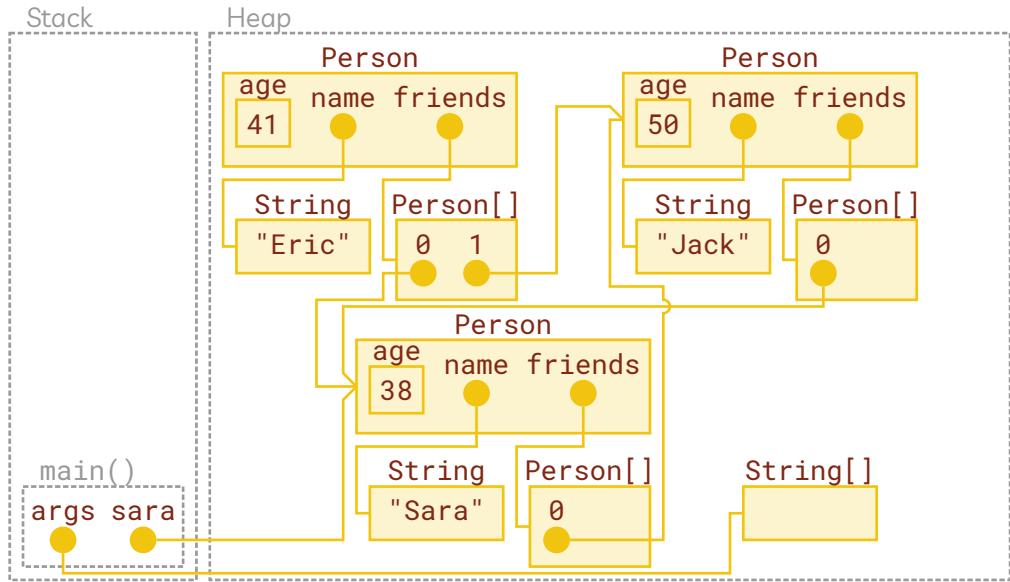
Garbage

```
public class Person {  
    private int age;  
    private String name;  
    private Person[] friends;  
}
```

```
//main()  
Person eric = new Person();
```



Complex garbage



(`length` field for arrays omitted)

[What is garbage here?]

Garbage collection

The JVM itself takes care of removing the garbage from the heap:

- decides **when**
- decides **what** garbage to remove
- trade-off between avoiding overhead and having some free space

The component of the JVM doing this cleaning is the **garbage collector** (GC)

- different GCs, different **how**

System.gc()

The developer may **suggest** the JVM to do garbage collection:

System class:

Type	Field	Description
static void	gc()	Runs the garbage collector in the Java Virtual Machine.

Runs the garbage collector in the Java Virtual Machine.
Calling the gc method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for reuse by the Java Virtual Machine. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all unused objects. There is no guarantee that this effort will recycle any particular number of unused objects, reclaim any particular amount of space, or complete at any particular time, if at all, before the method returns or ever.

Just a kind request...

Before GC

Before GC, the developer was responsible for freeing unused memory.

- e.g., `malloc()` → `free()`

Responsability → source of problems, when misbehavior

- forget to call `free()` → out of memory
- `free()` on wrong address → invalid write
- write over than reserved `malloc()` → possible chaos

No more problems with **automatic garbage collection!**

GC: cost

GC can be tricky:

- unused objects can form graph, possibly acyclic: what is actually garbage?
- garbage can be large

Doing GC takes time!

- "unpredictably" long
- "unpredictable" when

May be undesired in very specific scenarios.

But:

- you can "tune" your GC setting
- you can ask for GC when the moment is suitable
 - by calling `System.gc()` and hoping for the best

Setting the size of available memory

JVM (**java**) parameters:

- heap
 - starting size: **-Xms**
 - max size: **-Xmx**
- stack
 - size: **-Xss**

```
eric@cpu:~$ java MyBigApplication -Xmx8G
```

When exceeded:

- **java.lang.OutOfMemoryError**: Java heap space
 - GC made the effort, but failed!
- **java.lang.StackOverflowError**

Wrapper classes

There are cases when it is better (or required) to operate on basic types as non-primitive rather than primitive.

The JDK contains a set of **wrapper** classes, one for each primitive type:

- they are **immutable** (like `Strings`)
- they can be stored in the heap

Wrapper classes

- `Integer` for `int` (with different name)
- `Double` for `double`
- `Boolean` for `boolean`
- `Character` for `char` (with different name)
- ...

(There's also a `Void` class)

268 / 419

Constants and constructor

Integer class: (the same for the others)

The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is `int`.

In addition, this class provides several methods for converting an `int` to a `String` and a `String` to an `int`, as well as other constants and methods useful when dealing with an `int`.

Constants:

Modifier and type	Field	Description
static int	BYTES	The number of bytes used to represent an <code>int</code> value in two's complement binary form.
static int	MAX_VALUE	A constant holding the maximum value an <code>int</code> can have, $2^{31}-1$.
static int	MIN_VALUE	A constant holding the minimum value an <code>int</code> can have, -2^{31} .

Constructors:

Constructor	Description
<code>Integer(int value)</code>	Deprecated. It is rarely appropriate to use this constructor.
<code>Integer(String s)</code>	Deprecated. It is rarely appropriate to use this constructor.

Methods

"Like" constructors: (only the third is "really" a constructor-like method)

Modifier and type	Field	Description
static int	parseInt(String s)	Parses the string argument as a signed decimal integer.
static int	parseInt(String s, int radix)	Parses the string argument as a signed integer in the radix specified by the second argument.
static Integer	valueOf(int i)	Returns an Integer instance representing the specified int value.

```
int n = Integer.parseInt("1100110", 2); // -> 102
```

Others:

Modifier and type	Field	Description
int	intValue()	Returns the value of this Integer as an int.
long	longValue()	Returns the value of this Integer as a long after a widening primitive conversion.

Autoboxing, autounboxing

Compiler performs obvious implicit translations:

```
public void doIntThings(int n) { /* ... */ }
public void doIntegerThings(Integer n) { /* ... */ }
```

Original:

```
Integer i = 3;
doIntThings(i);
```

```
int n = 3;
doIntegerThings(n);
```

```
Integer i = 2;
i++;
```

Translated:

```
Integer i = Integer.valueOf(3);
doIntThings(i.intValue());
```

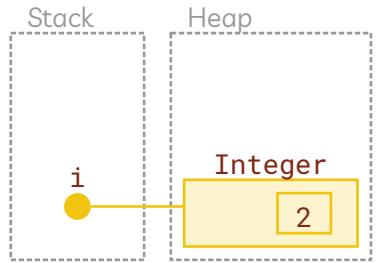
```
int n = 3;
doIntegerThings(Integer.valueOf(n));
```

```
Integer i = Integer.valueOf(2);
i = Integer.valueOf(i.intValue() + 1);
```

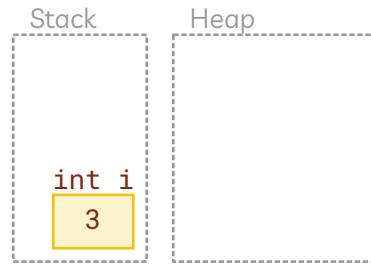
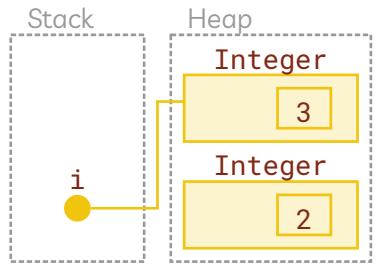
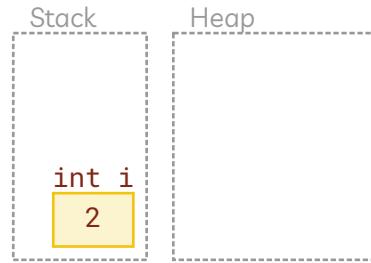
In general, use **int** when possible!

Diagram

```
Integer i = Integer.valueOf(2);  
i = Integer.valueOf(i.intValue()+1);
```



```
int i = 2;  
i++;
```



Boxing and equals

There is **no unboxing** for `==`!!!

```
Integer n = 300;
Integer m = 300;
int k = 300;
System.out.printf("n ?= m -> %b%n", n==m); // -> false!!!
System.out.printf("n ?= k -> %b%n", n==k); // -> true!!!
```

(%n in `printf()` is translated to newline!)

- In general, use `int` when possible!
- If using `Integer`, use `==` with care!
 - use `equals()`!

[What happens with `(n+1)==(m+1)?`]

(IDEs sometimes warns about misuse of `==`)

"Advanced" Input/Output (I/O)

with streams

Basic I/O

We already know how to do I/O

- of basic types (primitive and **Strings**)
- from/to stdin/stdout

```
BufferedReader reader = new BufferedReader(  
    new InputStreamReader(System.in)  
);  
/* ... */  
String line = reader.readLine();  
  
Scanner scanner = new Scanner(System.in);  
/* ... */  
String s = scanner.next();  
int n = scanner.nextInt();  
double d = scanner.nextDouble();
```

What about other types and other source/target devices?

Abstraction: I/O stream

Stream: endpoints, direction, type of data

- a stream of X between A and B

Two basic models:

- **output** stream to device where data can be written as `byte[]`
- **input** stream from a device where data can be read as `byte[]`



Interaction with a stream

Usage (key steps):

1. create the stream associating with the device
2. read or write `byte[]`

Device: file, "network", ...

Beyond basic models

Other more complex models in which there is some **processing** of the data:

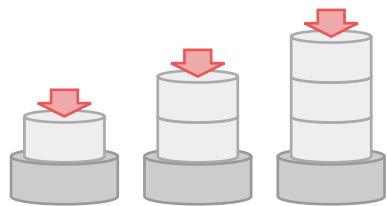
- compression/decompression of `byte[]`
- transformation of `byte[]` to/from other types
- ...

InputStream and OutputStream

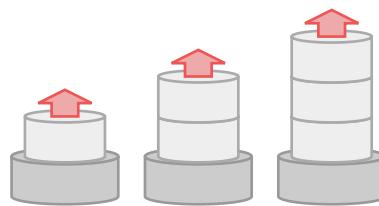
All I/O streams extend one between `InputStream` and `OutputStream`.

They can be composed:

- a stream can be build over another stream: the "result" is still a stream (of the same direction)
- data stream through streams (an instance of the `filter pattern`)
- each stream "is" the device for the stream above



Composition of `OutputStreams`

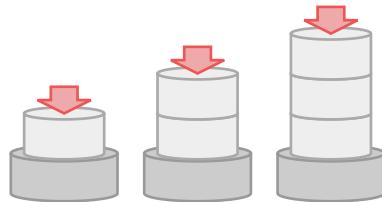


Composition of `InputStreams`

Point of view of the "user"

No need to change the code (the "user" of the stream is a developer):

- for **different devices** ← polymorphism
- for **different processings** ← polymorphism
- also for more complex (wrt `byte[]`) data types ← composition



How many I/O streams?

A lot!

We'll see a subset, for most common

- devices
- data transformation
- processing

I/O of `byte[]`

OutputStream class

This **abstract** class is the superclass of all classes representing an output stream of bytes. An output stream **accepts output bytes and sends them to some sink**.

Methods:

Mod. and Type	Method	Description
void	close()	Closes this output stream and releases any system resources associated with this stream.
void	flush()	Flushes this output stream and forces any buffered output bytes to be written out.
static OutputStream	nullOutputStream()	Returns a new OutputStream which discards all bytes.
void	write(byte[] b)	Writes b.length bytes from the specified byte array to this output stream.
void	write(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this output stream.
abstract void	write(int b)	Writes the specified byte to this output stream.

- Writes only `byte[]`
- Subclasses take devices (`OutputStream` is abstract, cannot be instantiated)
 - `OutputStream.nullOutputStream()` gives a device discarding written `byte[]`

Writing byte[]

OutputStream class:

Mod. and Type	Method	Description
void	write(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to this output stream.

Writes len bytes from the specified byte array starting at offset off to this output stream. The general contract for `write(b, off, len)` is that some of the bytes in the array b are written to the output stream in order; element `b[off]` is the first byte written and `b[off+len-1]` is the last byte written by this operation.

Other methods:

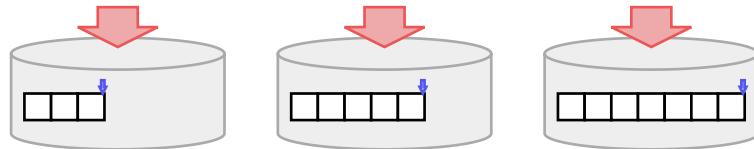
- `write(b)` is the same as `write(b, 0, b.length)`

Use of `write()`

```
byte[] data = /* ... */  
OutputStream os = /* ... */  
os.write(data, 0, 3); //A  
os.write(data, 3, 2); //B  
os.write(new byte[2]); //C
```

The device abstraction (might be a real device, or another `OutputStream`) of `os`:

- after A: contains 3 bytes; next write will start at 3 (0-indexes)
- after B: 5 bytes; **cursor** (= next write) at 5
- after C: 7 bytes; cursor at 7



Subclasses of `OutputStream`

`write(byte[] b, int off, int len)`

The general contract for `write(b, off, len)` is that some of the bytes in the array `b` are written to the output stream in order; element `b[off]` is the first byte written and `b[off+len-1]` is the last byte written by this operation.

"The general contract is [...]" : a message for:

1. the developer extending `OutputStream`
2. the curious user (another developer)
 - who can, in principle, ignore how the stream writes the data

`write(byte[] b)`

The general contract for `write(b)` is that it should have exactly the same effect as the call `write(b, 0, b.length)`.

Extending OutputStream

In principle, just `write(int b)` can be overridden! (Actually, **must** be, we'll see...) (In class description)

Applications that need to define a subclass of `OutputStream` must always provide at least a method that writes one byte of output.

But: (In `write(byte[] b, int off, int len)`)

The `write` method of `OutputStream` calls the write method of one argument on each of the bytes to be written out. Subclasses are **encouraged** to override this method and provide a **more efficient** implementation.

Internals

```
public void write(byte[] b, int off, int len) {
    for (int i = off; i<off+len; i++) {
        write(b[i]);
    }
}
```

Subclasses are **encouraged** to override this method and provide a **more efficient** implementation.

Means:

if there is a fixed cost (overhead) in writing on the device, regardless of data size, write more than one bytes at once

Associating with a device

File:

```
OutputStream os = new FileOutputStream(/* ... */);
byte[] data = /* ... */
os.write(data);
```

Network (TCP):

```
Socket socket = /* ... */
OutputStream os = socket.getOutputStream();
byte[] data = /* ... */
os.write(data);
```

Memory:

```
OutputStream os = new ByteArrayOutputStream();
byte[] data = /* ... */
os.write(data);
```

FileOutputStream

Package `java.io`

Class `FileOutputStream`

```
java.lang.Object  
    java.io.OutputStream  
        java.io.FileOutputStream
```

A file output stream is an output stream for writing data to a `File` or to a `FileDescriptor`. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one `FileOutputStream` (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

"depends upon the underlying platform": there is still a physical machine under the JVM!

Constructors:

Constructor	Description
<code>FileOutputStream(File file)</code>	Creates a file output stream to write to the file represented by the specified <code>File</code> object.
<code>FileOutputStream(File file, boolean append)</code>	Creates a file output stream to write to the file represented by the specified <code>File</code> object.
<code>FileOutputStream(String name)</code>	Creates a file output stream to write to the file with the specified name.
<code>FileOutputStream(String name, boolean append)</code>	Creates a file output stream to write to the file with the specified name.

From Socket

Socket class:

Mod. and Type	Method	Description
OutputStream	<code>getOutputStream()</code>	Returns an output stream for this socket.

The user does not need to know the actual class of the output stream returned by `getOutputStream()`.

ByteArrayOutputStream

Package java.io

Class ByteArrayOutputStream

```
java.lang.Object  
  java.io.OutputStream  
    java.io.ByteArrayOutputStream
```

This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using `toByteArray()` and `toString()`.

Mod. and Type	Method	Description
int byte[]	size() toByteArray()	Returns the current size of the buffer. Creates a newly allocated byte array.

Usage:

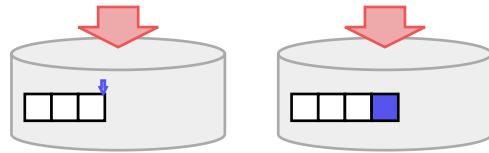
```
ByteArrayOutputStream baos = new ByteArrayOutputStream();  
byte[] data = /* ... */  
baos.write(data);  
byte[] written = baos.toByteArray();  
System.out.println(Arrays.equals(data, written)); // -> true
```

Why? As a fake device, to do to `byte[]` conversion, ...

End of Stream (EOS)

- A logical marker saying that there is no data after it
- Inserted by `close()`

```
byte[] data = /* ... */  
OutputStream os = /* ... */  
os.write(data, 0, 3); // A  
os.close(); // B
```



Usually, no other `write()` are possible after EOS has written.

EOS and `close()`

From `OutputStream.close()`:

Closes this output stream and releases any system resources associated with this stream. **The general contract** of `close` is that it closes the output stream. A closed stream cannot perform output operations and cannot be reopened.

The `close` method of `OutputStream` does nothing.

`OutputStream` is the base class, no specific device:

- "system resources" are the device
- "Closing a `ByteArrayOutputStream` has no effect. The methods in this class can be called after the stream has been closed [...]"

InputStream class

This abstract class is the superclass of all classes representing an input stream of bytes.

Methods (some):

Mod. and Type	Method	Description
abstract int	read()	Reads the next byte of data from the input stream.
int	read(byte[] b)	Reads some number of bytes from the input stream and stores them into the buffer array b.
int	read(byte[] b, int off, int len)	Reads up to len bytes of data from the input stream into an array of bytes.

- Reads only `byte[]`
- Subclasses take devices (`InputStream` is abstract, cannot be instantiated)
 - `InputStream.nullInputStream()` gives a device with no bytes to read (EOS is at position 0)

Reading byte[]

Mod. and Type	Method	Description
int	read(byte[] b, int off, int len)	Reads up to len bytes of data from the input stream into an array of bytes. Reads up to ten bytes of data from the input stream into an array of bytes. An attempt is made to read as many as ten bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer.

Attempt? We'll see...

Other methods:

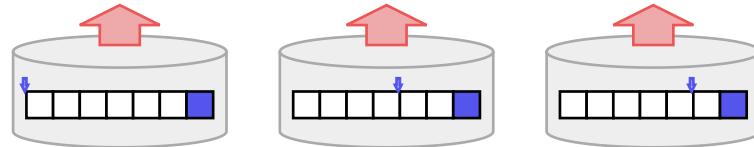
- `read(b)` is the same as `read(b, 0, b.length)`
- ...

Use of `read()`

```
byte[] buffer = new byte[100];  
InputStream is = /* ... */ //A  
is.read(buffer, 0, 4); //B  
is.read(buffer, 4, 1); //C
```

The device abstraction (might be a real device, or another `InputStream`) of `is`:

- after A: cursor at 0
- after B: cursor at 4
- after C: cursor at 5



Associating with a device

File:

```
InputStream is = new FileInputStream(/* ... */);
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);
```

Network (TCP):

```
Socket socket = /* ... */
InputStream is = socket.getInputStream();
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);
```

Memory:

```
InputStream is = new ByteArrayInputStream(/* ... */);
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);
```

FileInputStream

Package `java.io`

Class `FileInputStream`

```
java.lang.Object  
  java.io.InputStream  
    java.io.FileInputStream
```

A `FileInputStream` obtains input bytes from a file in a file system. What files are available depends on the host environment.

Constructors:

Constructor	Description
<code>FileInputStream(File file)</code>	Creates a <code>FileInputStream</code> by opening a connection to an actual file, the file named by the <code>File</code> object <code>file</code> in the file system.
<code>FileInputStream(String name)</code>	Creates a <code>FileInputStream</code> by opening a connection to an actual file, the file named by the path name <code>name</code> in the file system.

Note: no `boolean append!` (obviously)

ByteArrayInputStream

Package `java.io`

Class `ByteArrayInputStream`

```
java.lang.Object  
  java.io.InputStream  
    java.io.ByteArrayInputStream
```

A `ByteArrayInputStream` contains an internal buffer that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the read method.

"contains an internal buffer" → the device!

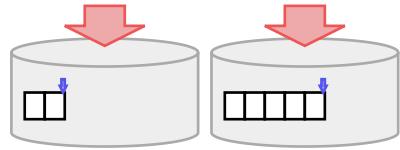
Constructors:

Constructor	Description
<code>ByteArrayInputStream(byte[] buf)</code>	Creates a <code>ByteArrayInputStream</code> so that it uses <code>buf</code> as its buffer array.

Difference in abstraction

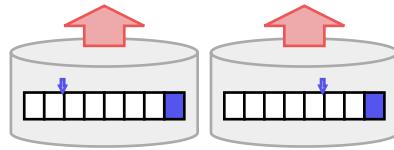
OutputStream

```
os.write(new byte[2]);  
os.write(new byte[3]);
```



InputStream

```
is.read(new byte[2]);  
is.read(new byte[3]);
```



No limit* on writeable data:

- no pre-existing EOS

Readable data is limited:

- an EOS exists!

*: in principle

Input from files and `byte[]`

Input from file with `FileInputStream`:

- created with `FileInputStream(File file)` (we'll see a `File` is)
- the file exists, hence it has a size, hence a EOS

Input from `byte[]` with `ByteArrayInputStream`:

- created with `ByteArrayInputStream(byte[] data)`
- the data exists, hence it has a size, hence a EOS

Output to files and `byte[]`

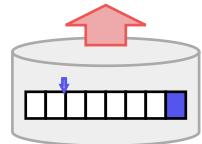
Output to file with `FileOutputStream`:

- created with, e.g., `FileOutputStream(File file, boolean append)`
- create the file (if not existing and possible), or write at the end of file → no limits to the writeable data!
 - possible limits related to underlying OS (obviously)

Output to `byte[]` with `ByteArrayOutputStream`:

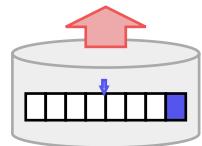
- "The buffer automatically grows as data is written to it."

Attempt of reading (with EOS)



Before: device still contains 5 bytes, the EOS

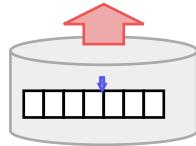
```
is.read(buf, 0, 2); // OK!
```



Before: device still contains 3 bytes, then EOS

```
is.read(buf, 0, 5); // ?
```

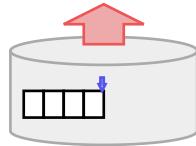
Attempt of reading (without EOS)



Before: device contains 3 **ready** bytes

```
is.read(buf, 0, 5); // ?
```

Other data might arrive (be ready) in the future...



Before: device contains no **ready** bytes

```
is.read(buf, 0, 5); // ?
```

Other data might arrive (be ready) in the future...

read()

Mod. and Type	Method	Description
int	read(byte[] b, int off, int len)	Reads up to len bytes of data from the input stream into an array of bytes.

Reads up to `len` bytes of data from the input stream into an array of bytes. An attempt is made to read as many as `len` bytes, but a **smaller number may be read**. The number of bytes actually read is returned as an integer.

This method **blocks until input data is available, end of file is detected**, or an exception is thrown.

If `len` is zero, then no bytes are read and `0` is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value `-1` is returned; otherwise, at least one byte is read and stored into `b`.

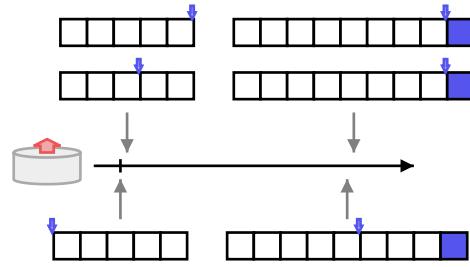
Attempt to read `len` bytes:

- if next byte in device is EOS, return immediately `-1`
- if 0 bytes ready: block and wait for data
- if $\geq \text{len}$ ready, read `len` bytes and return immediately `len`
- otherwise, read $n < \text{len}$ bytes and return immediately `n`

Example

Suppose the device:

- contains 5 bytes from time $t = 0$ to $t = 10$ (seconds)
- receives other 3 bytes, with a trailing EOS, at $t = 10$



```
is.read(buf, 0, 3);
// t=0 -> t~0, ret 3
is.read(buf, 0, 3);
// t~0 -> t~0, ret 2
is.read(buf, 0, 3); // BLOCK!
// t~0 -> t~10, ret 3
is.read(buf, 0, 3);
// t~10 -> t~10, ret -1
```

Blocking `read()`

"The device receives": the `InputStream` receives for the underlying level, eventually from a physical devices

Keyboard → `InputStream System.in`

- the user types (and hit enter) → `InputStream` receives bytes

Network connection → `InputStream getInputStream()`

- data from the network arrives to this socket → receives bytes

Internally

`read(byte[] b, int off, int len)` in `InputStream`:

The `read(b, off, len)` method for class `InputStream` simply calls the method `read()` repeatedly.

Mod. and Type	Method	Description
<code>abstract int</code>	<code>read()</code>	Reads the next byte of data from the input stream. Reads the next byte of data from the input stream. The value byte is returned as an <code>int</code> in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

`read()` read one `byte`, but returns an `int`: why?

- because the semantics of the return value includes a special value (-1) representing EOS
- domain of `byte` does not include -1!

-1 in `read()` and `write()`

Other alternatives for representing EOS:

- returning a `Byte`, `null` for EOS
 - object creation overhead; uses heap, slower
- throwing an exception (we ignore exceptions now; we'll see)
 - EOS is not an exception, indeed; it's the norm

Java inventors **chose** to use an `int` as return value

- for consistency `read(byte[], int, int)` also returns an `int`
- for consistency `write(int)` in `OutputStream` takes an `int`!

Mod. and Type	Method	Description
<code>abstract void</code>	<code>write(int b)</code>	Writes the specified byte to this output stream. Writes the specified byte to this output stream. The general contract for <code>write</code> is that one byte is written to the output stream. The byte to be written is the eight low-order bits of the argument <code>b</code> . The 24 high-order bits of <code>b</code> are ignored.

File

Surprisingly, the **File** class does not represent a file!

Package [java.io](#)

Class **File**

[java.lang.Object](#)
[java.io.File](#)

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent pathname strings to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames.

- "An abstract representation of file and directory **pathnames**".
- "system-independent view of hierarchical **pathnames**"

(a rather long description follows)

File system-independent view

The screenshot shows a JavaDoc page for the `java.io.File` class. At the top, there's a navigation bar with links for OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there are two sets of links: SUMMARY: NESTED | FIELD | CONSTR | METHOD and DETAIL: FIELD | CONSTR | METHOD. A search bar labeled "SEARCH: Search" is also present. The main content area starts with "Module java.base" and "Package java.io". The title "Class File" is bolded. Below it, the inheritance chain is shown: `java.lang.Object` and `java.io.File`. The "All Implemented Interfaces:" section lists `Serializable` and `Comparable<File>`. A code snippet follows, showing the class definition:

```
public class File
extends Object
implements Serializable, Comparable<File>
```

An abstract representation of file and directory pathnames.

User interfaces and operating systems use system-dependent *pathname strings* to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames. An *abstract pathname* has two components:

312 / 419

Not a file!

No methods for reading and writing!

Methods for:

- removing, renaming, listing (**File** represents dirs too), ...

Mod. and Type	Method	Description
boolean	canExecute()	Tests whether the application can execute the file denoted by this abstract pathname.
boolean	canRead()	Tests whether the application can read the file denoted by this abstract pathname.
boolean	canWrite()	Tests whether the application can modify the file denoted by this abstract pathname.
boolean	delete()	Deletes the file or directory denoted by this abstract pathname.
boolean	exists()	Tests whether the file or directory denoted by this abstract pathname exists.
boolean	isDirectory()	Tests whether the file denoted by this abstract pathname is a directory.
String[]	list()	Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
File[]	listFiles()	Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
boolean	mkdir()	Creates the directory named by this abstract pathname.
boolean	renameTo(File dest)	Renames the file denoted by this abstract pathname.
boolean	setExecutable(boolean executable)	A convenience method to set the owner's execute permission for this abstract pathname.

Open with FileOutputStream

`FileOutputStream(File file)` and `FileOutputStream(String name)`:

- if `file/name` does not exist, creates new one (if OS says it's possible)
- puts the cursor at 0

→ **existing content is cancelled!**

- if you want to append, use the constructors with `boolean append!`
 - puts cursor at the position equal to the length of file

Example: file copy

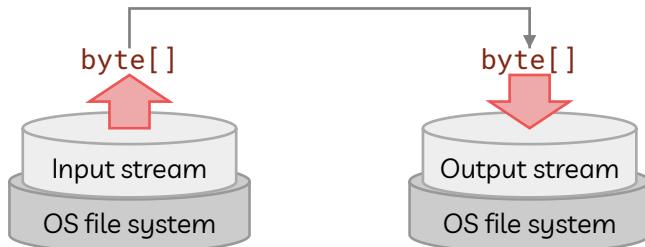
We want to develop an application that:

- receives two file names f_1, f_2 as command line arguments
- copies the content of file named f_1 to f_2
 - assume file f_1 exists and is readable
 - overwrite file f_2 if it exists

```
eric@cpu:~$ java FileCopier slides.zip copy-of-slides.zip
```

Sketch

1. build `FileInputStream` from `String f_1` ; build `FileOutputStream` from `String f_2`
2. iterate until EOS
 1. read `byte[]` from `FileInputStream`
 2. write `byte[]` to `FileOutputStream`



Stopping criterion

"Iterate until EOS"

Suppose to use a buffer of n bytes (i.e., to read n bytes for iteration):

- every iteration but second-last and last reads n bytes
- second-last reads $\leq n$
- last return $-1 \rightarrow \text{EOS}$

Possible code

```
public class FileCopier {  
    public static void main(String[] args) throws FileNotFoundException {  
        InputStream is = new FileInputStream(args[0]);  
        OutputStream os = new FileOutputStream(args[1]);  
        byte[] buffer = new byte[1024];  
        while (true) {  
            int nOfBytes = is.read(buffer);  
            if (nOfBytes == -1) {  
                break;  
            }  
            os.write(buffer, 0, nOfBytes);  
        }  
        is.close();  
        os.close();  
    }  
}
```

(Ignore `throws FileNotFoundException, IOException` for now.)

Make it more general

```
public class Util {  
    public static void copyAndClose(InputStream is, OutputStream os)  
    {  
        byte[] buffer = new byte[1024];  
        while (true) {  
            int nOfBytes = is.read(buffer);  
            if (nOfBytes == -1) {  
                break;  
            }  
            os.write(buffer, 0, nOfBytes);  
        }  
        is.close();  
        os.close();  
    }  
}
```

copyAndClose() "ignores" the specific kind of streams (devices):

- while developing/compiling: inheritance
- while executing: polymorphism

I/O of primitive types

Output with DataOutputStream

Package `java.io`

Class `DataOutputStream`

```
java.lang.Object  
    java.io.OutputStream  
        java.io.FilterOutputStream  
            java.io.DataOutputStream
```

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

Constructor	Description	
<code>DataOutputStream(OutputStream out)</code>	Creates a new data output stream to write data to the specified underlying output stream.	
Mod. and Type	Method	Description
void	<code>writeDouble(double v)</code>	Converts the double argument to a long using the <code>doubleToLongBits</code> method in class <code>Double</code> , and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
void	<code>writeFloat(float v)</code>	Converts the float argument to an int using the <code>floatToIntBits</code> method in class <code>Float</code> , and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
void	<code>writeInt(int v)</code>	Writes an int to the underlying output stream as four bytes, high byte first.
void	<code>writeLong(long v)</code>	Writes a long to the underlying output stream as eight bytes, high byte first.

"converts [...] high byte first" → **portable way**

Input with DataInputStream

Package java.io

Class DataInputStream

```
java.lang.Object
  java.io.InputStream
    java.io.FilterInputStream
      java.io.DataInputStream
```

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.

Constructor	Description	
<code>DataInputStream(InputStream in)</code>	Creates a DataInputStream that uses the specified underlying InputStream.	
Mod. and Type	Method	Description
double	<code>readDouble()</code>	See the general contract of the <code>readDouble</code> method of <code>DataInput</code> .
float	<code>readFloat()</code>	See the general contract of the <code>readFloat</code> method of <code>DataInput</code> .
int	<code>readInt()</code>	See the general contract of the <code>readInt</code> method of <code>DataInput</code> .
long	<code>readLong()</code>	See the general contract of the <code>readLong</code> method of <code>DataInput</code> .

"machine-independent" == "portable"

Filter pattern

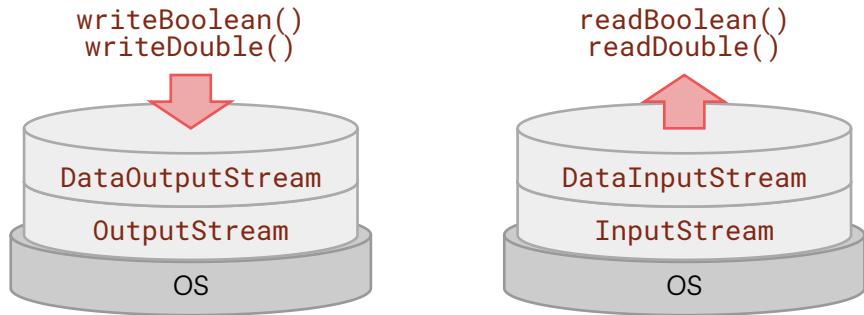
`DataOutputStream` extends `FilterOutputStream`

This class is the superclass of all classes that filter output streams. These streams sit on top of an already existing output stream (the *underlying output stream*) which it uses as its basic sink of data, but possibly transforming the data along the way or providing additional functionality.

`DataInputStream` extends `FilterInputStream`

A `FilterInputStream` contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.

Abstraction



Portability

Java developers built `DataInputStream` and `DataOutputStream` together:

- they read and write primitive types to `byte[]` accordingly
- every machine reads/write `byte[]` in the same way

⇒ portability!

E.g.:

- file written on a OS is readable from another OS
- data sent via socket is readable on the other endpoint

But...

Portability vs. protocol

Portability is granted for the single data item!

At the application level, parties (reader and writer) have to act accordingly!

- in practice, they have to share a protocol

"Real-life" example

Alice and Bob meet on chat:

- A: tell me your age, weight, and annual net income in k€
- B: 65, 95, 40
- A: 

Creative Bob:

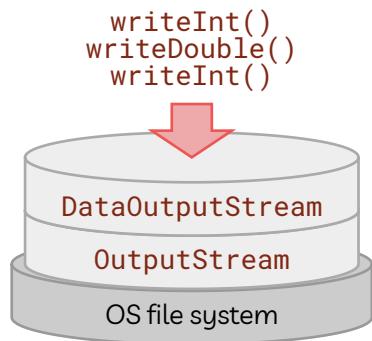
- B (unfair): 40, 65, 95
- A: 

What happened?

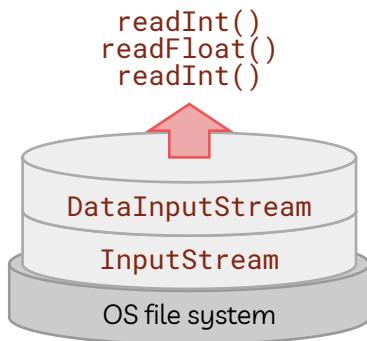
- each single number was correctly communicated!
- application protocol (order) was not respected (in the 2nd case)
 - with this data, Alice has no way for telling if Bob is complying the protocol!
 - maybe with other data, e.g., 20, 75, 40

Java streams example

Application saves data on file:



Application loads same file:



Writes 16 bytes:

- 4 for 1st `int`
- 8 for `double`
- 4 for 2nd `int`

Reads 12 bytes:

- 4 for 1st `int`
- **4** for `float`
- **4** for 2nd `int`

Disaster!

Writes 16 bytes:

- 4 for 1st `int`
- 8 for `double`
- 4 for 2nd `int`

Reads 12 bytes:

- 4 for 1st `int`
- **4** for `float`
- **4** for 2nd `int`

Read data "is different" than written data:

- and no way to realize it at runtime

File array (~3h)

1. Design and implement a class `FileArray` that maintains a **binary file-backed** array of `int` values, with constructors/methods to:

- build a new array with n random values $\sim U(\{0, \dots, 2^{10}\})$
- load an existing array
- print the array
- increment all elements of the arrays

2. Write an application that "uses" `FileArray`

- receives $m \geq 1$ parameters at command read
- 1st is a file pathname
 - if not existing, create with $n \sim U(\{1, \dots, 2^5\})$, otherwise, load
- from 2nd on, a list of one-char commands:
 - `i` for increment
 - `p` for print

FileArray

```
public class FileArray {  
    // loads an existing file  
    public FileArray(String filePathName) { /* ... */ }  
    // creates new file with n random elements  
    public FileArray(String filePathName, int n) { /* ... */ }  
    // pretty print with at most 5 aligned elements per row  
    public void print();  
    //increment all elements  
    public void incrementAll();  
}
```

Hints:

- for (pseudo)random generation, see [Random.nextInt\(\)](#)
- "ignore" exceptions
- use two "internal" methods:
 - read: file → `int[]`
 - write: `int[]` → file
 - e.g.: `print=read,print; incrementAll=read,inc,write`

Application using `FileArray`

Assuming `test.bin` exists with 8 values:

```
eric@cpu:~$ java FileArrayTester test.bin p i i p
[00-04]  8 73 12  4 99
[05-09] 13 23 75 33  9
[10-12] 18   6 78
[00-04] 10 75 14   6 101
[05-09] 15 25 77 35  11
[10-12] 20   8 80
```

And the file content is left accordingly.

Note:

- **alignment** and (zero-padded aligned) **indexes**
- column width is the shortest possible (2 at 1st `p`, 3 at 2nd `p`)

Hint: use `printf` or `String.format()`, e.g., `%3d`

Compression with GZIPOutputStream

Package `java.io`

Class `GZIPOutputStream`

```
java.lang.Object
  java.io.OutputStream
    java.io.FilterOutputStream
      java.util.zip.DeflaterOutputStream
        java.util.zip.GZIPOutputStream
```

This class implements a stream filter for writing compressed data in the GZIP file format.

Constructor	Description	
<code>GZIPOutputStream(OutputStream out)</code>	Creates a new output stream with a default buffer size.	
Mod. and Type	Method	Description
<code>void</code>	<code>finish()</code>	Finishes writing compressed data to the output stream without closing the underlying stream.
<code>void</code>	<code>write(byte[] buf, int off, int len)</code>	Writes array of bytes to the compressed output stream.

Decompression with GZIPInputStream

Package java.io

Class GZIPInputStream

```
java.lang.Object
  java.io.InputStream
    java.io.FilterInputStream
      java.util.zip.InflaterInputStream
        java.util.zip.GZIPInputStream
```

This class implements a stream filter for reading compressed data in the GZIP file format.

Constructor	Description	
GZIPInputStream(InputStream in)	Creates a new input stream with a default buffer size.	
Mod. and Type	Method	Description
void	close()	Closes this input stream and releases any system resources associated with the stream.
int	read(byte[] buf, int off, int len)	Reads uncompressed data into an array of bytes.

GZIP vs. zip

There are also `ZipOutputStream` and `ZipInputStream`:

- `GZIP` streams "just" compress/decompress binary data
- `Zip` streams operate with files
 - they can contain files and directories

GZIP file array (~1h, 3rd home assign.)

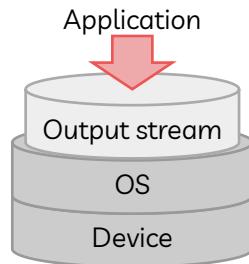
1. Design and implement a class `GZIPFileArray` that **extends** `FileArray` and maintains a **compressed** binary file-backed array of `int` values
 - with same functionalities of `FileArray` (see [specifications](#))
2. Write an application that "uses" `FileArray` or `GZIPFileArray`
 - receives $m \geq 1$ parameters at command read
 - 1st is a file pathname
 - if not existing, create with $n \sim U(\{1, \dots, 2^5\})$, otherwise, load
 - if filename ends with `.zip`, use `GZIPFileArray`, otherwise use `FileArray`
 - from 2nd on, a list of one-char commands (see [specifications](#))

Buffered I/O

OS and I/O

Most devices are actually read/written by the OS: (not, e.g., `ByteArrayOutputStream`)

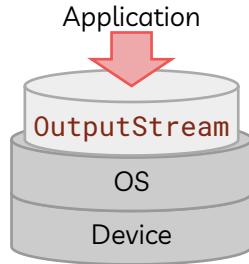
- OS performs reads and writes of data chunks of a size s_o that depends on the OS and the devices
 - e.g., 8 KB for files on disk
- invoking OS **is costly**



Applications may need to read/write data chunks with size $s_a \neq s_o$

- if $s_a < s_o$, OS is invoked more often than needed
→ inefficiency!

Writes to device



Application to `OutputStream`:

- `write()` 100 bytes, `write()` 1000 bytes, `write()` 100 bytes

`OutputStream` (JVM) to OS:

- `write()` 100 bytes, `read()` 1000 bytes, `write()` 100 bytes

OS to device:

- hopefully something optimal

Solution: buffered stream

A filter stream with a **buffer**.

BufferedOutputStream:

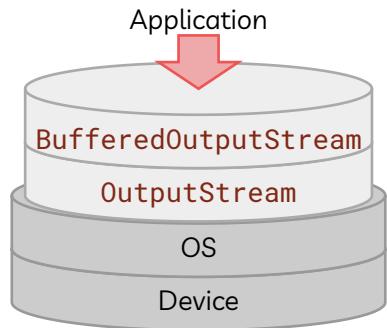
1. takes write requests as usual
2. puts data in a buffer; if full makes write requests to the underlying stream
 - or if explicitly instructed to

BufferedInputStream:

1. takes read requests as usual
2. if in buffer, takes from buffer, otherwise fill the buffer with a read from underlying stream

Writes to device with buffer

Assume buffer of 1000 bytes.



Application to `BufferedOutputStream`:

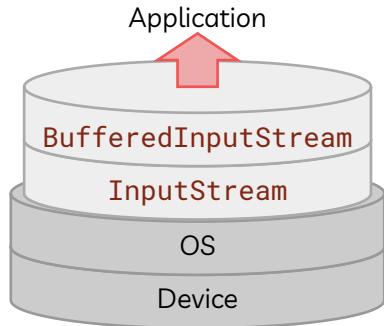
- `write()` 100 bytes, `write()` 1000 bytes, `write()` 100 bytes, ...

`BufferedOutputStream` to `OutputStream` and `OutputStream` to OS:

- `write()` 1000 bytes, ...

Reads from device with buffer

Assume buffer of 1000 bytes.



Application to **BufferedInputStream**:

- `read()` 100 bytes, `read()` 1000 bytes, `read()` 100 bytes, ...

BufferedInputStream to **InputStream** and **InputStream** to OS:

- `read()` 1000 bytes, `read()` 1000 bytes, ...

BufferedInputStream

Package `java.io`

Class `BufferedInputStream`

```
java.lang.Object  
    java.io.InputStream  
        java.io.FilterInputStream  
            java.io.BufferedInputStream
```

A `BufferedInputStream` adds functionality to another input stream—namely, the ability to buffer the input and to support the `mark` and `reset` methods. When the `BufferedInputStream` is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.

Constructor	Description
<code>BufferedInputStream (InputStream in)</code>	Creates a <code>BufferedInputStream</code> and saves its argument, the input stream <code>in</code> , for later use.
<code>BufferedInputStream (InputStream in, int size)</code>	Creates a <code>BufferedInputStream</code> with the specified buffer size, and saves its argument, the input stream <code>in</code> , for later use.

Use with `read` methods from `InputStream`.

BufferedOutputStream

Package `java.io`

Class `BufferedOutputStream`

```
java.lang.Object  
    java.io.OutputStream  
        java.io.FilterOutputStream  
            java.io.BufferedOutputStream
```

The class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.

Constructor	Description
<code>BufferedOutputStream (OutputStream out)</code>	Creates a new buffered output stream to write data to the specified underlying output stream.
<code>BufferedOutputStream (OutputStream out, int size)</code>	Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

Use with `write` methods from `OutputStream`.

Redefines `flush()`.

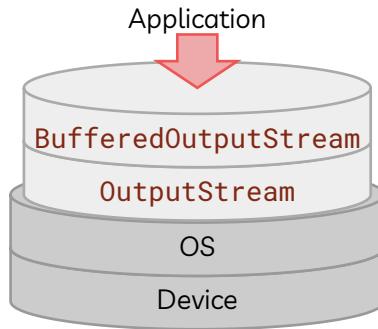
flush()

BufferedOutputStream:

1. takes write requests as usual
2. puts data in a buffer; if full makes write requests to the underlying stream
 - or if **explicitly instructed** to → **flush()**

```
public void flush()  
Flushes this buffered output stream. This forces any  
buffered output bytes to be written out to the  
underlying output stream.
```

- Invoking **flush()** propagates down to the JVM-OS boundary, not beyond!
- It **does not guarantee** that the data is actually written to the **physical device!**



When to use buffered streams?

Always!

- Code that uses a I/O stream, can use a I/O buffered stream too!
- Good practice:

```
OutputStream os = new BufferedOutputStream(  
    new FileOutputStream(file)  
)
```

Should you **flush()**?

- if your application is robust, you don't need to: content is flushed at **close()**
- **flush()** needed only for output stream:
 - **BufferedInputStream** reads from below more at least the requested data, possibly more

I/O of text data

Binary vs. text data

- Binary data is `byte[]`
- Text data is `char[]`

A `char` is encoded with one or more `bytes`: the precise way each `char` is encoded is specified in a **charset**.

- Java deals with charsets with the `Charset` class
 - in most cases, you don't need to mess with `Charset`

I/O of text data

Many application do I/O of text data, rather than binary data:

- HTTP protocol
- CSV files
- ...

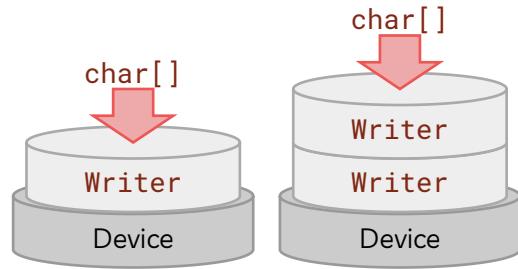
The JDK provides classes for text I/O that take care of `byte[]` ↔
`char[]`.

"Same" abstraction of `byte[]` streams, but with `char[]`:

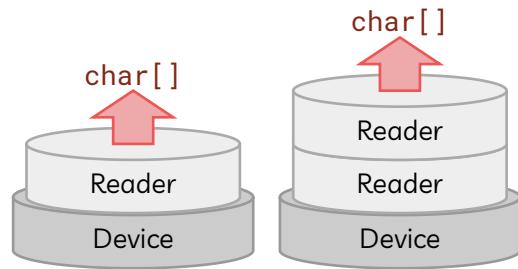
- `Writer`, `Reader` instead of `OutputStream`, `InputStream`
- can be composed with the filter pattern
- can be associated with device

Abstraction

Output of text:



Input of text:



Output with Writer

Package `java.io`

Class Writer

`java.lang.Object`
`java.io.Writer`

Abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Mod. and Type	Method	Description
<code>Writer</code>	<code>append(char c)</code>	Appends the specified character to this writer.
<code>Writer</code>	<code>append(CharSequence csq)</code>	Appends the specified character sequence to this writer.
<code>Writer</code>	<code>append(CharSequence csq, int start, int end)</code>	Appends a subsequence of the specified character sequence to this writer.
<code>abstract void</code>	<code>close()</code>	Closes the stream, flushing it first.
<code>abstract void</code>	<code>flush()</code>	Flushes the stream.
<code>static Writer</code>	<code>nullWriter()</code>	Returns a new Writer which discards all characters.
<code>void</code>	<code>write(char[] cbuf)</code>	Writes an array of characters.
<code>abstract void</code>	<code>write(char[] cbuf, int off, int len)</code>	Writes a portion of an array of characters.
<code>void</code>	<code>write(int c)</code>	Writes a single character.
<code>void</code>	<code>write(String str)</code>	Writes a string.
<code>void</code>	<code>write(String str, int off, int len)</code>	Writes a portion of a string.

char[] are Strings

`write(String str)` might use `String.toCharArray()`:

```
public void write(String str) throws IOException {
    write(str.toCharArray());
}
```

Strings are CharSequences:

```
public Writer append(CharSequence csq) throws IOException {
    write(csq.toString());
}
```

append()

Mod. and Type	Method	Description
Writer	append(char c)	Appends the specified character to this writer.
Writer	append(CharSequence csq)	Appends the specified character sequence to this writer.
Writer	append(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this writer.

append() methods return a **Writer**, actually this **Writer**

- can be used with chain invocation

```
Writer writer = /* ... */;  
writer  
    .append("Lorem ipsum dolor sit amet, ")  
    .append("consectetur adipiscing elit, ")  
    .append("sed do eiusmod tempor incididunt ")  
    .append("ut labore et dolore magna aliqua.");
```

Writing text to files with FileWriter

Package `java.io`

Class `FileWriter`

`java.lang.Object`
`java.io.Writer`
`java.io.OutputStreamWriter`
`java.io.FileWriter`

Writes text to character files using a default buffer size. Encoding from characters to bytes uses either a specified charset or the platform's default charset.

Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one `FileWriter` (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.

The `FileWriter` is meant for writing streams of characters. For writing streams of raw bytes, consider using a `FileOutputStream`.

Constructor	Description
<code>FileWriter(String fileName, Charset charset, boolean append)</code>	Constructs a <code>FileWriter</code> given a file name, charset and a <code>boolean</code> indicating whether to append the data written.

```
Writer writer = new FileWriter("file.txt");
writer.write("Hello world!");
writer.close();
```

Writing text to OutputStreamWriter

Package `java.io`

Class `OutputStreamWriter`

`java.lang.Object`
 `java.io.Writer`
 `.java.io.OutputStreamWriter`

An `OutputStreamWriter` is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.

Each invocation of a `write()` method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. Note that the characters passed to the `write()` methods are not buffered.

Constructor

Constructor	Description
<code>OutputStreamWriter(OutputStream out)</code>	Creates an <code>OutputStreamWriter</code> that uses the default character encoding.
<code>OutputStreamWriter(OutputStream out, String charsetName)</code>	Creates an <code>OutputStreamWriter</code> that uses the named charset.
<code>OutputStreamWriter(OutputStream out, Charset cs)</code>	Creates an <code>OutputStreamWriter</code> that uses the given charset.
<code>OutputStreamWriter(OutputStream out, CharsetEncoder enc)</code>	Creates an <code>OutputStreamWriter</code> that uses the given charset encoder.

```
OutputStream os = /* ... */;  
Writer writer = new OutputStreamWriter(os);
```

There is some buffering towards underlying `OutputStream`.

Writing "in memory"

CharArrayWriter

This class implements a character buffer that can be used as an `Writer`. The buffer automatically grows when data is written to the stream. The data can be retrieved using `toCharArray()` and `toString()`.

Note: Invoking `close()` on this class has no effect, and methods of this class can be called after the stream has closed without generating an `IOException`.

StringWriter:

A character stream that collects its output in a string buffer, which can then be used to construct a string. Closing a `StringWriter` has no effect. The methods in this class can be called after the stream has been closed without generating an `IOException`.

Writing with buffer

Package `java.io`

Class `BufferedWriter`

```
java.lang.Object  
  java.io.Writer  
    java.io.BufferedWriter
```

The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.

A `newLine()` method is provided, which uses the platform's own notion of line separator as defined by the system property `line.separator`. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.

In general, a `Writer` sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a `BufferedWriter` around any `Writer` whose `write()` operations may be costly, such as `FileWriters` and `OutputStreamWriters`.

Constructor	Description
<code>BufferedWriter(Writer out)</code>	Creates a buffered character-output stream that uses a default-sized output buffer.
<code>BufferedWriter(Writer out, int sz)</code>	Creates a new buffered character-output stream that uses an output buffer of the given size.

With `flush()`.

Input with Reader

Package java.io
Class Reader
java.lang.Object
java.io.Reader

Abstract class for reading character streams. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

Mod. and Type	Method	Description
abstract void	<code>close()</code>	Closes the stream and releases any system resources associated with it.
static Reader	<code>nullReader()</code>	Returns a new Reader that reads no characters.
int	<code>read()</code>	Reads a single character.
int	<code>read(char[] cbuf)</code>	Reads characters into an array.
abstract int	<code>read(char[] cbuf, int off, int len)</code>	Reads characters into a portion of an array.

`read()` works as for `InputStream`, but:

- no methods for reading directly to `String`!

Reading from...

- File with `FileReader`
 - Constructor: `FileReader(File file, Charset charset)`
- Any `InputStream` with `InputStreamReader`
 - Constructor: `InputStreamReader(InputStream in, Charset cs)`
- `char[]` and `String`:
 - `CharArrayReader(char[] buf)`
 - `StringReader(String s)`

Reading with buffer

BufferedReader:

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, **and lines**.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

```
public String readLine() throws IOException
```

Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), a carriage return followed immediately by a line feed, or by reaching the end-of-file (EOF).

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached without reading any characters

- Returns null if EOS is reached (null acts as -1).

Common usage

```
BufferedReader br = new BufferedReader(  
    new InputStreamReader(new FileInputStream(fileName)))  
);  
while (true) {  
    String line = br.readLine();  
    if (line == null) {  
        break;  
    }  
    /* do things with the line */  
}  
br.close();
```

(We are ignoring the management of the exceptions.)

(We'll see a more elegant way later.)

Printing anything

362 / 419

PrintStream

An output stream (extends `OutputStream`) that:

- add methods for printing anything
 - primitive types, `Strings`, objects
 - "printing" means outputting bytes of **string representation**
- **never throws exceptions**
- (buffered, optionally with auto flushing)

`System.out` is a `PrintStream`

No corresponding class for input:

- `System.in` is a "plain" `InputStream` with no added methods and with exceptions
- why? for output, the application has control on what to print, for input, no

PrintStream

Package java.io

Class PrintStream

```
java.lang.Object  
    java.io.OutputStream  
        java.io.FilterOutputStream  
            java.io.PrintStream
```

A `PrintStream` adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a `PrintStream` never throws an `IOException`; instead, exceptional situations merely set an internal flag that can be tested via the `checkError` method. Optionally, a `PrintStream` can be created so as to flush automatically; this means that the `flush` method is automatically invoked after a byte array is written, one of the `println` methods is invoked, or a newline character or byte ('\n') is written.

All characters printed by a `PrintStream` are converted into bytes using the given encoding or charset, or platform's default character encoding if not specified. The `PrintWriter` class should be used in situations that require writing characters rather than bytes.

Constructor	Description
<code>PrintStream(OutputStream out, boolean autoFlush, Charset charset)</code>	Creates a new print stream, with the specified <code>OutputStream</code> , automatic line flushing and charset.

Many other constructors:

- note the `Charset` argument, for conversion to strings

Printing anything

Mod. and Type	Method	Description
PrintStream	append(char c)	Appends the specified character to this output stream.
PrintStream	append(CharSequence csq)	Appends the specified character sequence to this output stream.
PrintStream	append(CharSequence csq, int start, int end)	Appends a subsequence of the specified character sequence to this output stream.
boolean	checkError()	Flushes the stream and checks its error state.
protected	clearError()	Clears the internal error state of this stream.
void		
PrintStream	format(String format, Object... args)	Writes a formatted string to this output stream using the specified format string and arguments.
PrintStream	format(Locale l, String format, Object... args)	Writes a formatted string to this output stream using the specified format string and arguments.
void	print(long l)	Prints a long integer.
void	print(Object obj)	Prints an object.
void	print(String s)	Prints a string.
PrintStream	printf(String format, Object... args)	A convenience method to write a formatted string to this output stream using the specified format string and arguments.

And many others...

PrintWriter

Basically, a `PrintStream` for underlying `char[]` streams:

Package `java.io`

Class PrintWriter

`java.lang.Object`
`java.io.Writer`
`java.io.PrintWriter`

Prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in `PrintStream`. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the `PrintStream` class, if automatic flushing is enabled it will be done only when one of the `println`, `printf`, or `format` methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking `checkError()`.

Socket I/O

Background

Basic notions:

- every **host** has a unique **IP address**
- every **process** on an host has a unique **port number**

TCP connection: a pair of "pipes" transferring bytes

- every byte sent on one endpoint reaches the other endpoint
- bytes arrives with the same order they have been sent
- no duplication

Server: a process waiting for connection requests on a port

Client: a process requesting a connection to a server

InetAddress

Package `java.net`

Class InetAddress

`java.lang.Object`

`java.net.InetAddress`

This class represents an Internet Protocol (IP) address.

Mod. and Type	Method	Description
static InetAddress[]	getByName(String host)	Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.
static InetAddress	getByAddress(byte[] addr)	Returns an InetAddress object given the raw IP address.
static InetAddress	getByName(String host, byte[] addr)	Creates an InetAddress based on the provided host name and IP address.
static InetAddress	getByName(String host)	Determines the IP address of a host, given the host's name.
static InetAddress	getLocalHost()	Returns the address of the local host.

No public constructors; many constructor-like methods.

Socket

Package `java.net`

Class `Socket`

`java.lang.Object`
`java.net.Socket`

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

Constructor	Description	
<code>Socket()</code>	Creates an unconnected Socket.	
<code>Socket(String host, int port)</code>	Creates a stream socket and connects it to the specified port number on the named host.	
<code>Socket(InetAddress address, int port)</code>	Creates a stream socket and connects it to the specified port number at the specified IP address.	
Mod. and Type	Method	Description
<code>void</code>	<code>close()</code>	Closes this socket.
<code>OutputStream</code>	<code>getOutputStream()</code>	Returns an output stream for this socket.
<code>InputStream</code>	<code>getInputStream()</code>	Returns an input stream for this socket.

Sample usage on **client side**:

```
Socket socket = new Socket("theserver.org", 10000);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
/* do I/O things */
```

ServerSocket

Package `java.net`

Class `ServerSocket`

`java.lang.Object`
`java.net.ServerSocket`

This class implements server sockets. A server socket waits for requests to come in over the network.

Constructor	Description
<code>ServerSocket(int port)</code>	Creates a server socket, bound to the specified port.

Mod. and Type	Method	Description
<code>Socket</code>	<code>accept()</code>	Listens for a connection to be made to this socket and accepts it.
<code>void</code>	<code>close()</code>	Closes this socket.

Sample usage on **server side**:

```
ServerSocket serverSocket = new ServerSocket(10000);
Socket socket = serverSocket.accept();
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
/* do I/O things */
```

Blocking `accept()`

```
public Socket accept() throws IOException
```

Listens for a connection to be made to this socket and accepts it. The method **blocks until a connection is made.**

Example: uppcaser server

Protocol (upon connection):

- client sends one text line l
- server replies with $l' = l$ converted to uppercase
 - if $l' = \text{BYE}$, server closes connection; otherwise waits for next line

Server:

- listens on port 7979
- handles 1 client at a time
- never terminates

(No bad things can happen: e.g., client does not close the connection.)

SimpleUppercaseServer

```
public class SimpleUppercaseServer {  
  
    private static final int PORT = 10000;  
    private static final String CLOSE_COMMAND = "BYE";  
  
    public static void main(String[] args) throws IOException {  
        ServerSocket serverSocket = new ServerSocket(PORT);  
        while (true) {  
            Socket socket = serverSocket.accept();  
            BufferedReader br = new BufferedReader(  
                new InputStreamReader(socket.getInputStream()));  
            BufferedWriter bw = new BufferedWriter(  
                new OutputStreamWriter(socket.getOutputStream()));  
            while (true) {  
                String line = br.readLine();  
                bw.write(line.toUpperCase() + System.lineSeparator());  
                bw.flush();  
                if (line.toUpperCase().equals(CLOSE_COMMAND)) {  
                    break;  
                }  
            }  
            socket.close();  
        }  
    }  
}
```

(imports omitted for brevity)

Socket `close()`

```
public void close() throws IOException
```

Closes this socket.

Once a socket has been closed, it is not available for further networking use (i.e. can't be reconnected or rebound). A new socket needs to be created.

Closing this socket will also close the socket's `InputStream` and `OutputStream`.

LineClient

```
public class LineClient {  
    public static void main(String[] args) throws IOException {  
        InetAddress serverInetAddress;  
        if (args.length > 1) {  
            serverInetAddress = InetAddress.getByName(args[0]);  
        } else {  
            serverInetAddress = InetAddress.getLocalHost();  
        }  
        Socket socket = new Socket(serverInetAddress, 10000);  
        BufferedReader br = new BufferedReader(  
            new InputStreamReader(socket.getInputStream()))  
        );  
        BufferedWriter bw = new BufferedWriter(  
            new OutputStreamWriter(socket.getOutputStream())  
        );  
        for (int i = 0; i < 10; i++) {  
            String sent = String.format("Hello world n. %d!", i);  
            bw.write(sent + System.lineSeparator());  
            bw.flush();  
            String received = br.readLine();  
            System.out.printf("Sent: %s%nReceived: %s%n",  
                sent, received  
            );  
        }  
        bw.write("bye" + System.lineSeparator());  
        bw.flush();  
        socket.close();  
    }  
}
```

(imports omitted for brevity)

Make it more general: line processing server

Protocol (upon connection):

- client sends one text line l
- if $l = l_{\text{quit}}$, server closes connection, otherwise replies with processed line $l' = p(l)$

Server:

- listens on port n_{port}
- handles 1 client at a time (motivation for **Simple** prefix...)
- never terminates
- **designed** to be extended
- $p : \text{String} \rightarrow \text{String}$, l_{quit} , port number are parameters

(No bad things can happen: e.g., client does not close the connection.)

377 / 419

SimpleLineProcessingServer

```
public class SimpleLineProcessingServer {  
  
    private final int port;  
    private final String quitCommand;  
  
    public SimpleLineProcessingServer(int port, String quitCommand) {  
        this.port = port;  
        this.quitCommand = quitCommand;  
    }  
  
    public void start() throws IOException {  
        ServerSocket serverSocket = new ServerSocket(port);  
        while (true) {  
            Socket socket = serverSocket.accept();  
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream));  
            BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream));  
            while (true) {  
                String line = br.readLine();  
                if (line.equals(quitCommand)) {  
                    break;  
                }  
                bw.write(process(line) + System.lineSeparator());  
                bw.flush();  
            }  
            socket.close();  
        }  
    }  
  
    protected String process(String input) {  
        return input;  
    }  
}
```

Method as parameter

$p : \text{String} \rightarrow \text{String}$ is a parameter.

p is valued by overriding `process()` while extending `SimpleLineProcessingServer`:

```
protected String process(String input) {  
    return input.toUpperCase();  
}
```

(We'll see another, radically different, option.)

Add some logging

Log on an `OutputStream` received as parameter:

- current date/time, client IP at connection
- current date/time, client IP, number of requests at disconnection

```
eric@cpu:~$ java SimpleLineProcessingServer 10000 bye
[2020-04-30 18:17:54] Connection from /127.0.0.1.
[2020-04-30 18:18:06] Disconnection of /127.0.0.1 after 2 requests
```

Constructor and fields

```
public class SimpleLineProcessingServer {  
    private final int port;  
    private final String quitCommand;  
    private final PrintStream ps;  
  
    public SimpleLineProcessingServer(int port, String quitCommand, OutputStream os)  
    {  
        this.port = port;  
        this.quitCommand = quitCommand;  
        ps = new PrintStream(os);  
    }  
  
    /* ... */  
}
```

We use a **PrintStream** because:

- we log text
- it does some buffering
- (we are lazy and don't care about errors while logging)

(No need to close **ps** here, because the server is supposed to never terminate.)

start() → start() + handleClient()

```
public void start() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
        Socket socket = serverSocket.accept();
        handleClient(socket);
    }
}

protected void handleClient(Socket socket) throws IOException {
    ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Connection from %2$s.%n",
    System.currentTimeMillis(),
    socket.getInetAddress().getHostAddress());
    BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
    int requestsCounter = 0;
    while (true) {
        String line = br.readLine();
        if (line.equals(quitCommand)) {
            break;
        }
        bw.write(process(line) + System.lineSeparator());
        bw.flush();
        requestsCounter = requestsCounter + 1;
    }
    socket.close();
    ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Disconnection of %2$s after %3$d requests.%n");
}
```

Might define a `private void log(String)` method.

(For serious logging, use `java.util.logging.Logger`)

Word counter server (~1h)

1. Design and implement a line processing server *application* (i.e., a class that extends `SimpleLineProcessingServer` with a `main()`) that:
 - returns the number of words in the input line l
 - exits with "bye"
 - listens on port 10000
2. Test it using telnet
 - try removing `flush()` call

(It likely won't work on repl.)

Vector processing server

Protocol (upon connection):

- client sends one real vector \vec{v}
- if $|\vec{v}| = 0$, server closes connection, otherwise sends as reply the processed line $\vec{v}' = p(\vec{v})$

Protocol detail: "send real vector" \vec{v}

1. send one **int** (4 bytes) $n = |\vec{v}|$
2. send n **doubles** (8 bytes each)

Server:

- listens on port 7979, handles 1 client at a time, never terminates

SimpleRealVectorProcessingServer: handleClient()

```
protected void handleClient(Socket socket) throws IOException {
    ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Connection from %2$s.%n",
    System.currentTimeMillis(),
    requestsCounter = 0;
    DataInputStream dis = new DataInputStream(new BufferedInputStream(socket.getInputStream()));
    DataOutputStream dos = new DataOutputStream(new BufferedOutputStream(socket.getOutputStream()));
    while (true) {
        double[] input = readVector(dis);
        if (input.length == 0) {
            break;
        }
        writeVector(dos, process(input));
        requestsCounter = requestsCounter + 1;
    }
    socket.close();
    ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Disconnection of %2$s after %3$d requests.%n");
}
```

Might extend [SimpleLineProcessingServer](#).

Protocol: `readVector()`, `writeVector()`

```
private double[] readVector(DataInputStream dis) throws IOException {
    int length = dis.readInt();
    double[] values = new double[length];
    for (int i = 0; i < values.length; i++) {
        values[i] = dis.readDouble();
    }
    return values;
}

private void writeVector(DataOutputStream dos, double[] values) throws IOException
    dos.writeInt(values.length);
    for (double value : values) {
        dos.writeDouble(value);
    }
    dos.flush();
}

protected double[] process(double[] input) {
    return input;
}
```

Might have defined `EnhancedDataInputStream`,
`EnhancedDataOutputStream`:

- with `double[] readDoubleArray()` and `void writeDoubleArray(double[])`

Message-based protocol

Previous examples implicitly defined a message-based protocol:

- `SimpleLineProcessingServer`: a message is a text line
- `SimpleRealVectorProcessingServer`: a message is vector encoded in a specified way

TCP is not message-oriented!

TCP: byte stream

TCP guarantees that:

- every byte sent on one endpoint reaches the other endpoint
- bytes arrive with the same order they have been sent
- no duplication

but it **does not guarantee** that one `write()` of m bytes on one endpoint corresponds to exactly one `read()` of m bytes on the other endpoint!

In practice:

- $n \geq 1$ `reads` might be needed to collect m bytes
- the last might include "other" bytes
- eventually, m bytes arrive

Example (of wrong solution)

Client C :

```
byte[] oBuffer = /* ... */  
os.write(oBuffer);
```

Assume oBuffer is $l \leq 1024$ at runtime.

Server S connected to C :

```
byte[] iBuffer = new byte[1024];  
int n = is.read(iBuffer);
```

- n might be something between 1 and l
- $iBuffer$ might actually contain only a (leading) portion of what oBuffer contained on the other endpoint!

Even worse

Client C ($\text{oBuffer1}, \text{oBuffer2}$ lengths are l_1, l_2):

```
os.write(oBuffer1);
os.write(oBuffer2);
```

Server S connected to C :

```
int n1 = is.read(iBuffer1);
int n2 = is.read(iBuffer2);
```

Possible outcomes:

- $n1 = l_1, n2 = l_2$ (fluke!)
- $n1 \leq l_1, n2 \leq l_2$
- $n1 > l_1, n2 \leq l_2 - (n1 - l_1)$
- ...

Protocol with `byte[]` messages

You cannot assume that 1 `read()` gives 1 message!

You need a protocol for taking the `byte[]` resulting from the concatenation of 1+ `read()`s and dividing it in **messages**.

If your protocol works with higher level encoding, this job can be partially done by JDK classes. E.g.:

- text lines: `BufferedReader` reads as many bytes for reaching the new line character
- double array: the first 4 bytes (`DataInputStream`) specify how many other bytes need to be read (at least)

Multithreading

392 / 419

Get rid of Simple...

```
public void start() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
        Socket socket = serverSocket.accept();
        handleClient(socket);
    }
}
```

`accept()` wait for connection requests and blocks until one client sends one.

`handleClient()` returns only upon completion of client handling.



This server handles at most 1 client at a time.

One at a time

Possible other clients hang until the handled one disconnects.

1. S is on `accept()`
2. Client C_1 connects to S ; S is on `handleClient()`
3. Client C_2 "connects" to S ; OS of S takes the connection request and waits for S to handle it
4. ...
5. C_1 says `bye`; S goes to `accept()`

This server is not particularly useful!

- that's why we called it `Simple...Server`

Goal

A server that can handle many client at a time.

```
public void start() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
        Socket socket = serverSocket.accept();
        handleClient(socket);
    }
}
```

$1 + n$ **concurrent** processes:

- one "always" waiting for connection requests (`accept()`)
- n , one for each client (`handleClient()`)

Thread

Thread = execution flow

- the JVM can execute more than one thread at a time
- a thread can be blocked (e.g., waiting for input) while the others continue to run

Process vs. thread:

- processes are concurrent flows managed by the OS
- threads are concurrent flows managed by the JVM
- both the JVM and the OS can exploit the hardware to actually run flows at the same time
 - (otherwise) they time-share the CPU

Thread

Package `java.lang`

Class Thread

`java.lang.Object`
`java.lang.Thread`

A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of `Thread`. This subclass should override the `run` method of class `Thread`. An instance of the subclass can then be allocated and started.

Mod. and Type	Method Description
<code>void</code>	<code>start()</code> Causes this thread to begin execution; the Java Virtual Machine calls the <code>run</code> method of this thread.
<code>void</code>	<code>run()</code> If this thread was constructed using a separate <code>Runnable</code> run object, then that <code>Runnable</code> object's <code>run</code> method is called; otherwise, this method does nothing and returns.

Two methods: we see the first.

1. define a class T that extends `Thread` and overrides `run()`
2. invoke $T.start()$ (that will invoke `run()`)

Example

```
public class SlowCounter extends Thread {  
    public void run() {  
        for (int i = 0; i < 10000; i++) {  
            if (i % 1000 == 0) {  
                System.out.printf("%s: %2d%n", toString(), i / 1000);  
            }  
        }  
    }  
}
```

```
SlowCounter c1 = new SlowCounter();  
SlowCounter c2 = new SlowCounter();  
c1.start();  
c2.start();
```

```
Thread[Thread-0,5,main]:  0  
Thread[Thread-1,5,main]:  0  
Thread[Thread-0,5,main]:  1  
Thread[Thread-1,5,main]:  1  
Thread[Thread-0,5,main]:  2  
Thread[Thread-1,5,main]:  2  
Thread[Thread-0,5,main]:  3  
Thread[Thread-1,5,main]:  3  
Thread[Thread-0,5,main]:  4  
Thread[Thread-1,5,main]:  4  
Thread[Thread-0,5,main]:  5  
...  
Thread[Thread-0,5,main]:  9  
Thread[Thread-1,5,main]:  9
```

The JVM executes until the last thread has ended.

398 / 419

start()

```
SlowCounter c1 = new SlowCounter();
SlowCounter c2 = new SlowCounter();
c1.start();
c2.start();
```

When `start()` is invoked:

1. the JVM starts a new thread, i.e., a new execution flow
2. the execution flow of the caller **immediately returns**
3. the new execution flow goes on with `run()`

(Thread is the execution flow; `Thread` is the class.)

start() vs. run()

Invoking `run()` **does not** cause the JVM to start a new thread!

```
SlowCounter c1 = new SlowCounter();
SlowCounter c2 = new SlowCounter();
c1.run();
c2.run();
```

```
Thread[Thread-0,5,main]: 0
Thread[Thread-0,5,main]: 1
Thread[Thread-0,5,main]: 2
Thread[Thread-0,5,main]: 3
Thread[Thread-0,5,main]: 4
Thread[Thread-0,5,main]: 5
Thread[Thread-0,5,main]: 6
Thread[Thread-0,5,main]: 7
Thread[Thread-0,5,main]: 8
Thread[Thread-0,5,main]: 9
Thread[Thread-1,5,main]: 0
Thread[Thread-1,5,main]: 1
Thread[Thread-1,5,main]: 2
Thread[Thread-1,5,main]: 3
Thread[Thread-1,5,main]: 4
Thread[Thread-1,5,main]: 5
Thread[Thread-1,5,main]: 6
Thread[Thread-1,5,main]: 7
Thread[Thread-1,5,main]: 8
Thread[Thread-1,5,main]: 9
```

Thread and memory

Each thread:

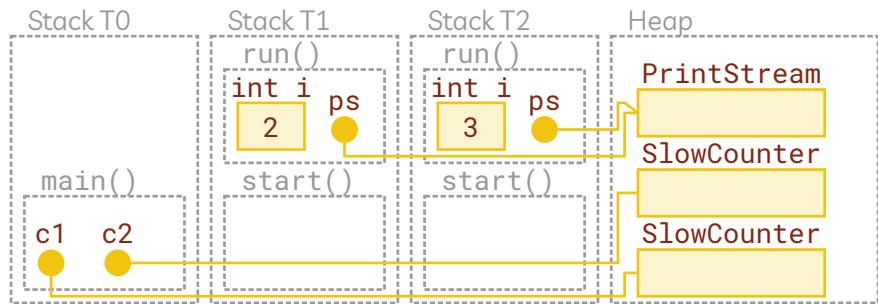
- has **its own stack**
- share the **unique heap**

Diagram

```
class SlowCounter extends Thread {  
    public void run() {  
        PrintStream ps = System.out;  
        for (int i = 0; i < 10; i++) {  
            ps.println(i);  
        }  
    }  
}
```

```
SlowCounter c1 = new SlowCounter();  
SlowCounter c2 = new SlowCounter();  
c1.start();  
c2.start();
```

At some point during the execution:



out is static in System!

Good practice

Assume C extends `Thread` is the class that does the job concurrently.

Put in C constructor all and only the things that has to be shared.

Multithreaded

LineProcessingServer

```
public class LineProcessingServer {  
    private final int port;  
    private final String quitCommand;  
  
    public LineProcessingServer(int port, String quitCommand) {  
        this.port = port;  
        this.quitCommand = quitCommand;  
    }  
  
    public void start() throws IOException {  
        ServerSocket serverSocket = new ServerSocket(port);  
        while (true) {  
            Socket socket = serverSocket.accept();  
            ClientHandler clientHandler = new ClientHandler(socket, quitCommand);  
            clientHandler.start();  
        }  
    }  
}
```

ClientHandler

```
public class ClientHandler extends Thread {  
    private final Socket socket;  
    private final String quitCommand;  
  
    public ClientHandler(Socket socket, String quitCommand) {  
        this.socket = socket;  
        this.quitCommand = quitCommand;  
    }  
  
    public void run() {  
        /* ... */  
    }  
  
    protected String process(String input) {  
        return input;  
    }  
}
```

Note: `run()` signature cannot be modified

- cannot add `throws IOException`

ClientHandler.run()

```
public void run() {
    try {
        BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
        while (true) {
            String line = br.readLine();
            if (line.equals(quitCommand)) {
                socket.close();
                break;
            }
            bw.write(process(line) + System.lineSeparator());
            bw.flush();
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
            socket.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Basically the same as before...

- with a lot of **try**, **catch**, **finally**... because **throws** cannot be used; we'll see

"Usage" of `LineProcessingServer`

1. Define a class H that:

- extends `ClientHandler`
- overrides `process()`

2. Define a class S that:

- extends (or "is like") `LineProcessingServer`
- instantiates H

A bit cumbersome:

- word counter: define H_1 and S_1
- uppcaser: define H_2 and S_2

Concurrent execution

The **same method** of the **same object** can be executed at the same time by different threads!

Idea:

- the server class S specify the application details:
 - how to process one request
 - what is the quit command
- **ClientHandler** just knows how to handle one client
 - i.e., manage the request-response interaction
 - delegates to S for actual server behavior

Better LineProcessingServer

```
public class LineProcessingServer {  
    private final int port;  
    private final String quitCommand;  
  
    public LineProcessingServer(int port, String quitCommand) {  
        this.port = port;  
        this.quitCommand = quitCommand;  
    }  
  
    public void start() throws IOException {  
        ServerSocket serverSocket = new ServerSocket(port);  
        while (true) {  
            Socket socket = serverSocket.accept();  
            ClientHandler clientHandler = new ClientHandler(socket, quitCommand);  
            clientHandler.start();  
        }  
    }  
  
    public String process(String input) {  
        return input;  
    }  
  
    public String getQuitCommand() {  
        return quitCommand;  
    }  
}
```

Agnostic ClientHandler

```
public class ClientHandler extends Thread {  
    private final Socket socket;  
    private final LineProcessingServer server;  
  
    public ClientHandler(Socket socket, LineProcessingServer server) {  
        this.socket = socket;  
        this.server = server;  
    }  
  
    public void run() {  
        try {  
            BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
            BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));  
            while (true) {  
                String line = br.readLine();  
                if (line.equals(server.getQuitCommand())) {  
                    socket.close();  
                    break;  
                }  
                bw.write(server.process(line) + System.lineSeparator());  
                bw.flush();  
            }  
        } catch (IOException e) { /* ... */}  
        finally { /* ... */}  
    }  
}
```

"Usage" of better `LineProcessingServer`

1. Define a class S that:

- extends `LineProcessingServer` and overrides `process()`

Elegant!

Note:

- at runtime, many threads may execute `process()` at the same time

Possible source of big problems!

Thread scheduling

The JVM:

- knows the set T of existing threads and at which **bytecode instruction** they are
- knows the subset $T' \subseteq T$ of not blocked threads
- knows the set C of available cores
 - hardware or virtualized, provided by the OS

At each time step:

- for each core in C
 - select one thread in T' and executes the next bytecode instruction

Thread interference

⇒ threads execute concurrently only apparently!

Suppose:

- only one core ($|C| = 1$)
- thread t_1 next instructions: a_1, b_1, c_1
- thread t_1 next instructions: a_2, b_2, c_2

Some possible actual executions:

- $a_1, b_1, c_1, a_2, b_2, c_2$
- $a_1, a_2, b_1, b_2, c_1, c_2$
- $a_2, b_2, a_1, c_2, b_1, c_1$
- ...

[Is $a_2, b_1, a_1, c_2, b_2, c_1$ possible?]

Bytecode and Java statements

One Java statement:

```
System.out.printf("a+b=%d%n", a + b);
```

→>> 1 bytecode instructions!

- in a multithreaded execution, this statement execution might be interleaved with other statements execution

Even worse!

- exact sequence of execution is **not predictable!**
- source of very bad bugs:
 - **not easily reproducible**
 - difficult to spot

Counter example

```
public class Counter {  
    private int c = 0;  
    public int incAndGet() {  
        c = c + 1;  
        return c;  
    }  
    public int decAndGet() {  
        c = c - 1;  
        return c;  
    }  
}
```

```
Counter c = new Counter();  
(new IncThread()).start();  
(new DecThread()).start();
```

```
public class IncThread extends Thread {  
    private final Counter c;  
    public CounterThread(Counter c) {  
        this.c = c;  
    }  
    public void run() {  
        System.out.print(c.inc() + " ");  
    }  
}  
  
public class DecThread extends Thread {  
    /* ... */  
}
```

Possible outputs:

- 1 0 ("lucky")
- 0 0
- -1 0

Eventually c=0; meanwhile could be -1, 0, 1.

synchronized

Methods defined with the **synchronized** modifier cannot be executed **on the same object** by more than one thread **at the same time**.

- from the point of view of the executing thread, the method execution is **atomic**

Atomicity is guaranteed just on the method!

AtomicCounter

```
public class AtomicCounter {  
    private int c = 0;  
    public synchronized int incAndGet() {  
        c = c + 1;  
        return c;  
    }  
    public synchronized int decAndGet() {  
        c = c - 1;  
        return c;  
    }  
}
```

```
Counter c = new Counter();  
(new IncThread()).start();  
(new DecThread()).start();
```

```
public class IncThread extends Thread {  
    private final Counter c;  
    public CounterThread(Counter c) {  
        this.c = c;  
    }  
    public void run() {  
        System.out.print(c.inc() + " ");  
    }  
}  
  
public class DecThread extends Thread {  
    /* . . . */  
}
```

Possible outputs:

- 1 0
- -1 0

0 0 is no more possible!

Thread-safe classes

A method invocation outcome in case of multiple threads:

- may be not specified (i.e., not predictable)
- may cause an exception to be thrown

The method/class is said to be not **thread-safe**.

Since atomicity is a common requirement, the JDK provides:

- thread-safe versions of some key classes
 - e.g., [AtomicInteger](#)
- ways to make thread-safe some non thread-safe classes
 - e.g., [Collections.synchronizedCollection\(\)](#)

synchronized block

`synchronized` can be applied to code blocks, instead of to entire methods:

```
Counter c = /* ... */  
for (int i = 0; i < 10; i++) {  
    synchronized (c) {  
        System.out.print(c.get() + " -> ");  
        c.inc();  
        System.out.println(c.get());  
    }  
}
```

The synchronized block can be executed on the same `Counter` object (`c`, here) by at most one thread at a time.

[What's the output with/without `synchronized` with two threads on the same counter?]

[What if `c` is instantiated here?]