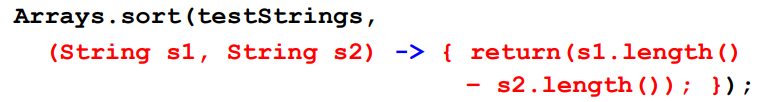
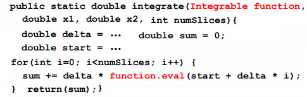


It knows 2nd arg is Comparator<T> and type is string. Reduction to:



Function is not arg. Instance of class that implement the interface is the argument. In lambda, you are implementing 1 abstract method of the interface.Numerical integration ex

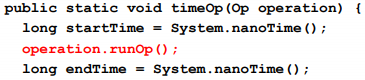


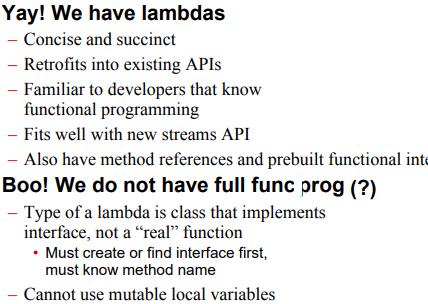
In test, call Util.integrate(f, x1,x2..)

We are defining Integrable int.’s eval func, as it takes x, ret. x\*2 or x\*3. In integrate func, it calculates pass arg (st+delta\*i) and takes return value.



We are impl. runOp method of the OP interface. runOP takes no parameters (void runOp()). We say that, it will sort array. Time calculation is easier.





**@FunctionalInterface,SAM** To check for compile errors.

*Public interface MyInt{*

*void print(Str a); // Single abst. met.*

*def. Pub. Void p2(...){*

*//some implemented method}*

*stat. Void p3(...){*

*// some imp. Static method}}*

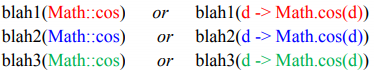
This is still SAM. Only 1 abstract met. It can be used for lambda exp.

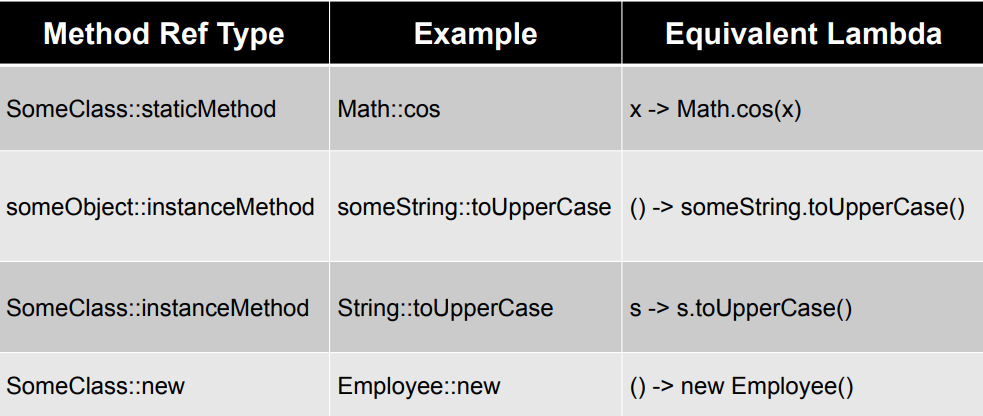
*MyInt m = (Str t) -> Syso(t);* // Here, we are implementing print method of the MyInt interface. We can still use: *m.p2(...) or MyInt.p3(...)* improvemts:



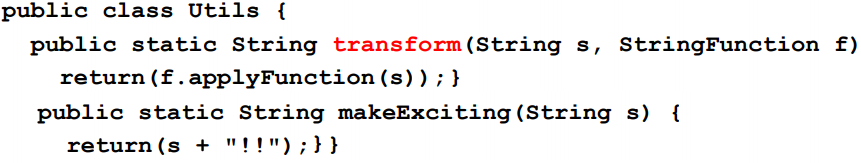
Sin and exp methods are static, use Math::sin. Take x as param, ret. sinx. The ret type of Math::sin determined from the context. Since intTest expects Integrable, it is Integrable.

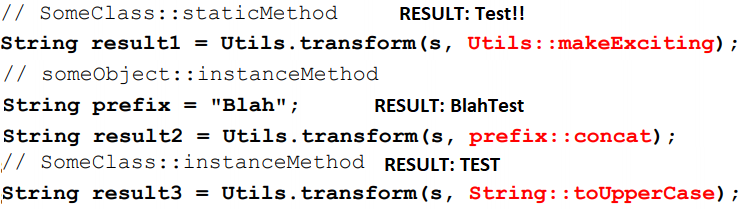
*Integ. i1 = Math::sin; intTest(i1, 0, 2);*











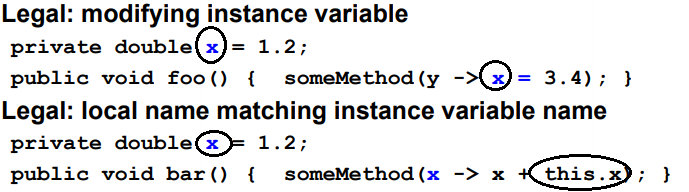
*Old:* StrFunc sf = s->Utils.makeEx(s);

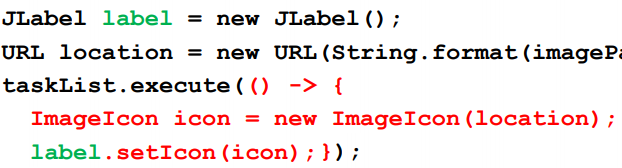
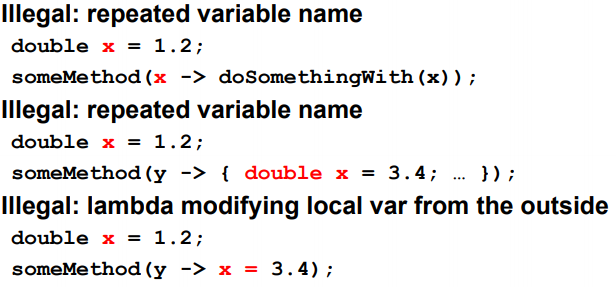
Syso(sf.applyFunc(“some str”));

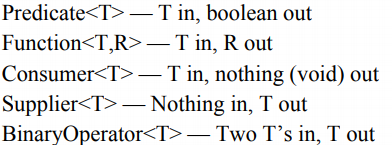
*New:* StrFunc sf2 = Utils::makeEx;

Syso(sf2.applyFunc(“some str”));

We are impl. applyFunction met. of the StringFunction int.We cant modify local vars. They must be (effective final). But you can change assignment variables inside that object:

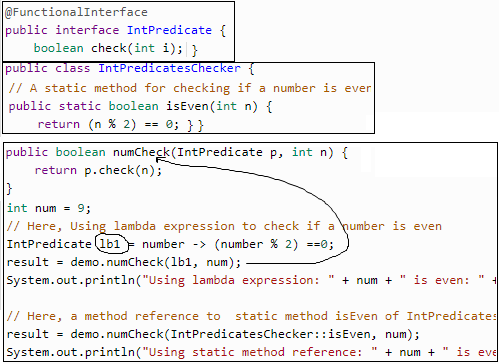


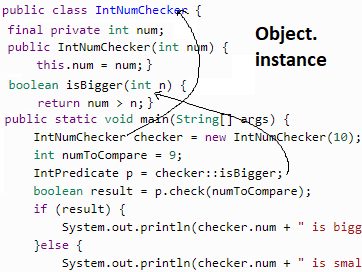


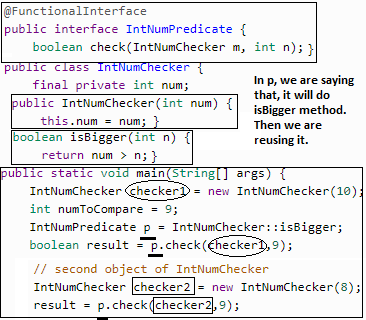


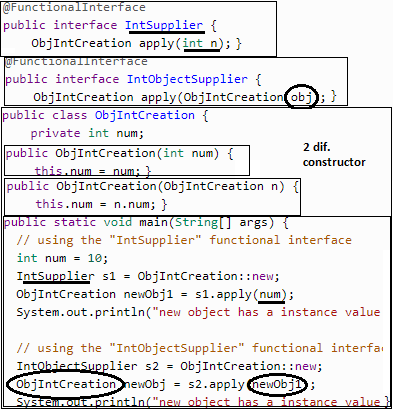
Ex: *DoubleBinaryOperator f = (d1,d2) -> Math.cos(d1 + d2);*

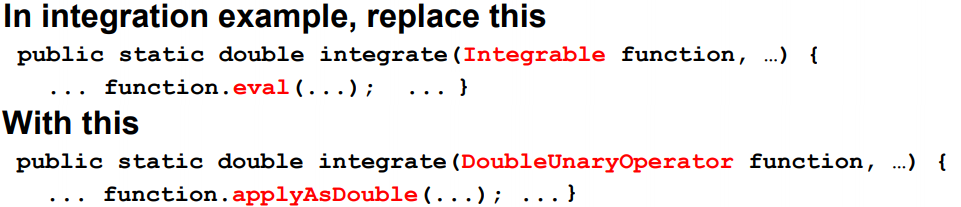
We can still use the same test met.



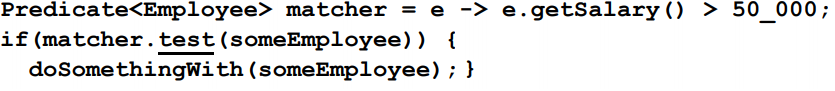




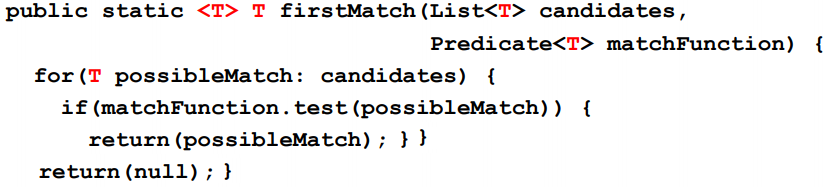


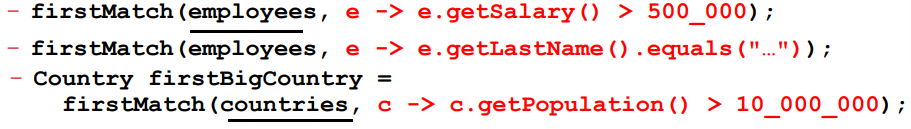


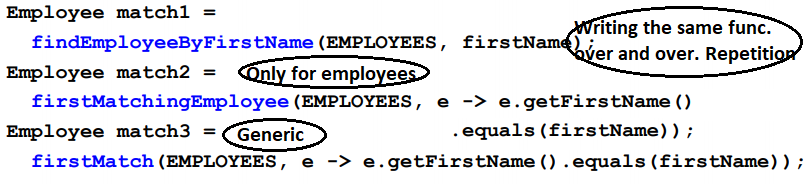
**Predicate:** test a condition



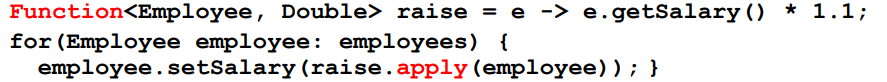
Improv. in predicate, make it Generic



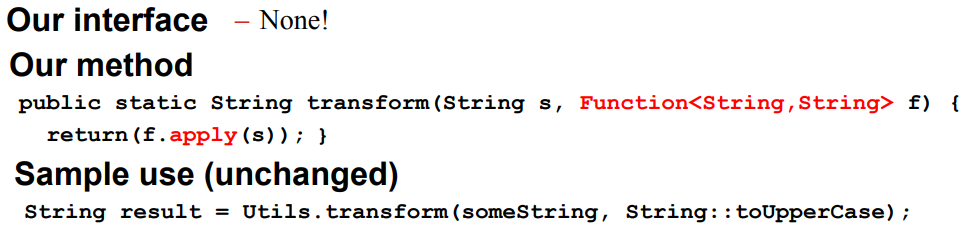
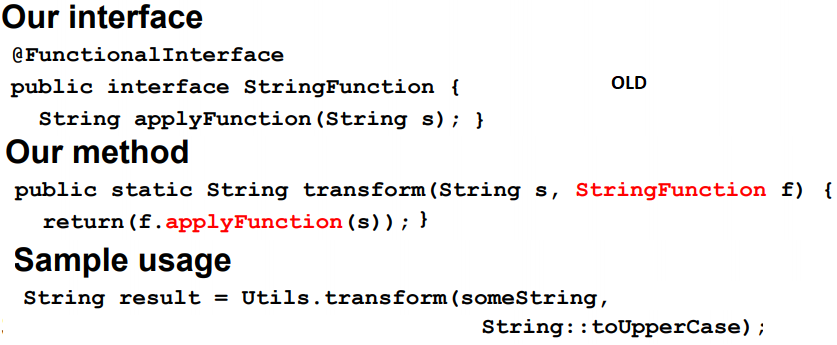


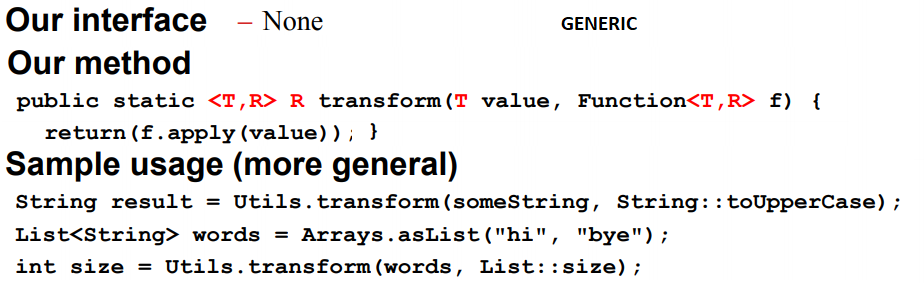
Overall improvements:  


**Function**:take T ret R, R apply(T t)

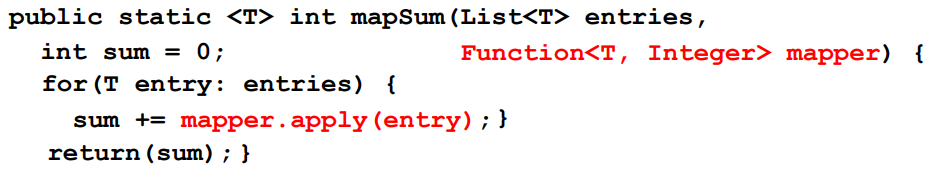


Old vs ref1 vs ref2 using Func int.

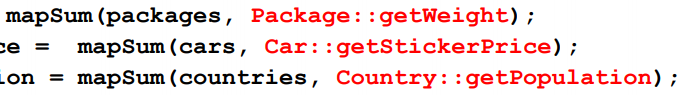




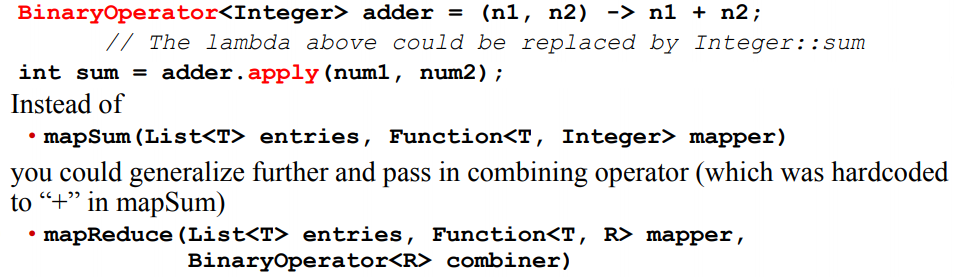
Without Function, we need two seperate function for sum up Em.salary and country. Population. Using Function int., we can define:



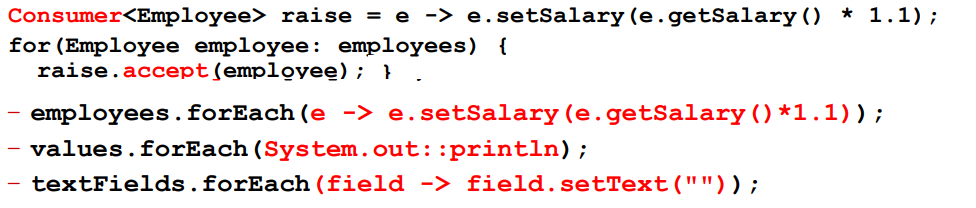
And call this function by:



We can further improve this function. In mapSum, we are using + operator. Instead of +, we can use any combiner(BinaryOperator interface)

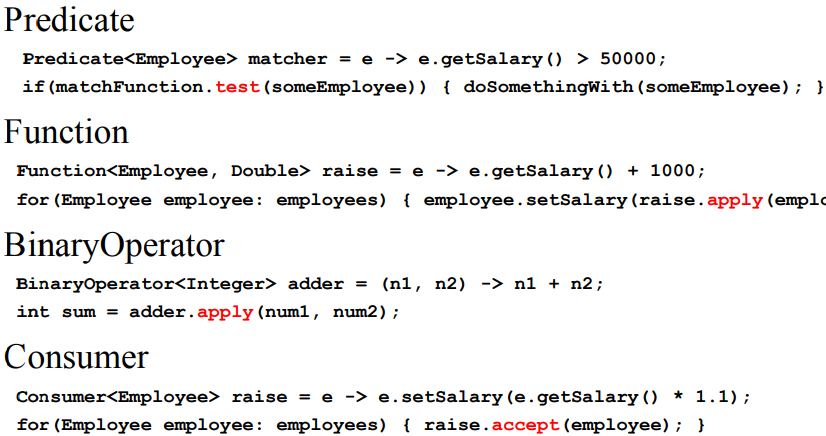


**Consumer**

****

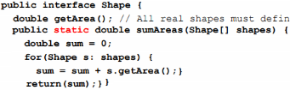
**Supplier:**

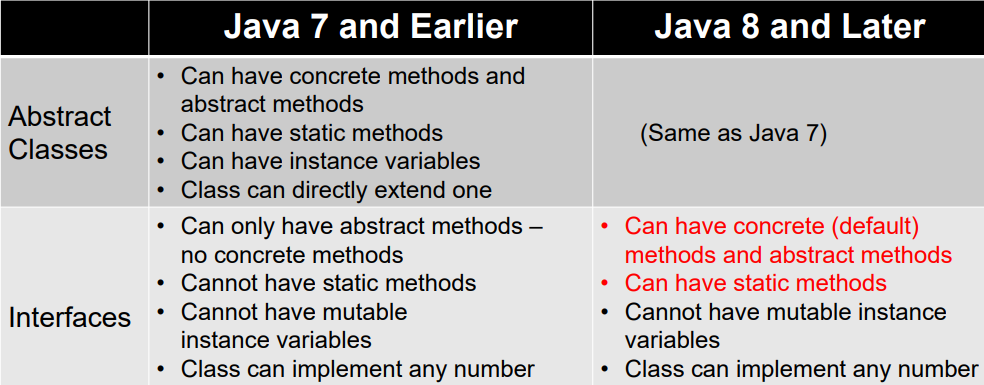
****



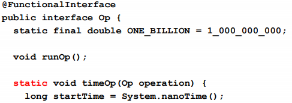
**STATIC METHOD** In prev ex, we had Shape abs. Cl. and a ShapeUtils

class that has sumAreas method. Now

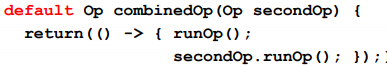


****

No ShapeUtils cl. Sh. has 1 abs. met., SAM. old, we had TimingUtils meas. Op time by timeOP, 1.sütun. Now, we can put timeOp in Op interface

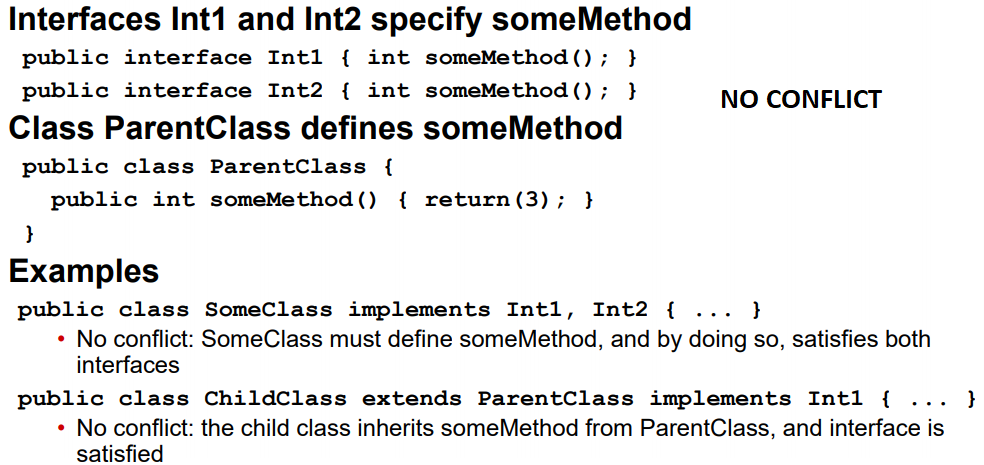


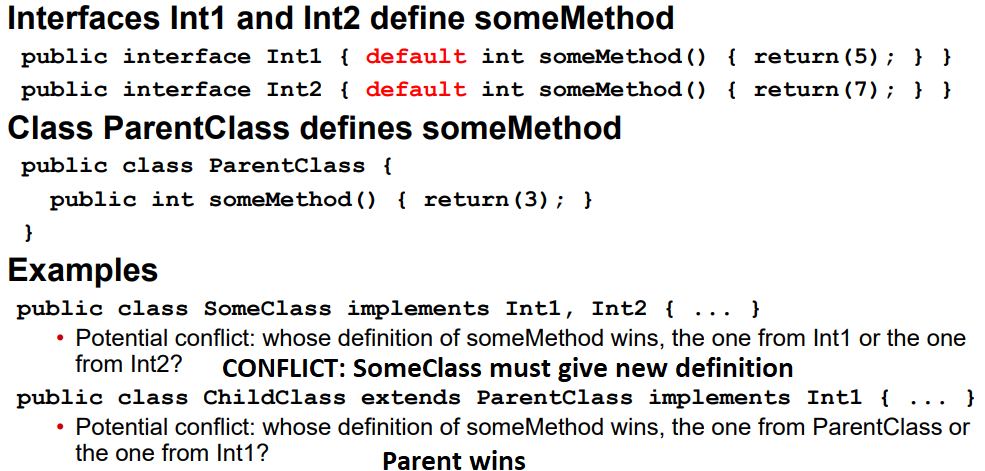
To measure time, use *Op.timeop(()-> meth());* ONE\_BILLION is final &public, we cant have private and mutable member in int. Int. is for public. **DEFAULT METHOD** in Op int.

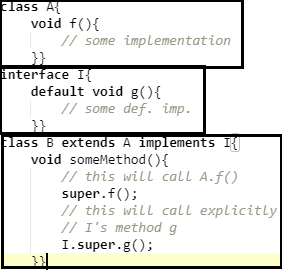




1 sortArr.,2 wasteT. op **CONFLICTS**



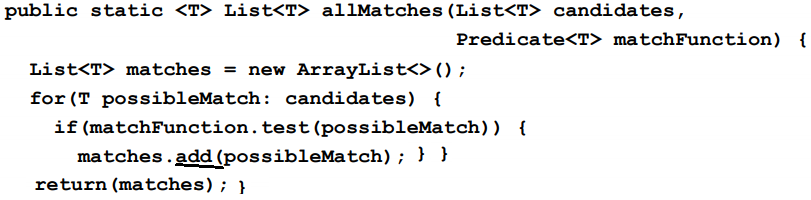




**Higher Order Functions: Method that Ret. Functions**



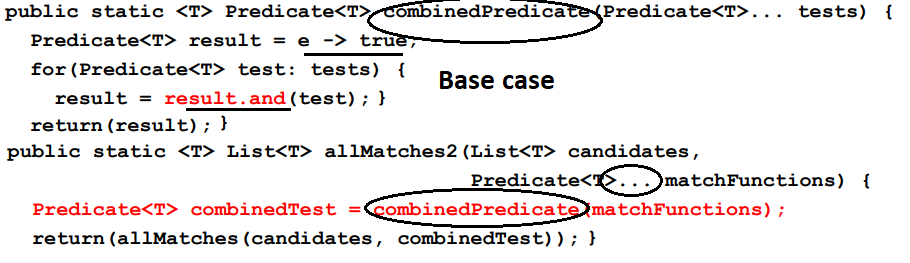
**Predicate**: and, or, negate, isEqual



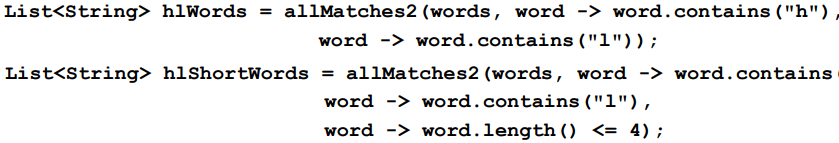
Above, only 1 func. Usage:



Impr. for any number of predicates:



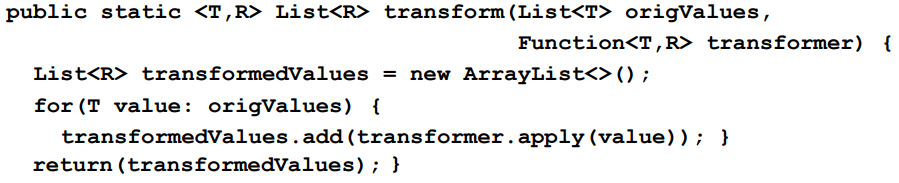
Usage in improvement allMatches2

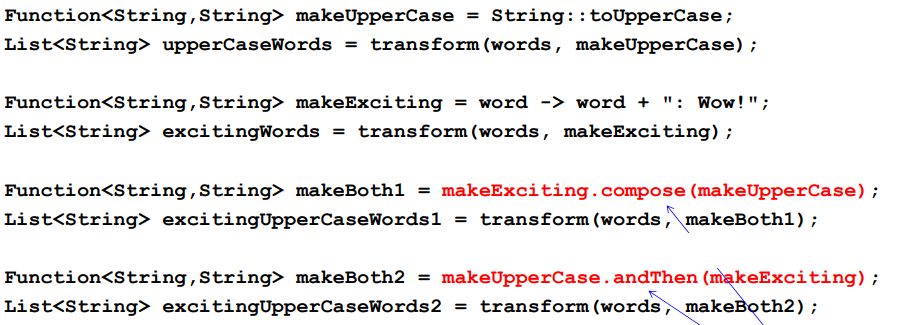


**And**, is a met. of the int. that takes 2 param, 1 this obj, 2 param. obj. Ret. 1&2 of these (similar to combineOp)

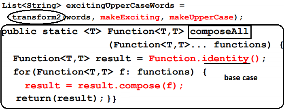
**Function**: compose, andThen, identity. f1.compose(f2), first f2, then pass the result to apply method of f1.

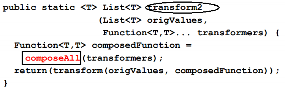
F1.andThen(f2), first f1, then pass the result to f2’s apply method. 1. sütun orta transform met.:

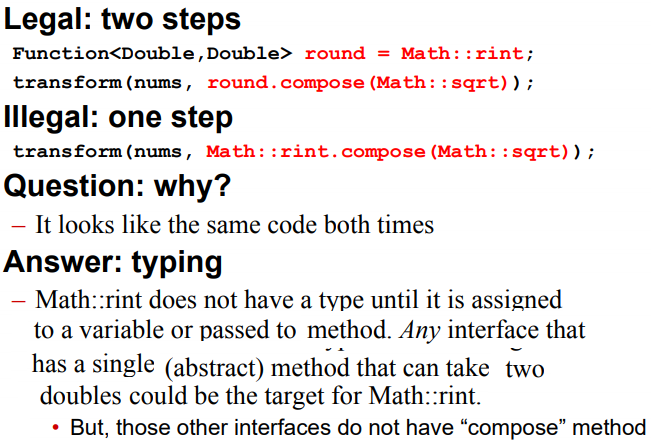




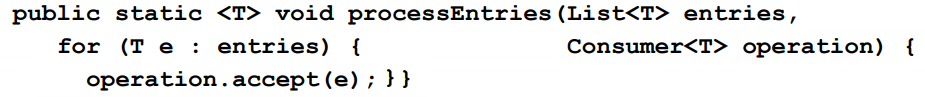
Improvements: compose, andThen methodlarını dışarıda yazmak yerine şu şekilde kullanabiliriz:

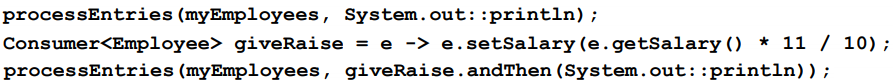




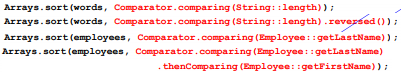


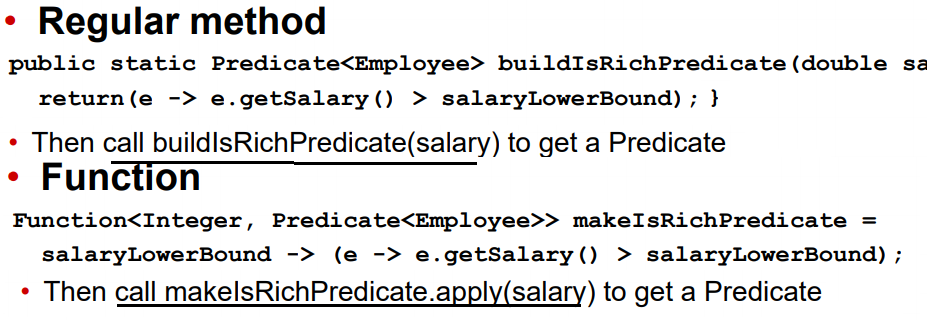
**CONSUMER**: andThen, accept





**COMPARATOR**: comparing, reversed, thenComparing

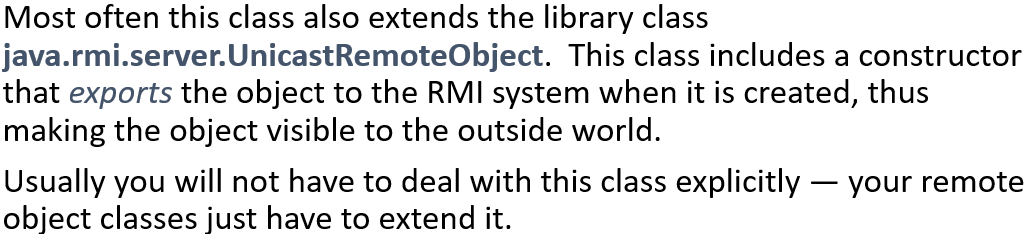


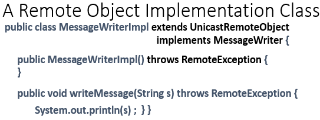


**RMI, Remote Method Invocation** Calling remotely as if it is here. 1st pass args to remote comp. 2nd get ret. value. *Res = obj.met(arg);* //obj is remote ref. Remote int. is needed. Local knows that those methods are impl.(protocol).Remote int. is a normal Java int., which must extent the marker int. ..*Remote.* All methods in a remote int. must be declared to throw the *RemoteEx*. for issues in network or RMI mechanism.



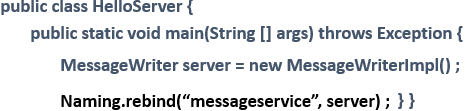
Remote has NO methods or fields. Just a marker, tells RMI to treat the int as remote int. A remote obj. is an instance of a cl that imp. a remote int.



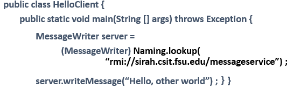


To compile, *rmic MessageWriterImpl*.

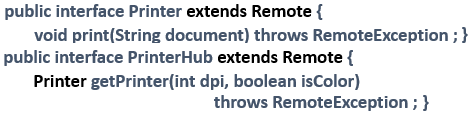
Until now, we have the obj. We need client and server. Assume that both c and s have copies of all class files.

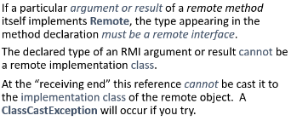


Rem. obj name server. It is interface! Naming.rebind puts a ref to RMI registery. Clients will get that ref.

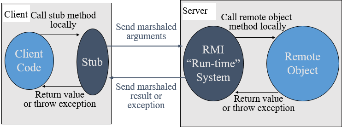


Client obj. is interface too! port: 1099 Compile server and client. Args(any type of) is collected and shipped to the server. Results/ ex. are shipped back to client. Client can obtain another rem. obj. ref. through RMI.





**STUBS:** Remote ref. is ref to local obj. that imp. The same remote int. as the remote obj.Local obj is called stub Client invocations are actually local invocations on the stub class. Args. and res. ser.(marshalled) for sending.



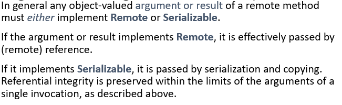
Input to rmic is remote imp. class.

RMI run time finds the object with help of reflection. Stub will extract results/exception. Stub is called proxy. We think we are dealing with remote obj., actually we are dealing with local obj. Stub has the same int. When stub code is generated, it is final class. Stub is wrapper for remote int. Compile-time type safety. It has port, name of host, unique id. It keeps the state, not the code. **Serialization(short ver.)**

Write out the root of the tree:

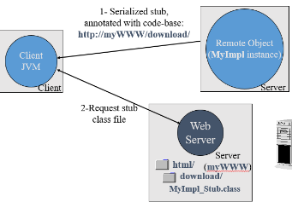
*out.writeObject(d) ;//* out and in could be network, pipe, file. Series of bytes. Read later: *Node e =(Node) in.readObject()* Original tree is reprocuded. E will have the copy of the original structure. In the graph, a is ref. twice. While reprod., it will not create a 2. It uses cache & know that a’ is created, it finds that a’ and puts a ref. Serialization is recursive process.



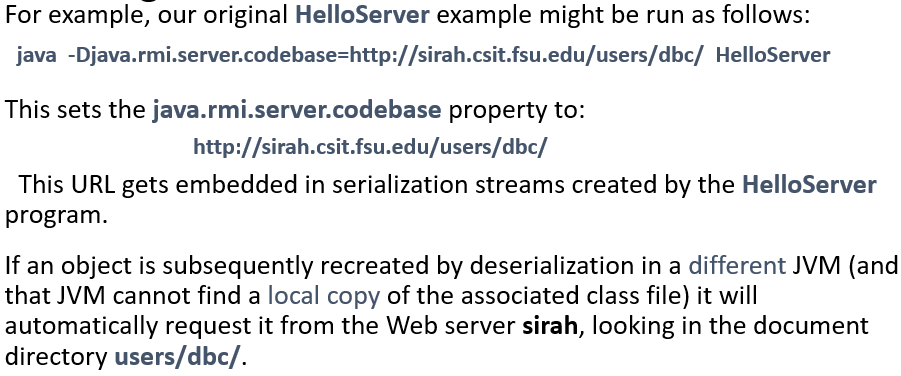
**Dynamic Class Loading**

Serialized obj. dont contain the actual JVM instructions (the byte code), that impl. methods on the local object. Stores ALL data fields, not any code. When obj. is serialized, only the state. When deserialized, if it cannot find cl. file UnmarshallEx, ClassNotFoundEx.

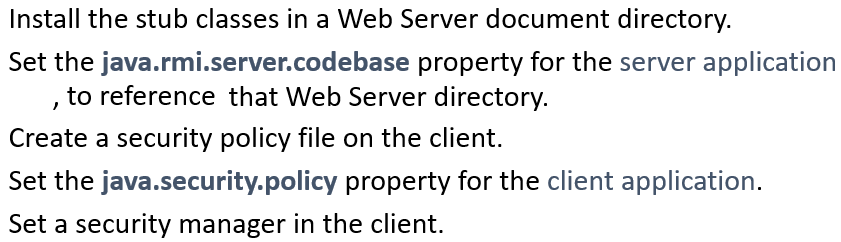
Two way to get class file: 1, manually copy class files for all stub classes (either classpath or to current dir.) 2, put it in webserver(FTP)



You request the server, it(RMI registery) says URL, you download MyImplstub.cl(compiled). How to put annotation(URL)? Codebase



Before downloading, Policy: *grant {permission java.security. AllPer. “”, “” ;} ;*, all perm. to everyone. Bad *java –Djava.security.policy=policy.all HelloClient.* **Summary of Policy&man**.



**RMI Registry**, stores ref of remote obj. Remote obj. might have a remote method. It can be called from the client. Synch. is same as multith.

**Garbage collection**, local: if local obj is not referenced, it is not accessable, COULD be garbage collected. In remote, remote ref. layer holds reference count. Client says that “I dont need that obj, count--, if count=0, garbage collection”. Client may fail. Solution: Renew a lease **Arguments**:

***Local primitive****:* int->Integer(boxed)

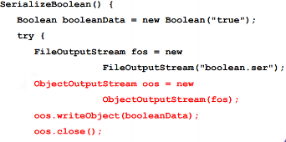
***Local obj:*** serialize(it should imp.

Serializable), values are copied(pass by value) ***Remote obj:*** You have its stub(remote ref. to remote obj) Pass by remote reference(stub). Stub is serialized and send(like local obj). Not the remote obj is sent. *Ex*: Local has y=f(a), a is stub of the remote obj. a is serialized and sent to server. It deserialize it, downloads .class from URL, construct a’ (copy of that object). Now, it knows how to access remote obj. It uses ex: a.h(), another remote obj is called. Rem. ref. is a stub, it knows how to access and where of the original object. **Serialization** We don’t need meth., we need state, class info(which class) Serializable is inherited.

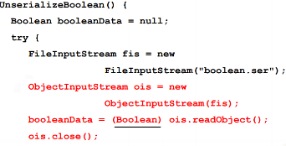
ObjectOutputStream-> ser.(flattening)

ObjectInputStream->deser.(reconst)

Class should imp. Serializable int. Also needs to provide a default const(with no args) NotSerializableEx. is thrown if you try to serialize non-serializable obj. **Transient**: a thread is non-serializable class. If your class has thread, you don’t want to serialize thread, o.w. exception. Or, some big array that you don’t want to serialize.



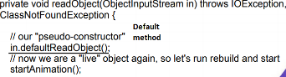
Read obj: it returns Object, typecast it



**Version problem:** If you update the class file( a new field), try to read serialized obj, InvalidClassEx. all cl. maintains in a field called *serialVersionUID*. You can manually control it. *serialver MyClass*

*->MyClass static final long serialVersionUID =1123123L*; Copy number in class so that anycode containing field assumes it has that version. It is about serialization, not about versioning.

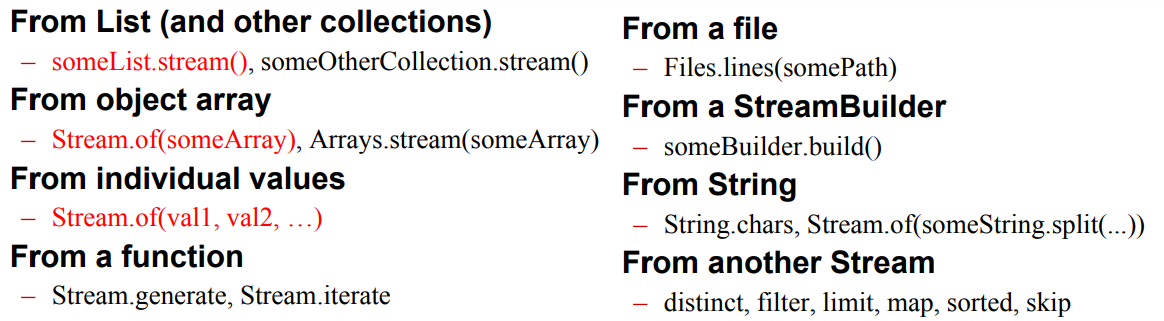
**Own Write/ ReadObj().**You still need default imp. + some extra->animation

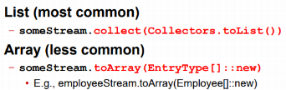


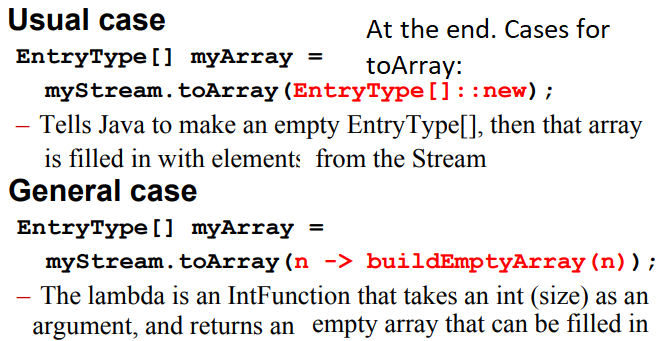
**Externalizable**: Instead of imp Serializable, imp. Externalizable. You must implement writeExternal, readExternal methods as you like. It is your protocol. Performance is better since serializable uses reflection. **Streams** Lazy eval, auto parallel, infinite. They don’t store data. List, file, array store the data. A,b,c in 1by1, a’,b’,c’ out in the same order, 1by1. IOStreams: InputStream, OutputStream, oos...(in serialization). Java8: Stream<T>. (ex: Stream<Str>). Also, IntStream, DoubleStream for numbers. Support lambda(take lambda as arg). Streams do not modify underlying data structures, the expression modifies. No index access. Can output as Lists or Arrays.

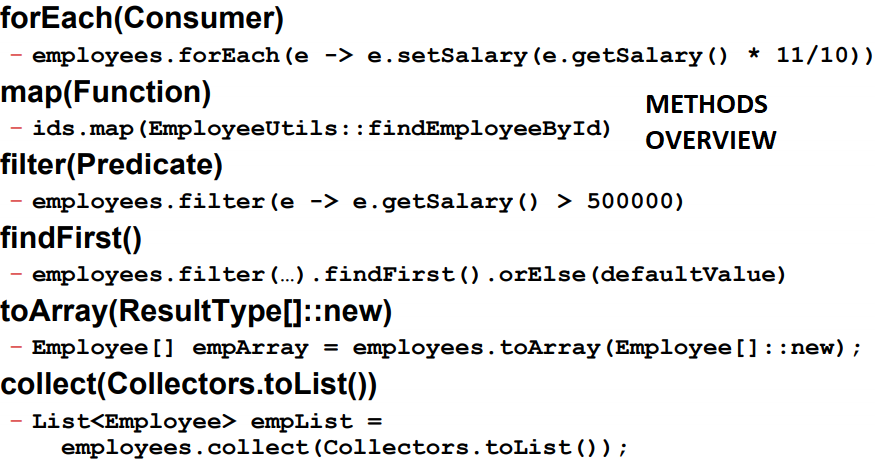


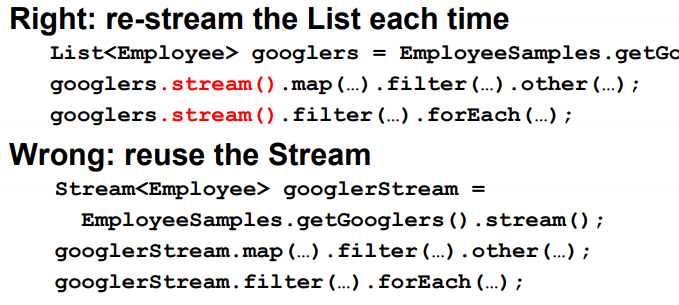
When you build a stream, it does not copy the data. It builds pipeline of operations. Emp e1 = …; Emp e2 = …; Stream.of(e1,e2,...).map(…).other(…);



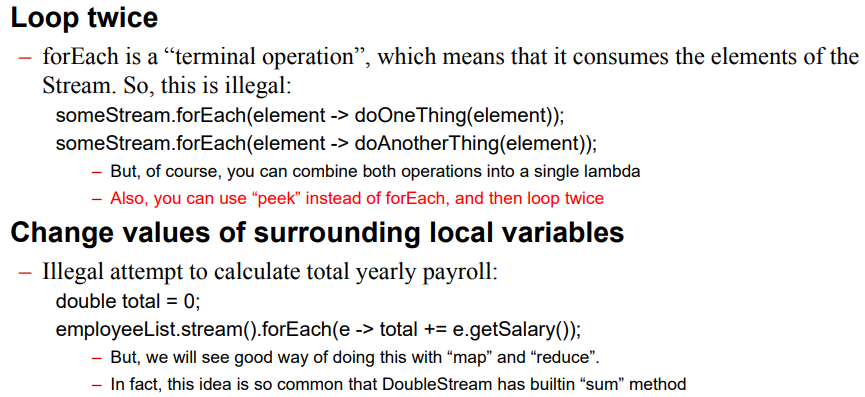


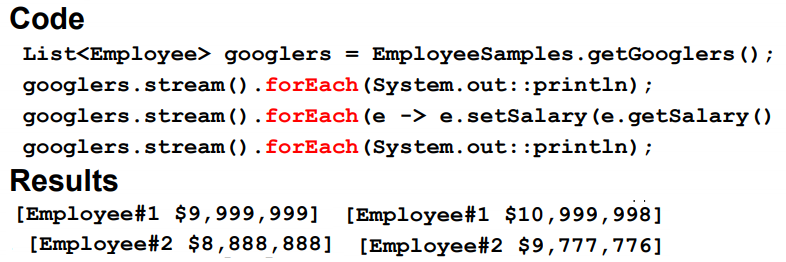


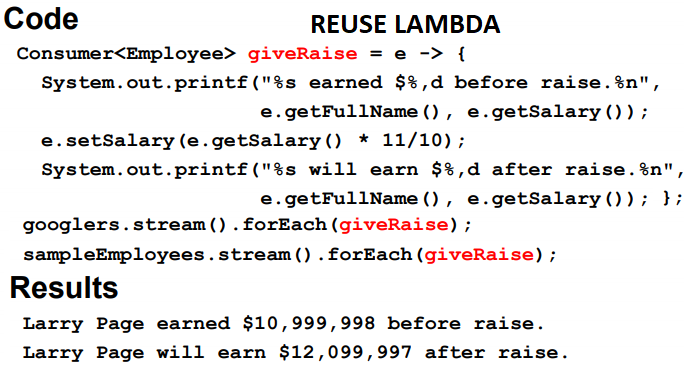




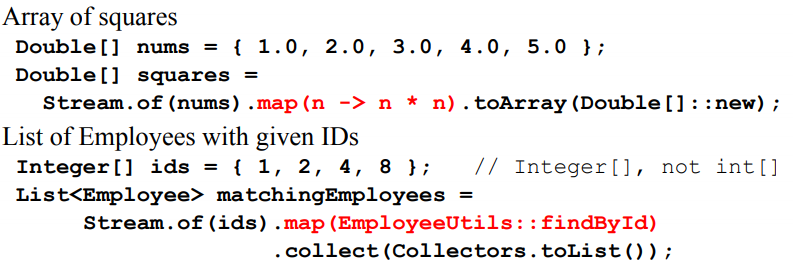
GooglerSt. store nothing. It’ not data structure. You can do only one chain of operation on Stream. **ForEach,** consumer. Illegal ops: loop twice, change local var., break/return early

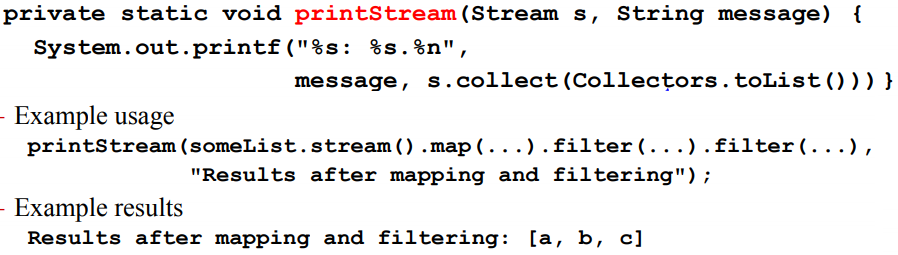




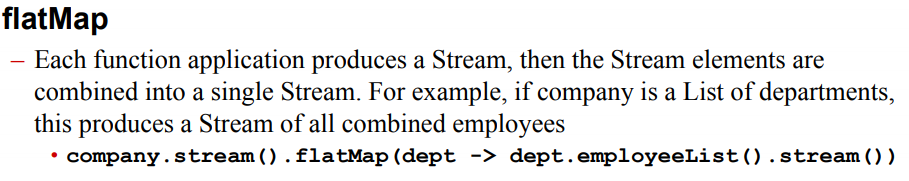


**MAP** Produces a new Stream

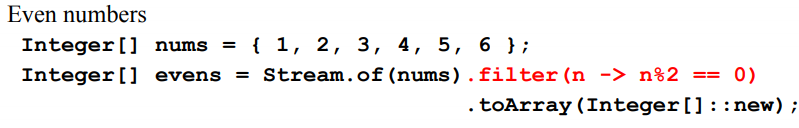




***mapToInt*** – Applies a func. that prod. an Integer, but the result Stream is an IntStream, not Stream<Integer>. Convenient because IntStream has no-arg. methods like sum, min, max.



**Filter**, predicate



**FindFirst**, returns 1 element. As soon as finds the 1 element, it stops. Returns Optional!

