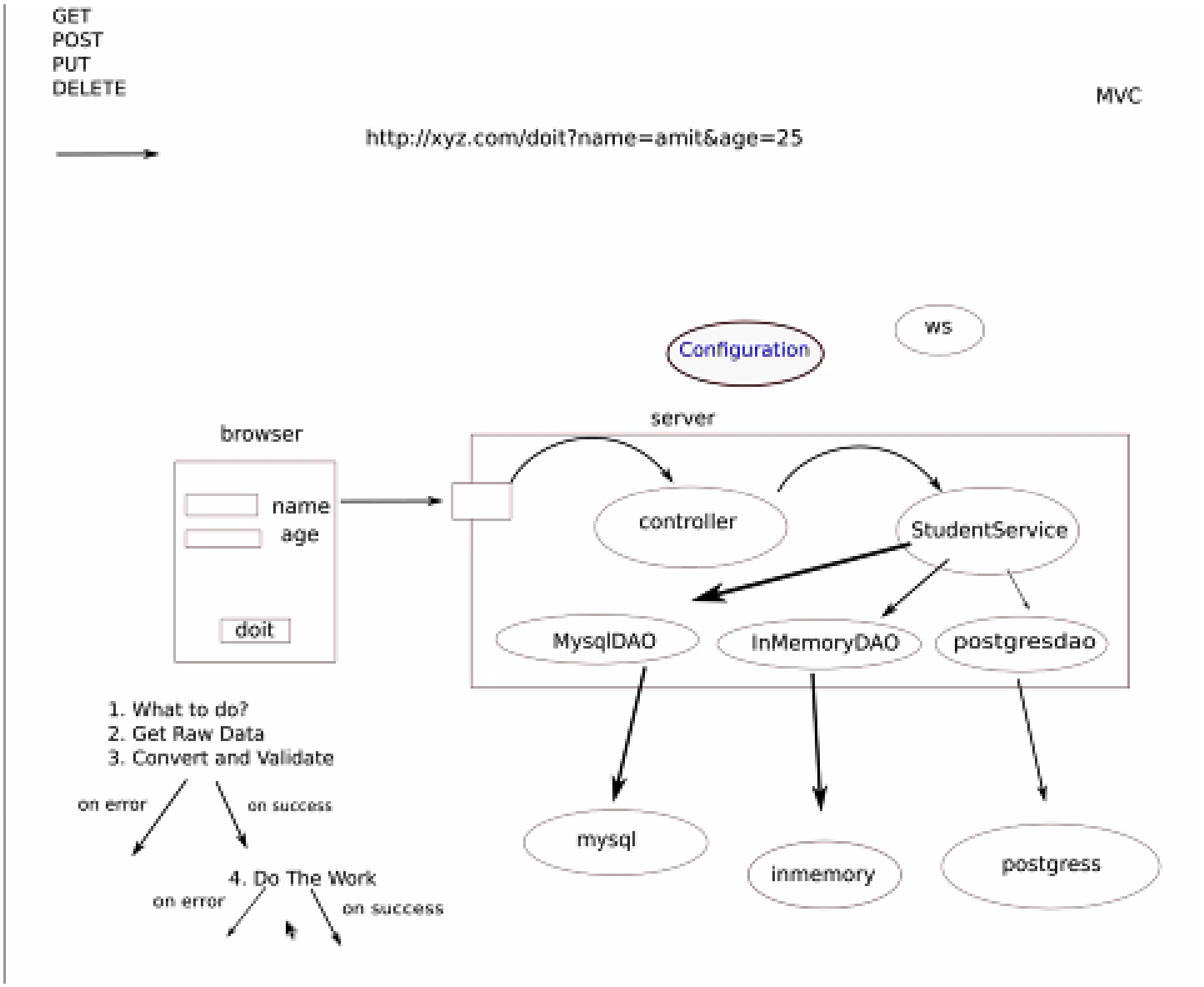
Gmail **Sanaz Arabzadeh Esfarjani <sanaz.azh@gmail.com>**

Java Intermediate notes

Sanaz Arabzadeh Esfarjani <sarabzadeh@salesforce.com> Wed, Apr 5, 2023 at 4:35 PM
To: sanaz.azh@gmail.com



Controller: the first component that is listening to the client request. The first point of entry

- Will be responsible for doing those 3 items:
  - What to do?
  - Get raw data
  - Convert and validate (on error, on success)

- **Separation of concern**: we don't want to do all the writing here. The service is where our business logic is (our app) : our service.

The request comes in: get me all info of a student with ID=10, Controller converts the ID to int and validate and sends it to the Service

Service: it has some business logic. It needs to get data from a db.

- We don't want to put db interaction code in the service
- **Dao:** to move db handling code out and put it in a handler object. In Spring Boot: **repository**
  - These **dao objects** have no idea about the business logic
  - They only can move data in and out of db (CRUD operations)

Git: https://github.com/anildi/IntJava

1. Start Intellij: **new -> Project from Version Control** use that git URL and the location of the project/

Maven saves artifactory: inside **.m2** in the home directory. Where we can find a repo and a dir called **repository:**

- **Settings.xml**: we tell Maven where to get stuff
- **repository/** we find all dependencies:
  - **org/hamcrest/hamcrest-all** has all versions of that library
  - **Hamcrest-all-1.3.jar** actual library

Create:

1. **Domain object** inside package.**domain**
2. **Dao** inside package.**dao**
3. **Service** inside package.**service**
4. **App** inside package.**app (this is called by our controller?)**
5. **src/main/resources** where we put configurations/resource
   a. Abc.properties (**we then read that in daoFactory and** using **bundle** and instantiate an impl of dao interface based on the value in properties)
6. **src/main/java** for source code

1. First, create a **domain object: Customer**
   All java classes should be **packages: some.some.som..**

   For 0 or more phone numbers:
   - List of phone numbers
   - Set of phone numbers
   - **Or just var arg like : String…**

   Interface Collection:
   - We cannot get one item, we get all items
   - More specific: **Set:**
     - No Ordering in location
     - No duplicate
     - Does not get the ith element
     - We can retrieve the first or ith one!
     - The order of elements that we return can change from one call to another
   - **List:**
     - Make it an ordered collection (location by i location)
     - Can access ith element
     - The order of all elements returned does not change from one call to another

   Constructors options that we have**:**
   1. **Use builder pattern**
   2. **Call one constructor from another**
   3. **Use init(....) for each constructor and** define **init(...) as the** function with setting all of those properties.

String… : we can only have one var arg constructor and it has to be the last one. For zero or more values. So the users can either specify or not specify it.

```
    public Customer(Long id, String name, LocalDate dataOfBirth, CustomerStatus status, String... phoneNumbers) {
                this.id = id;
                this.name = name;
                this.dataOfBirth = dataOfBirth;
                this.status = status;
                this.phoneNumbers = Set.of(phoneNumbers); // factory method in Set interface
}
```

OR

```
   public Customer(Long id, String name, LocalDate dataOfBirth, CustomerStatus status, String... phoneNumbers) {
```

```
        this.id = id;
        this.name = name;
        this.dataOfBirth = dataOfBirth;
        this.status = status;
//        this.phoneNumbers = Set.of(phoneNumbers); // factory method in Set interface
        this.phoneNumbers = new HashSet<>();
        for (String pn : phoneNumbers) {
            this.phoneNumbers.add(pn);
        }
    }
```

When using this. It has to be the **first thing inside the constructor.** or using Set:

```
    public Customer(Long id, String name, LocalDate dataOfBirth, CustomerStatus status, String... phoneNumbers) {
        this(id, name, dataOfBirth, status, Set.of(phoneNumbers));
//        this.id = id;
//        this.name = name;
//        this.dataOfBirth = dataOfBirth;
//        this.status = status;
////          this.phoneNumbers = Set.of(phoneNumbers); // factory method in Set interface
//        this.phoneNumbers = new HashSet<>();
//        for (String pn : phoneNumbers) {
//            this.phoneNumbers.add(pn);
//        }
    }
```

Zero-arg constructor:
for frameworks that make instances of this object model (using reflection) like Springboot, we need an empty constructor so the framework create an object (they want to see zero-arg constructors) and then populate it

```
public Customer() {} // for frameworks that make instances of this object model ( using reflection) like Springboot, we need an empty constructor so the framework create an object (they want to see zerp-arg constructors) and then populate it
```

Add toString() method to Domain Object for debugging and printing…

equal() method for comparing instances of that class (objects) -

Creating a DAO:

- We for now create a hashmap in DAO so we can store and retrieve data
- When inserting a new record, after successfully putting at the database it gets its ID and db often creates that. In general if the user asking you a create a new thing, the one piece of data that they don't have is the id of the newly created object. **So we can return the id of newly created object as the return type. Or return the newly created object with ID in it.**

In Java, we make copy of collections: we make a copy of reference. They both refer to the same object. If through the list, we change the customer object it is reflective and it changes the object. These copies of shallow copies.

How can we prevent the object from changing?

- Try to create immutable objects. For example, the Java **String** class is immutable. When we change a String, we make a new copy of the object.
- So when we change that object, we make a new copy of that object.
- So when **set methods are called, we can get rid of set methods => we have an immutable object**
-

Who is creating these service, dao, app, controller objects? (look at the diagram above**)**
When are they created?
How many of them are created?

- To make the HashMap stays around so it won't get created each time is:
    1. Create a service globally (instead of instantiating it for every request)
    2. Make the map **static: there is one map**

Strategy pattern

- We have different strategies for doing the same work, use **interface**
- **Program to an interface Not the implementation**

Factory pattern:

- We move out the decision making of which dao to instantiate out of **service** and into **DAO factory. Factory is not an application code but a configuration code.**
-

```
/ knows how to read properties file
ResourceBundle bundle = ResourceBundle.getBundle("sarab");
```

—-

To make a domain object like Customer sortable, comparable etc, we need to make that class implement **Comparator interface:**

1. **Implement Comparator interface (NOT needed since we're** using lambda to create define our own comparator:
   a. `Comparator<Customer> lambda1 = Comparator.comparing(Customer::getName);`
2. **In application class,** (after we instantiate the Service object):
   a. Create some persistent entries for customers : `init(app.service)` where **init** function create 5 customers using **service.insertCustoemr()** method
3. In **main of application class calls** method: **sortLambda()**
4. Create **sortLambda()** method in application class that uses **lambda for comparison (**`Collections.sort(customers, lambda1)`**):**

```
private void sortLambda() {

    List<Customer> customers = service.getAllCustomers();

    Comparator<Customer> lambda1 = Comparator.comparing(Customer::getName);

    Collections.sort(customers, lambda1);

    System.out.println(customers);

    for (Customer c : customers) {
        System.out.println(c);
    }
}
```

Sort: **Sorts the specified list into ascending order, according to the natural ordering of its elements.**

Signature 1:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Sort takes list of type **T, (**you can pass in sort method, a list of some type T such that T extends Comparable<T>), means either that type **T** implements that interface **Comparable** (or if it is class, extends that)

Signature 2:

```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

Sort takes a list of type **T** and a Comparator **of type T**

So for **Student** domain object to be of type T : we need to implement **Comparable:**

Public class Student implements **Comparable<String>** {....} and implement **compareTo** method from that interface.

To return an immutable vs mutable list:

```
public List<Student> getAll() {
//        return List.copyOf(students.values());
        return new ArrayList<>(students.values()); // to return a mutable list
    }
```

Different Strategies (an Interface implementations) - BUT Lambda is a better approach for implementing an interface

```
class App {
   NameComparator nc = new NameComparator();
   DOBComparator dc = new DOBComparator();
   List<String> students = new ArrayList<>();

   Collections.sort(students, nc);
   Collections.sort(students, dc);
}

// In application code
// Each of these are different strategies
class NameComparator implements Comparator<Student> {

   @Override
   public int compare(Student o1, Student o2) {
       return o1.getName().compareTo(o2.getName());
   }
}

class DOBComparator implements Comparator<Student> {

   @Override
   public int compare(Student o1, Student o2) {
       return o1.getDob().compareTo(o2.getDob());
   }
}
```

Lambda

Lambda

```
class Scratch {
   public static void main(String[] args) {
       Function<Integer, String> dd = integer -> integer.toString();

       Function<Integer, String> d2 = integer -> myMethod(integer);
       Function<Integer, String> d2 = Scratch::myMethod;
       d2.apply(1);
   }

   public static String myMethod (Integer i) {
       return i.toString();
   }
}


List<Student> students = service.getAllStudents();

Collections.sort(students, (o1, o2) -> o1.getName().compareTo(o2.getName()));
Comparator<Student> lambda4 = Comparator.comparing(Student::getName);

// for one statement no need for return and {}
Comparator<Student> lambda3 = (o1, o2) -> o1.getName().compareTo(o2.getName());

// Compiler figures out the type of o1 and o2: no need to pass in the types
Comparator<Student> lambda2 = (o1, o2) -> {
   return o1.getName().compareTo(o2.getName());
};

Comparator<Student> lambda1 = (Student o1, Student o2) -> {
   return o1.getName().compareTo(o2.getName());
};
```

Method references

ClassName::Action

java.util.function

Some Interfaces that will become your friends

- **Predicate<T> -** perform some test on the argument of type T and return true or false
- **Function<T, R> -** take in an argument of type T, perform an operation on it, and return some (possible different) type R
- **Consumer<T> -** take in an argument of type T and return void. i.e., consume the argument
- **Supplier<T> -** take no arguments and return something of type T

- There are also two argument variations of some of the above.
- And interfaces dealing with primitive types, to avoid the expense of auto boxing.

Function

Let's say you have a list of Employee objects and you want to convert each employee object to a string that includes their name and salary. You can use the Function interface to define a function that takes an Employee object as input and returns a formatted string.

```
public class Employee {
   private String name;
   private double salary;

   public Employee(String name, double salary) {
       this.name = name;
       this.salary = salary;
   }

   public String getName() {
       return name;
   }

   public double getSalary() {
       return salary;
   }
}

public class Main {
   public static void main(String[] args) {
       List<Employee> employees = new ArrayList<>();
       employees.add(new Employee("Alice", 50000));
       employees.add(new Employee("Bob", 60000));
       employees.add(new Employee("Charlie", 70000));

      // One way defining Function:
      Function<Employee, String> employeeToString = employee -> employee.getName() + " earns " +
employee.getSalary();

       List<String> employeeStrings = new ArrayList<>();
       for (Employee employee : employees) {
           String employeeString = employeeToString.apply(employee);
           employeeStrings.add(employeeString);
       }

      // Other way defining Consumer:
      Consumer<Employee> printName = employee -> System.out.println(employee.getName());
      employees.stream().forEach(printName);

       System.out.println(employeeStrings);
   }
}
```

Filter

- To create a filter for a list of our **Customer,** use **Predicate interface**

```
public static void main(String[] args) {
   SortingApp app = new SortingApp();
   init(app.service);

   app.sortLambda();
   System.out.println("------------");
   app.sortLambdaDOB();
   System.out.println("-----filtered------");
   app.findByFilterCustomers();
 }


private void findByFilterCustomers() {
   List<Customer> customers = service.getAllCustomers();

   List<Customer> filteredCustomers = findBy(
           customers,
           c -> c.getDataOfBirth().getYear() > 1990);

   for (Customer c : filteredCustomers){
       System.out.println(c);
   }
}

private <T>List<T> findBy(List<T> list, Predicate<T> predicate) {
   List<T> filteredList = new ArrayList<T>();

   for (T item : list) {
       if (predicate.test(item)) {
           filteredList.add(item);
       }
   }
   return filteredList;
}
```

Map:

```
List<String> mappedCustomers = customers.stream()
        .filter(customer -> customer.getStatus() == Customer.CustomerStatus.Privileged)
        .map(Customer::getName).collect(Collectors.toList());
```

Peak to peak elements:

```
List<Long> mappedCustomers = customers.stream()
       .peek(s -> System.out.println("In peak with customer: " + s))
       .filter(c -> c.getStatus() == Customer.CustomerStatus.Normal)
       .peek(s -> System.out.println("In peak with after filter: " + s))
       .map(c -> c.getDataOfBirth().until(LocalDate.now(), ChronoUnit.YEARS)) // to calculate age based on
LocalDate DOB
       .collect(Collectors.toList());
```

To calculate average age of customers with status == Restricted

```
private void findAveCustomerAgeByStatus() {
   List<Customer> customers = service.getAllCustomers();

   OptionalDouble averageAge = customers.stream()
           .filter(c -> c.getStatus() == Customer.CustomerStatus.Restricted)
           .mapToInt(c -> Math.toIntExact(c.getDataOfBirth().until(LocalDate.now(), ChronoUnit.YEARS)))
           .peek(c -> System.out.println(c))
           .average();

   System.out.println(averageAge);
}
```

FlatMap

- **Second for loop… when the result of map (which wraps a stream around it) has list or set and we need to iterate over those elements as well.**
- Given a list of set of phone numbers, we want to return a list of phone numbers **[[....], [....], [.....]] Set<phone number>**
- you can use Java Streams to flatten the sets and collect the items into a list.
    - We pass in **stream to flatMap()** and
    - flatMap will **stream that:**

```
/**
* 8. Write a method to return all the phone numbers of all customers. Make sure your test data includes at least
some customers with phone numbers.
*/
private void findPhoneNumbers() {
   List<Customer> customers = service.getAllCustomers();

   List<String> phoneNumbers = customers.stream()
           .peek(s -> System.out.println("peek 1: " + s))
           .flatMap(s -> s.getPhoneNumbers().stream())
           .peek(s -> System.out.println("peeking: "+ s))
           .collect(Collectors.toList());

   System.out.println(phoneNumbers); // List<String>
}

// Or returns a string that are joined by , instead of returning a list of strings
private void findPhoneNumbers() {
   List<Customer> customers = service.getAllCustomers();

   var phoneNumbers = customers.stream()
           .flatMap(s -> s.getPhoneNumbers().stream())
           .collect(Collectors.joining(", "));

   System.out.println(phoneNumbers); // String
}
```

peek 1: Customer{id=3, name='Tuesday', dataOfBirth=1990-08-12, status=Restricted, phoneNumbers=[234 3453 5456]}
peeking: 234 3453 5456
peek 1: Customer{id=4, name='Blue', dataOfBirth=1991-09-24, status=Normal, phoneNumbers=[]}
peek 1: Customer{id=5, name='Kia', dataOfBirth=1988-12-26, status=Restricted, phoneNumbers=[3456 4567 45674, 234 3453 5456]}
peeking: 3456 4567 45674
peeking: 234 3453 5456
[3245 3458 34566, 234 3453 5456, 234 3453 5456, 234 3453 5456, 3456 4567 45674, 234 3453 5456]

forEach

- It takes a Consumer as an argument and calls it's accept method for each element in the Iterable.
- So a forEach method is now available for all Iterable classes.

```
default void forEach(Consumer<? super T> action) {
   Objects.requireNonNull(action);
   for (T t : this) {
       action.accept(t);
   }
}
//call forEach
List<String> ls = ...
ls.forEach(s → System.out.println(s))
```

Map returns a stream of object but we can't do math on stream of object so we need to use the following instead:
Map from stream of objects to stream of primitive use:
.mapToLong()
.mapToInt()
…

OperationalDouble

```
/**
* 7. Write a method to calculate the average age of Customers who have the status
* Restricted.
*/
private void findAveCustomerAgeByStatus() {
   List<Customer> customers = service.getAllCustomers();

   customers.stream()
            .filter(c -> c.getStatus() == Customer.CustomerStatus.Restricted)
            .mapToLong(c -> c.getDataOfBirth().until(LocalDate.now(), ChronoUnit.YEARS))
            .peek(c -> System.out.println(c))
            .average().ifPresent(System.out::println);
}
```

`OptionalDouble` is a class in Java that represents an optional double value. It is a container object that may or may not contain a double value. If a value is present, the `OptionalDouble` will contain the value, and if a value is not present, the `OptionalDouble` will be empty.

The `OptionalDouble` class is part of the `java.util` package and was introduced in Java 8. It is often used in situations where a method may or may not return a value, and the caller needs to know whether the value is present or not.

DoubleConsumer: which a consumer where the value is of type **double.** It has bunch of methods for double values.

Find the first phone number for each customer if exists:

```
/**
* 9. Write a method to return only the first phone number, if any, for all customers. For your test data, make
sure that some of your customers have multiple phone numbers, and at least one customer has no phone numbers.
*/
private void findFirstPhoneNumbers() {
   List<Customer> customers = service.getAllCustomers();

   var result = customers.stream()
            .map(s -> s.getPhoneNumbers().stream().findFirst()) // Stream<Optional<String>>
            .filter(Optional::isPresent)
            .map(Optional::get)    // Stream<String>
            .collect(Collectors.toList());

   var result2 = customers.stream()
            .flatMap(s -> s.getPhoneNumbers().stream().findFirst().stream())
            .collect(Collectors.toList());

   System.out.println(result);
   System.out.println(result2);
}
```

Generic

You see when we create a new reference to our lint object, we can add string to a list of integers and it errors out not in compilation but runtime error:
When we create a `List badList = lint;` we create a list that contains **anything!**
We need to use generics **with type to avoid such issues.**
Warning: Raw use of parameterized class lint

```
public class GenericDemo {

   public static void main(String[] args) {
       GenericDemo gd = new GenericDemo();
       gd.whatIsAlList();
   }
```

```
    public void whatIsAlList() {

        List<Integer> lint = new ArrayList<>();
        lint.add(10);

//        lint.add("asf");

        List badList = lint; // create a new reference to lint list object
        badList.add("bad input"); // we could add a string to a list of integer!
        // the place that gets me an issue:
        // class java.lang.String cannot be cast to class java.lang.Integer
        for (Integer i : lint) {
            System.out.println("i: " + i);
        }
    }
}
```

Generics is strictly compile time…….and eventual the bytecode will be just (all types will be erased):
//Type Erasure
List lint = new ArrayList<>();

In java we can overload the methods (same name, different signature) but if we define two methods like this:

```
public void fun(List<Integer> i) {

}

public void fun(List<String> i) {

}
```

Compiler errors out : **'fun(List<Integer>)' clashes with 'fun(List<String>)'; both methods have same erasure**
Which points to the same **erasure** issue that the bytecode won't have those types (types will be erased) because both of those **List** will be just **List** without those generic types

```
/**
* Class Hierarchy                                    Collection
*
* Object                                             List<Object> ==> Anything that is an Object, i.e.
everything
*
* Number                                             List<Number>  ==> Anything that is a number
*
* Integer, Double, AtomicInteger                     List<Integer> ==> Only Integers
*
*/
```

If we had **sum(List<Number> input) {...}** we would get an error when trying to use it for **lInt** list. To be able to use the **sum** method we need to use **generic wildcard <? Extends Number>**

You just want to read from it: use it as a producer of your values: (extends)

```
public void genericDemo() {

    List<Number> lNumber = new ArrayList<>();
    lNumber.add(10);
    lNumber.add(23.34534);
    lNumber.add(20);

    double result = sum(lNumber);
    System.out.println("result : " + result);

    List<Integer> lInt = new ArrayList<>();
    lInt.add(10);
    lInt.add(20);
```

```
    sum(lInt);
}

public double sum(List<? extends Number> input) {
    double sum = 0;
    for (Number number : input) {
        sum += number.doubleValue();
    }

    return sum;
}
```

Another use case: when we want to put things into it:
We have a method that takes a list of integers and an array of integers and add that arr to that list. It works for a list of integers but not with a list of Numbers even though a list of Numbers can have integers.
We should use **generic wildcard ? super - consumer**

```
// ------------ Consumer
Integer[] iarr = {3, 4, 888};
add(lInt, iarr);
System.out.println("lInt: " + lInt);

add(lNumber, iarr);

public void add(List<? super Integer> input, Integer[] arr) {
    for (Integer i : arr) {
        input.add(i);
    }
}
```

Generic wildcards are a way to specify that a method or class should accept any type of object, but still restrict some types of operations that can be performed on that object.

The wildcard character (`?`) is used to denote a generic wildcard. There are three types of wildcard: `? extends`, `? super`, and `?`.

Here are some examples of using generic wildcards in Java:

1. `List<?>` **- This declares a List that can hold any type of object. For example:**

```
List<?> myList = new ArrayList<String>();
List<?> myOtherList = new ArrayList<Integer>();
```

2. `List<? extends Number>` **- This declares a List that can hold any type of object that extends Number. For example:**

```
List<? extends Number> myList = new ArrayList<Integer>();
List<? extends Number> myOtherList = new ArrayList<Double>();
```

3. `List<? super Integer>` **- This declares a List that can hold any type of object that is a super class of Integer. For example:**

```
List<? super Integer> myList = new ArrayList<Number>();
List<? super Integer> myOtherList = new ArrayList<Object>();
```

Producer **extends,** Consumer **Super**

- **If the collection is a producer of values: you want to take things out of it. Everything taken out of it is at least at that type**
- **If the collection is a consumer of values: you want to put things into it. Give me a list of anything in which I can put integers (for example)**

We can understand what the methods do by looking at their signature and types:

```
/*
      we can see the source/producer and the destination/consumer
*/
static <T> void copy(List<? super T> des, List<? extends T> src) {}

/*
   Collection of anything
   In this method, you cannot put anything in or out - we don't know what the type id
   The only thing we can assume is that they are Objects

   It works here because in Java we use .equals() method to check equality which takes Object
   Here we don't care about the type of object and we just count the frequency of it in a collection of anything!
*/

static int frequency(Collection<?> c, Object o) {

   int freq = 0;
   for (Object obj: coll){
       if (obj.equals(o)) {
           freq ++;
       }
   }
   return freq;
}

}
```

Generic types (2 types)

```
class Pair<T, U>{
   public Pair(T first, U second) {
       this.first = first;
       this.second = second;
   }
   public T first;
   public U second;
}

Pair<String, Integer> psi = new Pair<>("Boo", 4);

String s = psi.first;
int y = psi.second;
```

Exceptions

- Check exception

When you acquire IO resource like open file etc **it is very important to give that resource back and close it: This pattern:**

If you open the resource, you should make sure it gets close before you exit. The following is not working because if there is an error the file won't be closed!

We can add **finally** after catch but **fis**

```
   /**
    * read a byte from a file and return
```

```
     */
    public void readFile(String fileName) {

        try {
            FileInputStream fis = new FileInputStream("pom.xml");
            int c = fis.read();

            System.out.println("c: " + c);

            fis.close();

        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```

It gets into a nasty structure like:

```
    /**
     * read a byte from a file and return
     */
    public void readFile(String fileName) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("pom.xml");
            int c = fis.read();

            System.out.println("c: " + c);

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                fis.close();
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }

    }

}
```

Instead use:

```
/**
* you open all of your resources in that () in try and compiler will make sure that it will close your resource
* not matter
*
*/
public void readFileWithTryWithResources(String fileName) {
    try (FileInputStream fis = new FileInputStream("pom.xml");) { // open all of you resources here

        int c = fis.read();
        System.out.println(c);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Use **AutoCloseable:** https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/AutoCloseable.html

For example we have a class that instances of it being used and we want to make sure that it is closed all instances after being used by devs **like a file.**

- We use **AutoCloseable** class for it
- We instantiate the class inside **try()** so the compiler can close it properly : a `try`-with-resources block for which the object has been declared in the resource specification header

```
/**
* people will use instances of this class and it is imporatnt to shut it down properly
*/
class MyClass implements AutoCloseable{

   int i = 0;

   public MyClass() {
       // some setup
   }
   @Override
   public void close() {
       // Must be closed by programmer. Do important shutdown
   }
}

public class ExceptionDemo {

   public static void main(String[] args) {
       ExceptionDemo ed = new ExceptionDemo();
       ed.readFile("pom.xml");
   }

   /**
    * you open all of your resources in that () in try and compiler will make sure that it will close your
resource
    * not matter
    *
    */
   public void readFileWithTryWithResources(String fileName) {
       try (MyClass mc = new MyClass();
            FileInputStream fis = new FileInputStream(fileName);) { // open all of you resources here

           int c = fis.read();
           System.out.println(c);

       } catch (IOException e) {
           e.printStackTrace();
       }
   }

}
}
```

Exception with Lambda

We need to catch exceptions in streams. **You're not allowed to not catch exceptions inside lambda. We have to catch exceptions inside a lambda.**

How can we deal with an exception in a lambda without aborting and continuing processing the rest.

```
package sarabzadeh.app;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class ExceptionLambdas {

    public static void main(String[] args) {
        ExceptionLambdas el = new ExceptionLambdas();
        el.process();
    }

    public void process() {
        List<String> fileName = List.of("pom.xml", "doesnotexist");
        try {
            List<String> result = readFirstLinesWithoutStream(fileName);
            for (String line : result) {
                System.out.println("line: " + line);
            }
        } catch (IOException e) {
            System.out.println("Exception in process: " + e);
            e.printStackTrace();
        }

    }


    public List<String> readFirstLinesWithoutStream(List<String> fileNames) throws IOException {
        List<String> result = new ArrayList<>();

        for (String fileName : fileNames) {
            try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
                String line = br.readLine();

                result.add(line);
            }
        }
        return result;
    }


    /**
     * Scenario: going through some files, some may fail and some may not,
     * so if we want to continue
     *
     */
    public List<String> readFirstLinesWithStream(List<String> fileNames) {
        List<String> result = new ArrayList<>();

        fileNames.stream()
                .map(fileName -> {
                    try (BufferedReader br = new BufferedReader(new FileReader(fileName));) {
                        String line = br.readLine();
                        return line;
                    } catch (IOException e) {
                        throw  new RuntimeException(e);
                    }
                })
                .collect(Collectors.toList());

        return result;
```

```
    }

    /**
     * to not blow out: we use flatmap to continue and also catch errors
     * when we use map we HAVE TO return something

     */
    /**
     * to not blow out: we use flatmap to continue and also catch errors
     * when we use map we HAVE TO return something, in case of error, an empty stream
     *
     * flatMap, we give it a stream, it will stream out and if we give it an empty stream, it will throw
it away!
     * when we use a map: we have to return somethin: we throw an exception it blows out everythiung
     * with flatMap: we return a stream and we take advantage of flatmap throws away an empty stream
when we error and so
     * the flatMap goes up for the next item (it has throwned out the empty stream)

     */
    public List<String> readFirstLinesWithStreamFlatMap(List<String> fileNames) {
        List<String> result = fileNames.stream()
                .flatMap(fileName -> {
                    try (BufferedReader br = new BufferedReader(new FileReader(fileName));) {
                        String line = br.readLine();
                        return Stream.of(line);
                    } catch (IOException e) {
                        return Stream.empty();
                    }
                })
                .collect(Collectors.toList());

        return result;
    }

    /**
     * Instead of stream , we reutn an optional
     * flatMap was doing some filtering stuff for us.
     * With Optional we need to first check if there is something present and
     * then get the stream of string from stream of optional string

     */
    public List<String> readFirstLinesWithStreamTheOptional(List<String> fileNames) {


        var result = fileNames.stream()
                .map(fileName -> {
                    try (BufferedReader br = new BufferedReader(new FileReader(fileName));) {
                        String line = br.readLine();
                        return Optional.of(line);
                    } catch (IOException e) {
                        return Optional.<String>empty();
                    }
                })
                .filter(Optional::isPresent)
                .map(Optional::get)
                .collect(Collectors.toList());

        return result;
    }

    /**
```

```
     * What if we want to go through the list and get items and if there is an error, I also get the
error
     * It has to be a list of either string or an exception (different use case)
     * we could sorft of do it with Optional when we return a result without any value (we can use some
libraries to do it)
     */
    public void callReadFirstLIneWithWrapper(List<String> fileName) {
        List<Wrapper<String>> result = readFirstLinesWithWrapper(fileName);

        result.forEach( w-> {
            if (w.value != null) {
                String line = w.value;
            } else {
                Exception e = w.exception;
            }
        });
    }

    public List<Wrapper<String>> readFirstLinesWithWrapper(List<String> fileNames) {

        var result = fileNames.stream()
                .map(fileName -> {
                    try (BufferedReader br = new BufferedReader(new FileReader(fileName));) {
                        String line = br.readLine();
                        return Wrapper.ofValue(line);
                    } catch (IOException e) {
                        return Wrapper.ofError(e);
                    }
                })
                .collect(Collectors.toList());

        return result;
    }

}

/**
* What if we want to go through the list and get items and if there is an error, I also get the error
* It has to be a list of either string or an exception (different use case)
* we could sorft of do it with Optional when we return a result without any value (we can use some
libraries to do it)
*/

class Wrapper<T>{ // This T is for instances and cannot work for static methods for that we need to declare T

    public T value;
    public Exception exception;

    private Wrapper(T value, Exception exception) { // we make constructor private but instead provide static
methods below
        this.value = value;
        this.exception = exception;
    }

    public static <T> Wrapper <T> ofValue(T value) {
        return new Wrapper<T>(value, null);

    }
    public static <X> Wrapper <X> ofError(Exception e) {
        return new Wrapper<X>(null, e);

    }
```

```
}
```

Reflection

JVM has to go to the class path and find the byte code (.Class file) and loads it and runs bunch of test and creates an object class for that class.

The first thing we have to do is when we're using reflect API is to get class object for the class we're interested in:

1. Class.forName() it might go and load the class if it's not loaded yet. (load the **byte code .class file == load the class)**

When running JUnit, it uses reflection to get a reflection of our classes that we test.

Annotation

Java has 3 phases/states:

1. **Source code :** src
2. **ByteCode** .class file
3. **Loaded into JVM**

Retention: in which of theses 3 phases is this annotation going to be available:

- **SOURCE (**used only by compiler at compile time like **Override**)
- **CLASS(**annotation to be recorded in the class file (bytecode) **but need not be retrieved by the VM at runtime - default behavior)**
  - Writing tools to manipulate bytecode
- **RUNTIME(**Annotations are to be recorded in the class file by the compiler and retained by the VM at run time, so they may be read reflectively.**)**

You have to write your validation for your custom-defined annotations.

We can also inject out annotations as a dependency injection to our class