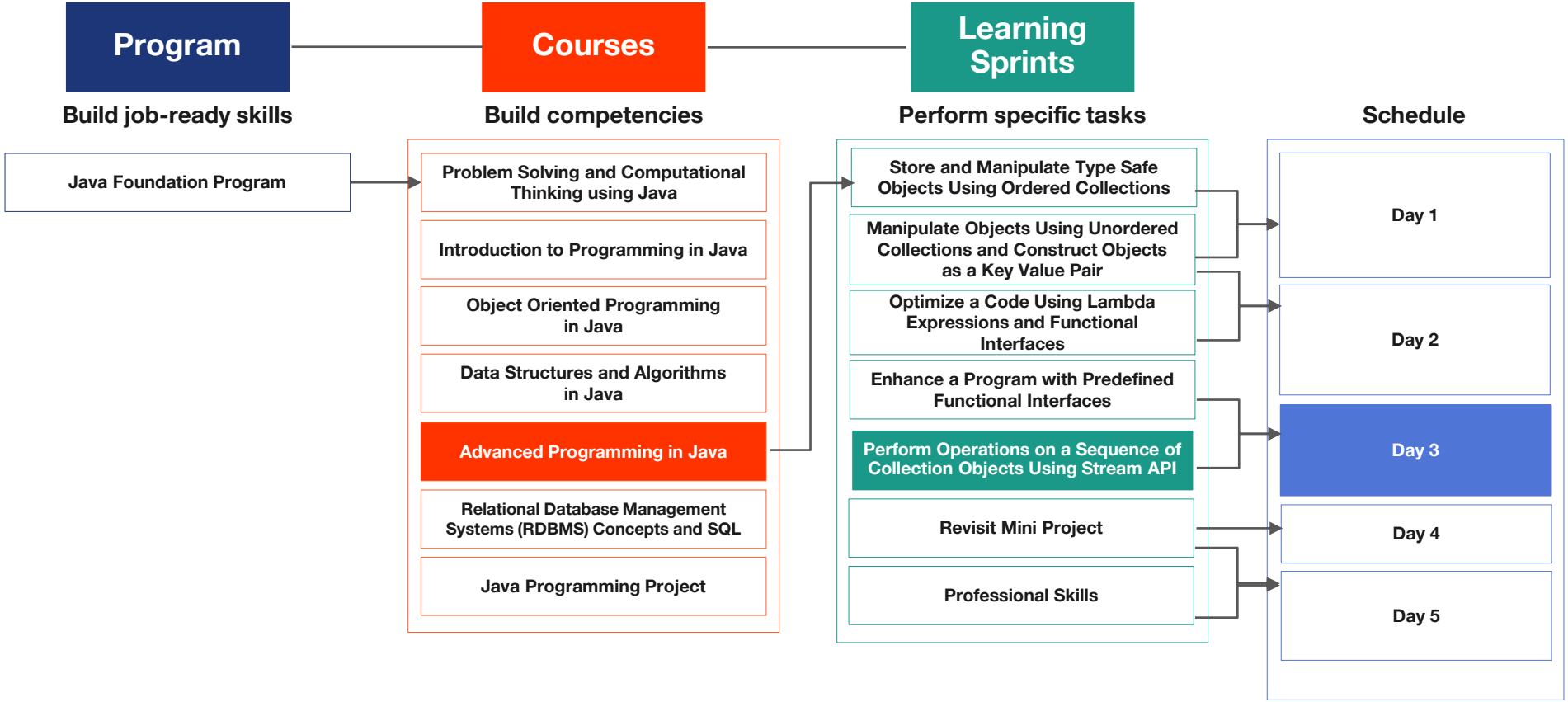
Java Program: Course 5: Plan



Transaction_date	Price	Payment_	Name	City	State	Account_Created	US Zip
01-02-2009 04:53	1200	Visa	Betina	Parkville	MO	01-02-2009 04:42	64152
01-02-2009 13:08	1201	Mastercar	Federica e An	Astoria	OR	01-01-2009 16:21	97103
01-04-2009 12:56	1202	Visa	Gerd W	Cahaba He	AL	11/15/08 15:47	35243
01-04-2009 13:19	1203	Visa	LAURENCE	Mickleton	NJ	9/24/08 15:19	8056
01-04-2009 20:11	1204		Fleur	Peoria	IL	01-03-2009 09:38	61601
01-02-2009 20:09	1205	Mastercar	adam	Martin	TN	01-02-2009 17:43	
01-05-2009 02:42	1206	Diners	Stacy	New York	NY	01-05-2009 02:23	10002
01-02-2009 09:16	1207	Mastercar	Sean	Shavano P	TX		78230
01-05-2009 10:08	1208	Visa	Georgia	Eagle	ID	11-11-2008 15:53	83616
01-02-2009 14:18	1209	Visa	Richard	Riverside		12-09-2008 12:07	8075
01-02-2009 07:35	1210	Diners	Hani	Salt Lake C	UT	12/30/08 5:44	84111
01-06-2009 07:18	1211	Visa	asuman	Chula Vista	CA	01-06-2009 07:07	91910
01-01-2009 02:24	1212	Visa	Lisa	Sugar Land	TX	01-01-2009 01:56	77478

Think and Tell

The sales and analysis department of a grocery chain spread across the US has the sales data of its stores available in a raw format. The department needs to analyze the data to perform various tasks.

According to you, what difficulties are likely to arise while dealing with a raw dataset?

Sales Analysis

The sales and analysis department needs to:

- Find all the customers from a given city
- Find the customers who have made purchases above $1,200
- List customers who had used visa cards for payments
- Change all customer names to uppercase, if they are not in the proper format

How can we perform these tasks using lambdas and functional interfaces?

Sales Analysis - Using Lambdas

1. Read the customer data from the file and then populate it to customer object
2. Add the objects to a list
3. Implement a predicate to filter the data
4. Use the Function<T,R> interface to ensure the names are changed to upper case
5. Perform all the enlisted tasks

```
private List<Customer>
filter(List<Customer> customerList,
Predicate<Customer> pre)
    {
     List<Customer> clist = new
         ArrayList<>();
     for(Customer c: customerList)
     {
            if(pre.test(c))
            {
                clist.add(c);
            }
        }
        return clist;
    }
```

Let Us Discuss

- Can we make the code more concise?
- What can we do if we encounter null values while reading the raw customer data?
- Can we ensure that the filter() method does not return a null list?
- How do we handle the null pointer exception situation?

Perform Operations on a Sequence of Collection Objects Using Stream API

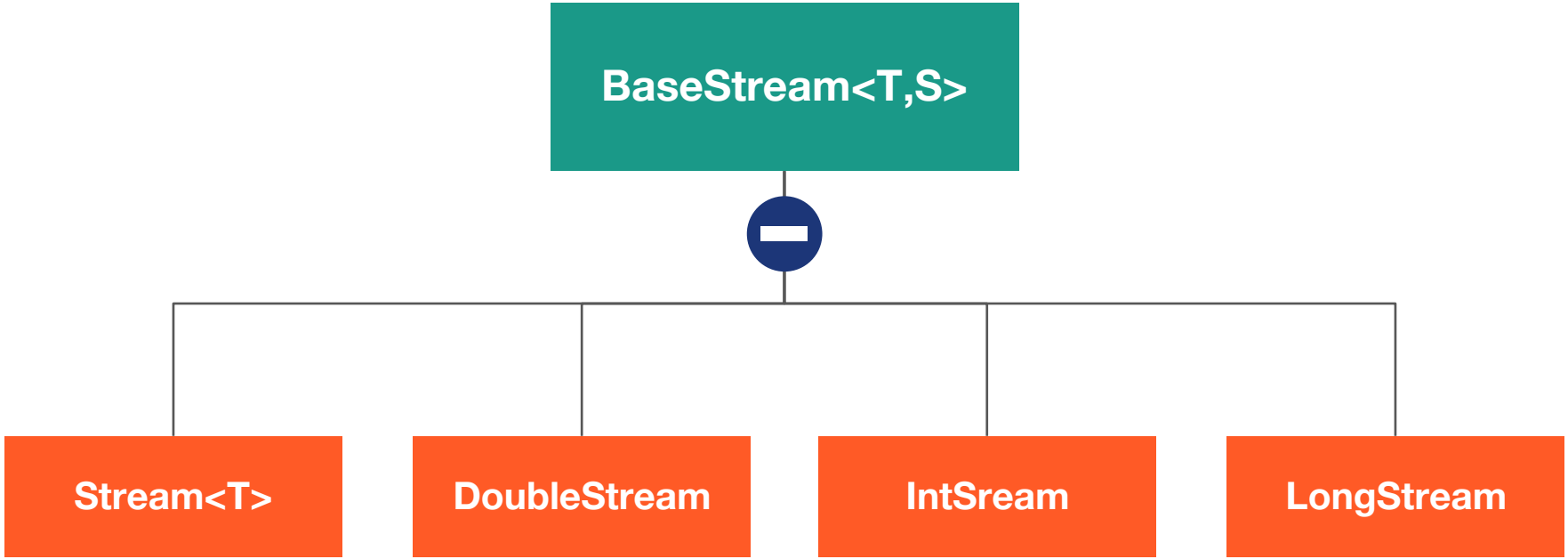
Learning Objectives

- Explain streams and its functions
- Implement stream API
- Use intermediate and terminal operations
- Apply methods of the optional class

Java8 - Stream API

- Stream API is used for processing a sequence of collection objects
- It is a series of objects that consist of methods to produce filtered results
- It is not a part of any collection; it gets insight from various data sources or mediums, such as, arrays and collections
- Stream does not alter the actual state of the array or collection; it only supplies the result according to the conveyed methods
- The need for external iteration of a stream is not necessary, as a stream has its own internal iteration techniques; the developer need not externally write a for loop or an iterator

The java.util.Stream Package



Streams – How Do They Work?

- Stream takes a source such as a collection or an array and works on it
- It performs aggregate operations like
  - filtering a stream based on a condition
  - finding certain elements based on a clause
  - finding the maximum and minimum elements in the stream, etc.
- These operations are called ***intermediate operations*** and they return a stream
- Most of the stream operations return a stream itself which can be pipelined; the pipelining is called ***terminal operations***
- Stream operations do iterations internally over the source element, unlike collections which need an explicit iteration

Steps to Work With Streams

- Convert a collection such as an ArrayList into a stream
- Perform operations on the stream
- Finally, accumulate the elements of the stream into another collection

Intermediate Operation

- **`filter:`**
  - This method is used for selecting elements according to the predicate passed as an argument
  - The syntax of `filter()` is as below
    ```
    Stream<T> filter(Predicate<? super T> predicate)
    ```
  - The `filter()` method lets you filter a stream
- **`map:`**
  - This method returns a stream consisting of the results of applying the given function to the elements of the same stream
  - The syntax of the `map()` method:
    ```
    <R> Stream<R> map(Function<? super T,? extends R> mapper)
    ```

Intermediate Operation (contd.)

- **`sorted:`**
  - The `sorted()` method is used for sorting the stream in a natural order or by using a comparator
  - The syntax for `sorted()` is as below
    - `Stream<T> sorted()`: Returns a stream consisting of the elements of the same stream, sorted according to the natural order
    - `Stream<T> sorted(Comparator<? super T> comparator)`: Returns a stream consisting of the elements of the same stream, sorted according to the provided Comparator

Examples on Intermediate Operations

```
Arrays.asList("Reflection","Streams","String");

Stream<String> stream = names.stream().filter(s->s.startsWith("S"));
```

- A list of string elements is passed as a stream to the filter method
- The filter method applies a condition using a predicate stating that only those elements that start with S should be in the preceding stream object

```
List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6, 8);

Stream<Integer> sortedList = list.stream().sorted();
```

- It returns a stream consisting of the elements of the integer stream in a sorted order

Quick Check!

What is the output of the below code?

```
Arrays.stream(new int[]{11, 23, 3, 45, 5}).filter(p->p>10);
```

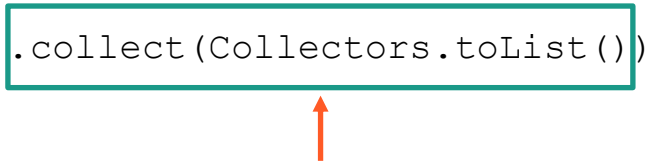
1. 11, 23, 45
2. 3
3. An integer stream

Terminal Operations

- **`collect`**: The `collect()` method of the stream interface is used to accumulate elements of the stream into a collection
- **`forEach:`** This method is used for iterating through every element of the stream
- **`reduce:`** This operation lets you compute a result by using all the elements available in a stream

Examples on Terminal Operations

```
List<String> names = Arrays.asList("Reflection","Streams","String");
List<String> result =
names.stream().filter(s>s.startsWith("S")).collect(Collectors.toList());
result.forEach(i->System.out.println(i));
```



- Stream methods can be chained one after another
- The filter() method produces a stream that is further collected into a list using the collect() method which ends in a terminal operation

Quick Check!

Guess the output of the below code.

```
int minimum = Arrays.stream(new int[]{1,2,3,4,5}).min().getAsInt();
System.out.println(minimum);
```

1. 0

2. 1

3. Compilation error - getAsInt() cannot be used

Quick Check!

Fill in the blank with the correct word.

Stream _____ is the concept of chaining operations together.

1. linking
2. pipelining
3. chaining

Sales Analysis

The sales and analysis department of a grocery chain spread across the US has the sales data of its stores available in a raw format. The department needs to perform the following tasks.

1. Find all the customers from a given city
2. Find the customers who have made purchases above $1,200
3. List the customers who had used visa cards for payments
4. Change all customer names to uppercase, if they are not in the proper format

Write a program to achieve the same using the Java 8 Stream API.

Optional Class

- The optional classes were introduced in Java 8 to counter NullPointerException that most users face
- This exception is thrown when we try to access an object that holds a null value
- The Optional classes provide methods to check if the value is present or if it is null

Methods of the Optional Class

- Below are a few methods present in the optional class

Method	Description
`public static <T> Optional<T> empty()`	It returns an empty Optional object. No value is present for this Optional.
`public static <T> Optional<T> of(T value)`	It returns an Optional with the specified non-null value.
`public static <T> Optional<T> ofNullable(T value)`	It returns an Optional describing the specified value, if it is non-null, otherwise it returns an empty Optional.
`public T get()`	It returns the value, if it is present in this Optional, otherwise it throws NoSuchElementException.
`public boolean isPresent()`	It returns true if there is a value present, otherwise false.
`public T orElse(T other)`	It returns the value if present, otherwise returns other.
`public T orElseGet(Supplier<? extends T> other)`	It returns the value if present, otherwise invokes other and returns the result of that invocation.

Using the Methods of the Optional Class

- To verify if a method does not throw null values, perform the following:

Wrap the object with optional

```
Optional<Address> op = Optional.ofNullable(voter.getAddress());
```

Use the method of optional class to check if the method returns a value and not null

```
if(op.isPresent()) {
     System.out.println("If address is null print voter "+voter);
}
```

Using the Methods of the Optional Class (contd.)

A method can also return an Optional

```
public Optional<Voter> findVotersByZipCode(List<Voter> voters,int zipcode)
    {
        Optional<Voter> voter = voters.stream().filter
         (p->p.getAddress().getPincode() == zipcode).findFirst();
        return voter;
    }
```

- It retrieves the voters of a locality
- The `findFirst` method returns an `Optional object`

Quick Check!

Which one of the following methods can be used to check null on an optional variable in Java 8?

1. `isPresent()`

2. `isNullable()`

3. `isPresentable()`

4. `isNotNull()`

Quick Check!

Fill in the blank with the correct word.

Optional has a special _____ value instead of wrapped null.

1. Optional.of()
2. Optional.empty()
3. Optional.isPresent()

Usage of Optional Classes

Write a program to determine the voters who are eligible for voting from a given voter's list. Eliminate the voters who do not have a proper address.

- Display all invalid voters who have their address given as null
- Replace the voters whose address is shown as null with a dummy value and mark them as invalid voters
- Throw an exception if the voter name is null
- Find voters in a specific locality using the zip code

Key Takeaway

- Stream API in Java 8
- Intermediate operations
- Terminal operations
- Optional class