Java-programming

Working with Java SE

Today

- We'll start with a recap of dates and interfaces
  - First at awtrainer git pull…
  - Then at your own git directory git push….

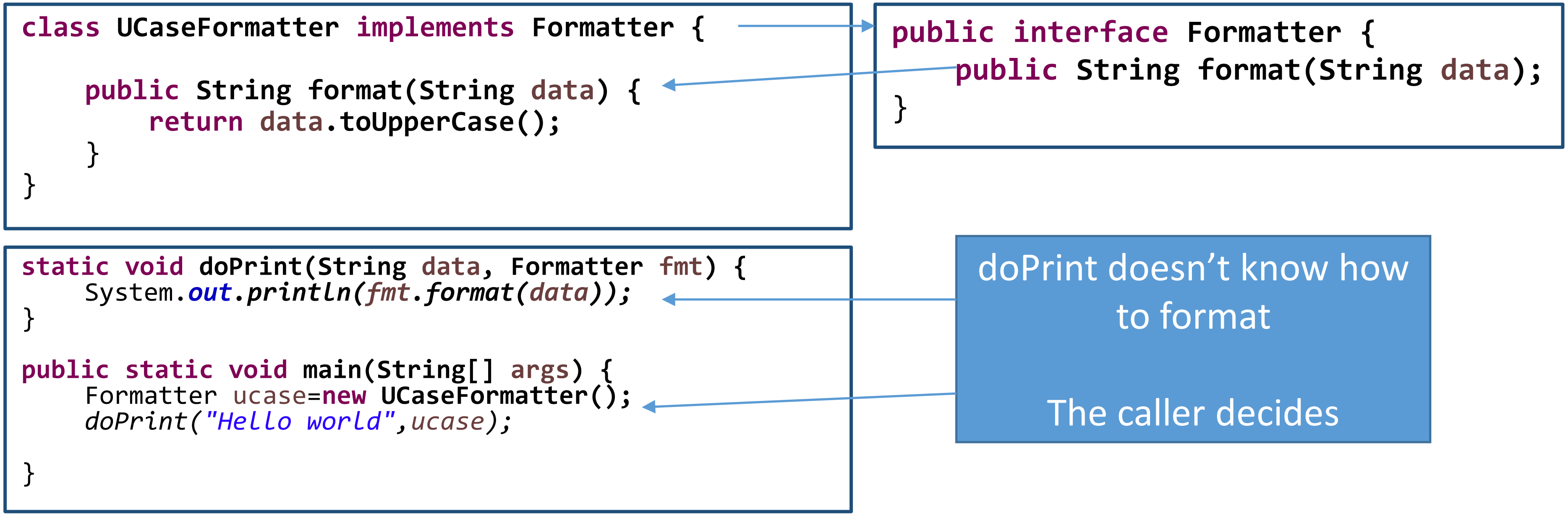
- You will learn to understand functional interfaces and how lambda expressions are used with those. Most importantly you will learn to work with different collection types

Interfaces

Inheritance: Interface

Up to JDK 1.7

- Interface just names methods that must be implemented elsewhere
  - Kind of completely abstract class, no method has an implementation
- Interfaces are not instantiated
  - No constructor
- Fields may be declared to an interface, but they are automatically “static final”
- Class promises to give implementation to the methods of an interface with “implements” -keyword



```java
class UCaseFormatter implements Formatter {

    public String format(String data) {
        return data.toUpperCase();
    }
}
```

```java
public interface Formatter {
    public String format(String data);
}
```

```java
static void doPrint(String data, Formatter fmt) {
    System.out.println(fmt.format(data));
}

public static void main(String[] args) {
    Formatter ucase=new UCaseFormatter();
    doPrint("Hello world",ucase);

}
```

doPrint doesn’t know how to format

The caller decides

Demo

- Clone an object
- Implement a very simple interface

Exercise

- You company uses two kinds of entities to perform work: people to whom salary is paid and companies that invoice their work. We are implementing financial application that can handle both

- In your Techniques-project create interfaces-package and the following interface and classes there
  - Interface Worker
    - void pay(amount)
  - Person implements Worker
    - Encapsulated name field
    - pay prints "[name] is payed [amount] EUR and [amount*0.25] as social security fees"
  - Company implements Worker
    - Encapsulated name field
    - pay prints "[name] invoices [amount]EUR + VAT [amount*0.24] EUR"

- In your "main"-class create finances(Worker w) –method (meaning the class that contains the main-method)
  - Calls w.pay(2000);

- And in the main
  - create Person and Company objects
  - Pass them to finances-method

Collections

Collections

- Java.util -package declares quite a few interfaces and classes for mainining collections of actual data items
  - All are generic types
- Basic collection types (interfaces)
  - List<Titem>
    - Most often used implementation is ArrayList
    - ”Dynamic array”: add/insert/delete/sort items
    - Indexed access
  - Set<Titem>
    - Most often used implementation is HashSet
    - Some similarities to list, but items must be unique
  - Queue<Titem>
    - FIFO or LIFO
  - Map<Tkey,Titem>
    - Most often used implementation is HashMap
    - You store key-value pairs
    - Typically you access items through their key

Both List and Set inherit Collection-interface

Collection, typical operations

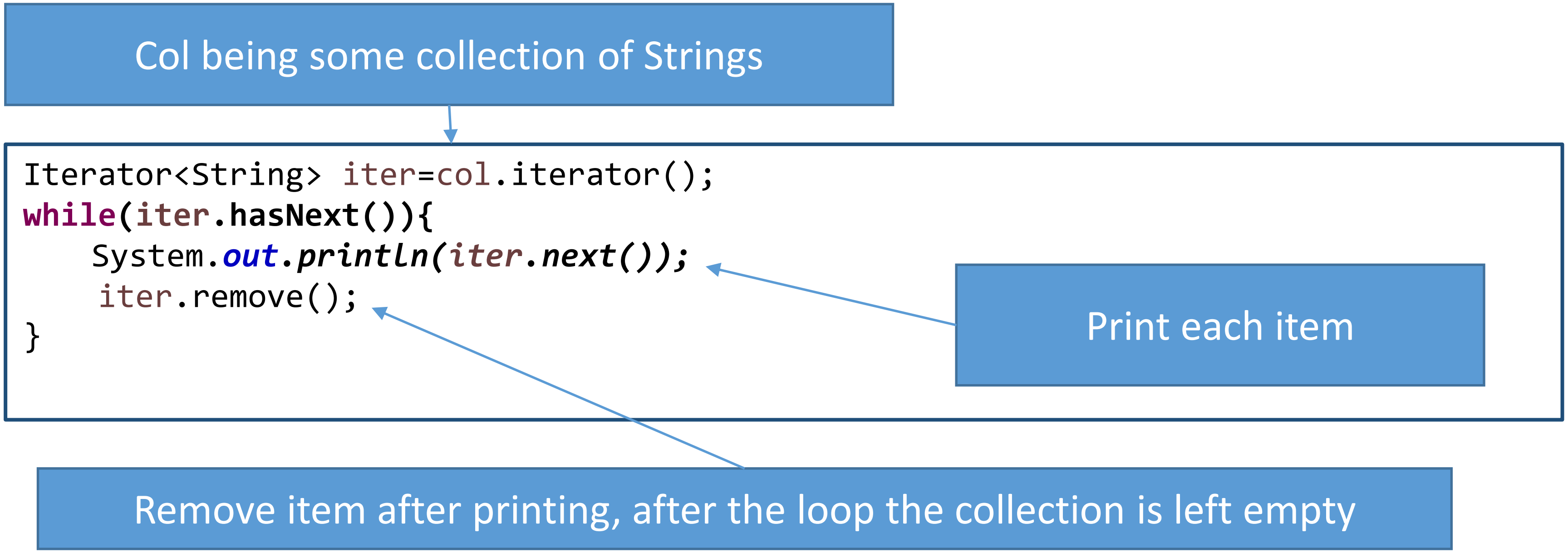
T being the generic type used in the collection

- boolean add(T item)
  - Return value indicates whether the item was actually added or not
- boolean remove(T item)
  - Was it removed or not
- boolean addAll(Collection<T> items)
- boolean removeAll(Collection<T> items);

- int size()
- boolean contains(T item)

Iterating the collection

- Collection provides iterator-method that returns the Iterator-object for collection
  - Iterator.hasNext(), Iterator.next()
- Removing items from the collection while looping though the collection is in most cases not allowed
  - With iterator you may accomplish that



Col being some collection of Strings

```
Iterator<String> iter=col.iterator();
while(iter.hasNext()){
    System.out.println(iter.next());
     iter.remove();
}
```

Print each item

Remove item after printing, after the loop the collection is left empty

List implementations

- ArrayList
  - Basic choice
  - Quick access with indexes
  - Inserts and removes to the middle of the list are (a little) slow
- Vector
  - Basically, a thread-safe ArrayList
- LinkedList
  - Slower indexed access, quicker inserts and removes to the middle

Set-implementations

- HashSet
    - Basic choice
    - Order of the items is not guaranteed
- TreeSet
    - Items remain in their natural order
- LinkedHashSet
    - Items remain in the order they are placed into the set

Exercise

- Create collections-package and CollectionsTest-class therein
  - Again create a separate functions to test following features

- Create a HashSet of Strings holding weekdays
  - Iterate the set to show the items
  - Try adding Monday twice
  - Change the set to TreeSet
    - And show contents

- Create a HashSet of Persons
  - Try adding a Person –object by the same name (not the same object) twice
  - ???

Queue-implementations

- LinkedList
  - In addition to List-interface also implements Queue-interface
  - With push and pop methods you have LIFO-queue (Stack)
- PriorityQueue
  - Maintains items in their natural order or the order determined by the Comparator passed in to the constructor
  - pop removes and returns the first item

Map implementations

- HashMap
  - Order of the items is not guaranteed
- TreeMap
  - Items are in natural order of their keys
- LinkedHashMap
  - Items are in order they are placed into the map

Exercise

- Create a HashMap<string,Person>,
  - The key should be “social security number” associated with the person object
  - Figure out different ways to loop through the HashMap

- Create a HashMap<Integer,Integer>
  - Build a loop of 1000 iterations
  - On each iteration generate a random number between 1-100
    - Use that number as a key to HashMap, the actual value should be counter: how many times on those 1000 iterations was this value generated.

Exercise

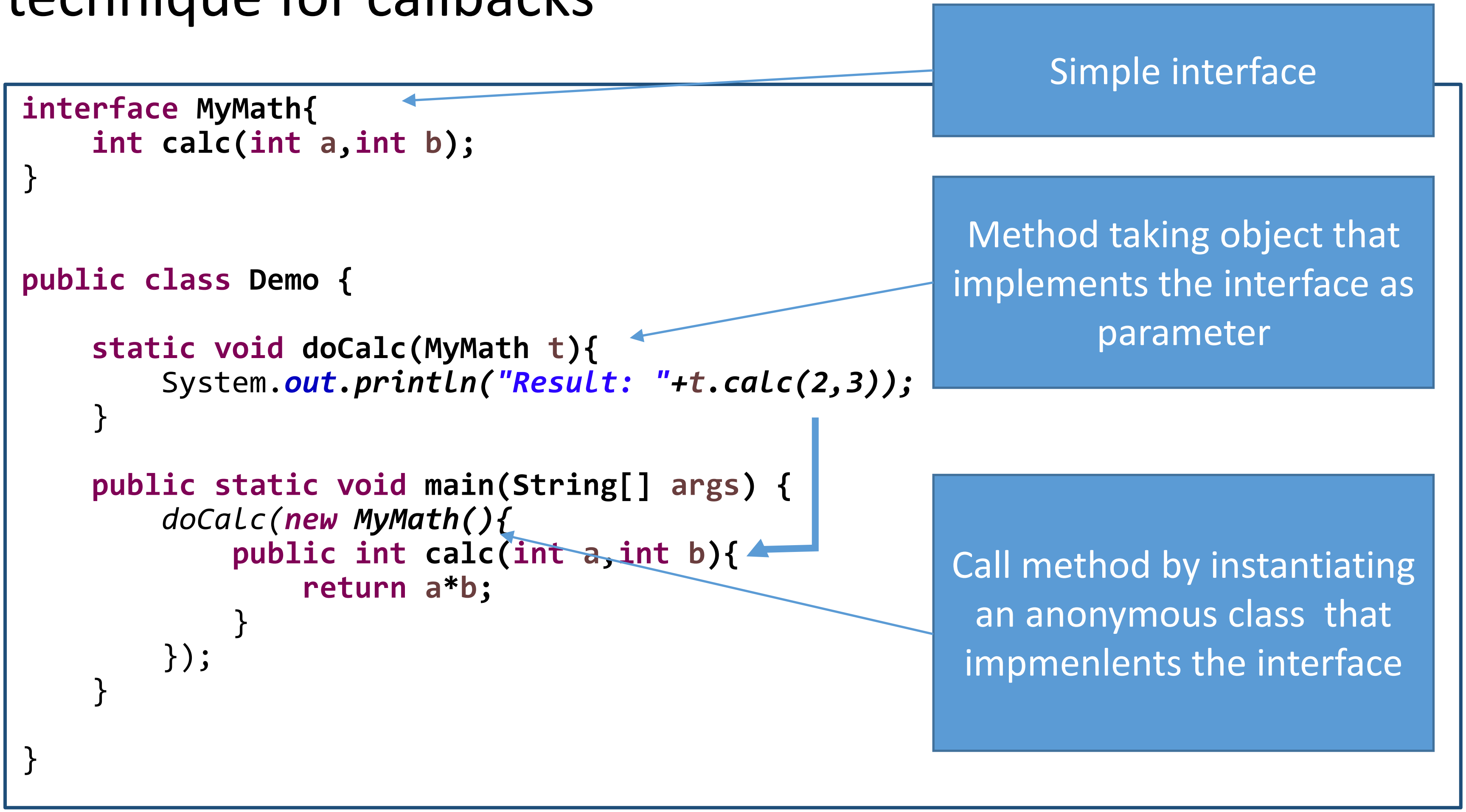
- Add encapsulated field int age to Person
- Create ArrayList<Person>
  - As static field to your CollectionsTests-class
  - Initialize it to contain several Person objects with various names and ages
  - Loop through the list and show name and age of each person.
    - You might want to implement toString-method to the Person-class

Lambdas

And functional interfaces

First there were anonymous classes

- Use of anonymous classes is the traditional java-technique for callbacks



```
interface MyMath{
    int calc(int a,int b);
}


public class Demo {

    static void doCalc(MyMath t){
        System.out.println("Result: "+t.calc(2,3));
    }

    public static void main(String[] args) {
        doCalc(new MyMath(){
            public int calc(int a,int b){
                return a*b;
            }
        });
    }

}
```

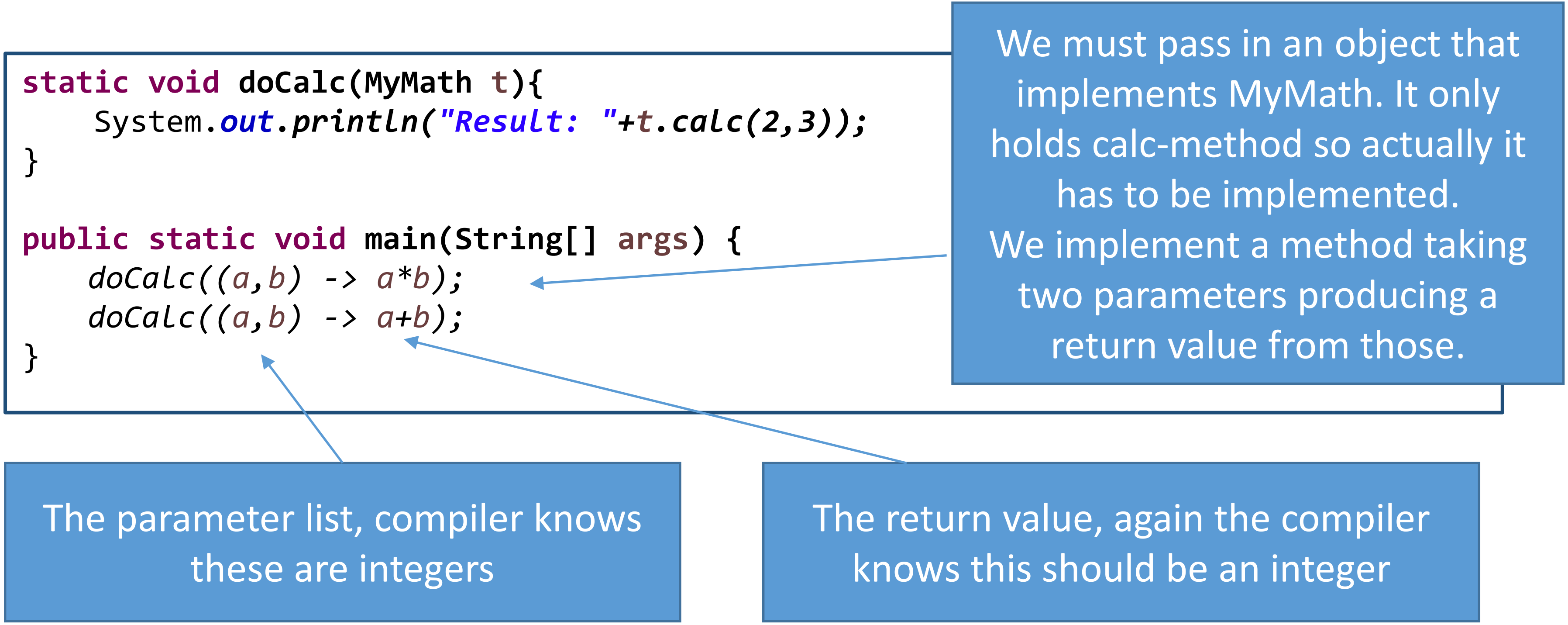
Simple interface

Method taking object that implements the interface as parameter

Call method by instantiating an anonymous class that impmenlents the interface

Then came lambda-expressions (Java8)

- Shorthand for implementing anonymous class against an interface with just one method
  - Actually compiler doesn’t generate the class file as it would in the sample on previous page



```
static void doCalc(MyMath t){
    System.out.println("Result: "+t.calc(2,3));
}

public static void main(String[] args) {
    doCalc((a,b) -> a*b);
    doCalc((a,b) -> a+b);
}
```

We must pass in an object that implements MyMath. It only holds calc-method so actually it has to be implemented. We implement a method taking two parameters producing a return value from those.

The parameter list, compiler knows these are integers

The return value, again the compiler knows this should be an integer

When declaring the lambda

- You are always implementing a method declared in some interface
- You declare the parameter list
  - Compiler will know the types of parameters from the method declaration in the interface
  - A single parameter doesn’t have to be surrounded by parenthesis
  - Parameter list of zero or two or more parameters must have the parenthesis
- And what is done with the parameters, actual method body
  - In most cases the body is just a single statement that can be evaluated into return value of the method.
  - If a block-is defined (multiple statements) the body also needs to contain actual return-statement
  - In the body you can use variables from “outer” scope but they must be final or effectively final

```
Runnable r = () -> System.out.println("Hello");
ActionListener l = e -> System.out.println(e.getActionCommand()) ;
Predicate<String> p = s -> !s.isEmpty();

MyMath c= (a,b) -> {
    if (a<=0) return b;
    if (b>100) return a;
    return a-b;
}
```

Exercise

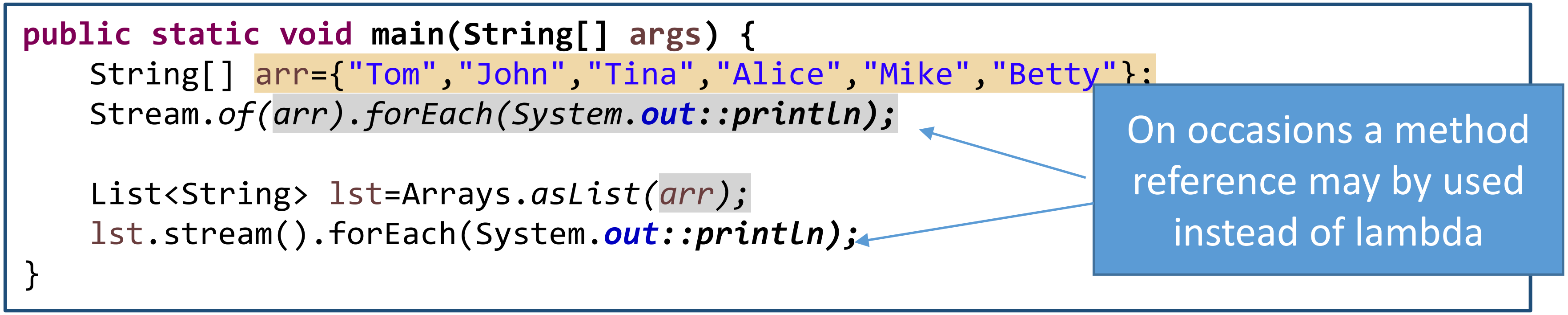
- Remember the Worker-interface, it only had one method
  - When testing finances, could you use lambda expression as a parameter?
- Earlier we created ArrayList<Person>
  - Sort it by name
  - Sort it by age
  - You need to study ArrayList-documentation and sort method there in

Exercise (extra)

- Implement Calculate-interface with total-method taking two double-parameters and returning a double value.
- Implement showVatPrice that takes net-price (double), vat (double) and Calculate-object as parameter. It uses Calculate.total to get the total price, then calculates vatAmount=total-net And displays all three values
- Call showVatPrice first with an instance of anonymous class, then using lambdas. Pass vat in different ways
  - Absolute euro amount
  - Percentage-value 24
  - Percentage-value 0.24
  - And provide correct algorigthm for total in each case
- Create sum-method to this class. Use method-reference to use it as algorithm for total.
- Add a default method to the interface, for example
  - double mult(int times, double a, double b) (times*(a+b))
- Test by:
  - Calculate c=(a,b) -> a+b;
  - System.out.println("Total "+c.times(4,2.1,3.2);

Using streams against collections

- Starting from Java8 you have been able to query the stream of values from a collection
- Removes the need of building a loop of your own to manipulate collections
    - Process each item somehow
    - Filter items based on criteria
    - Sort the items
- In most cases “streaming” provides the best performance



```
public static void main(String[] args) {
    String[] arr={"Tom","John","Tina","Alice","Mike","Betty"};
    Stream.of(arr).forEach(System.out::println);

    List<String> lst=Arrays.asList(arr);
    lst.stream().forEach(System.out::println);
}
```

On occasions a method reference may by used instead of lambda

About streams

- Streams are not data constructs, they operate on data of original collection
- Original collection is not modified, you cannot for example delete data from the collection during the stream operation
- No random access (indexes), items “flow” in stream and you access them sequentially
- Lazy, only amount of work absolutely needed is done
- Also support parallel operations, several threads operate on the collection
- Designed for lambdas…

Some functional interfaces

java.util.function-package declares some functional interfaces often encountered on stream-operations

- Though they are used on other occasions also

Function<T,R>	Take parameter of type T, return value of type R.
Predicate<T>	Take parameter of type T, return boolean. Does condition apply to parameter.
Consumer<T>	Operate on parameter of type T without returning a value
Supplier<T>	No parameter but returns a value of type T.
BinaryOperator<T>	Take two T's as parameter and return T

Some stream methods

In these examples lst is a List<String>

- Filtering, supply a Predicate

```
lst.stream()
    .filter(s -> s.startsWith("T"))
    .forEach(System.out::println);
```

- Sorting, supply a Comparator

```
lst.stream()
    .sorted((a,b) -> a.length() - b.length())
    .forEach(System.out::println);
```

- distinct - ensure each item is processed only once
  - Kind of make the list a set
- peek - somehow operate on item before final operations
  - Takes a consumer as a parameter

Exercise

- Continue working with ArrayList<Person>

- Using streams
  - Sort it by age and display items
  - Only display items whose age>18 (or any other value you choose)
  - Display items whose age>18 sorted by name

Mapping

- Mapping “converts” each item in a stream to a new item

```
lst.stream()
    .map(s -> s.length())
    .forEach(System.out::println);
```

- Reduce produces a single value from the values in the stream

```
int sumOfLens=lst.stream()
    .map(s -> s.length())
    .reduce(0,(a,b) -> a+b).intValue();
```

Exercise

- Still working with ArrayList<Person>
- Use map to create stream of strings (the names of Persons)
  - And only display those
- Find the oldest person-worker
  - Which stream-function to use???
  - Do not try use any of the functions on the next slide

Optional

- Some stream operations return an Optional-value
  - Sum, max, min,findFirst, findAny….
- Optional-object encapsulates the actual return value but also provides functionality for situations where the value doesn’t exist
  - opt.get() - get the actual return value
  - opt.orElse(defVal) - get the value if it exists, else defVal
  - opt.ifPresent(consumer) - pass the valube to the consumer

Exercise

- Find the age of the oldest Person
  - Now use map and max

Collecting stream data

- Stream-result may be collected into a new collection
  - Now we create a new data construct

Array	Stream.toArray
List	Stream.collect(Collectors.toList())
Set	Stream.collect(Collectors.toSet())
Map	Stream.collect(Collectors.toMap(….))
String	Stream.collect(Collectors.joining(…));

Specialized primitive streams

- Java.util.stream provides specialized streams for primitive types
  - IntStream
  - LongStream
  - DoubleStream
- These can be generated
  - Stream.mapToInt(…)
  - Stream.mapToLong(…)
  - Stream.mapToDouble(…)
- Especially IntStream and LongStream supply static methods for populating the stream

Exercise

- Study IntStream
  - Generate 1000 random integers (1-100) into a List of integers
  - And from that stream create a HashMap as we did earlier
    - Value (1-100) is the key to the data
    - The actual data behind the key is counter, how many times that value is generated

Exercise

- You should have a Company-class
    - Add field ArrayList<Person> employees (may be public)

- Create couple Company-objects and place couple Persons on their employees-list
- Place the Company-objects into an ArrayList
- Study Stream's flatMap-method
    - How do you print all the employees of both companies?

So today

- We have worked with several types of collections
  - List,Set,Map
- Learned how to use Lambda-expressions
- Learned how to use streams with Collections