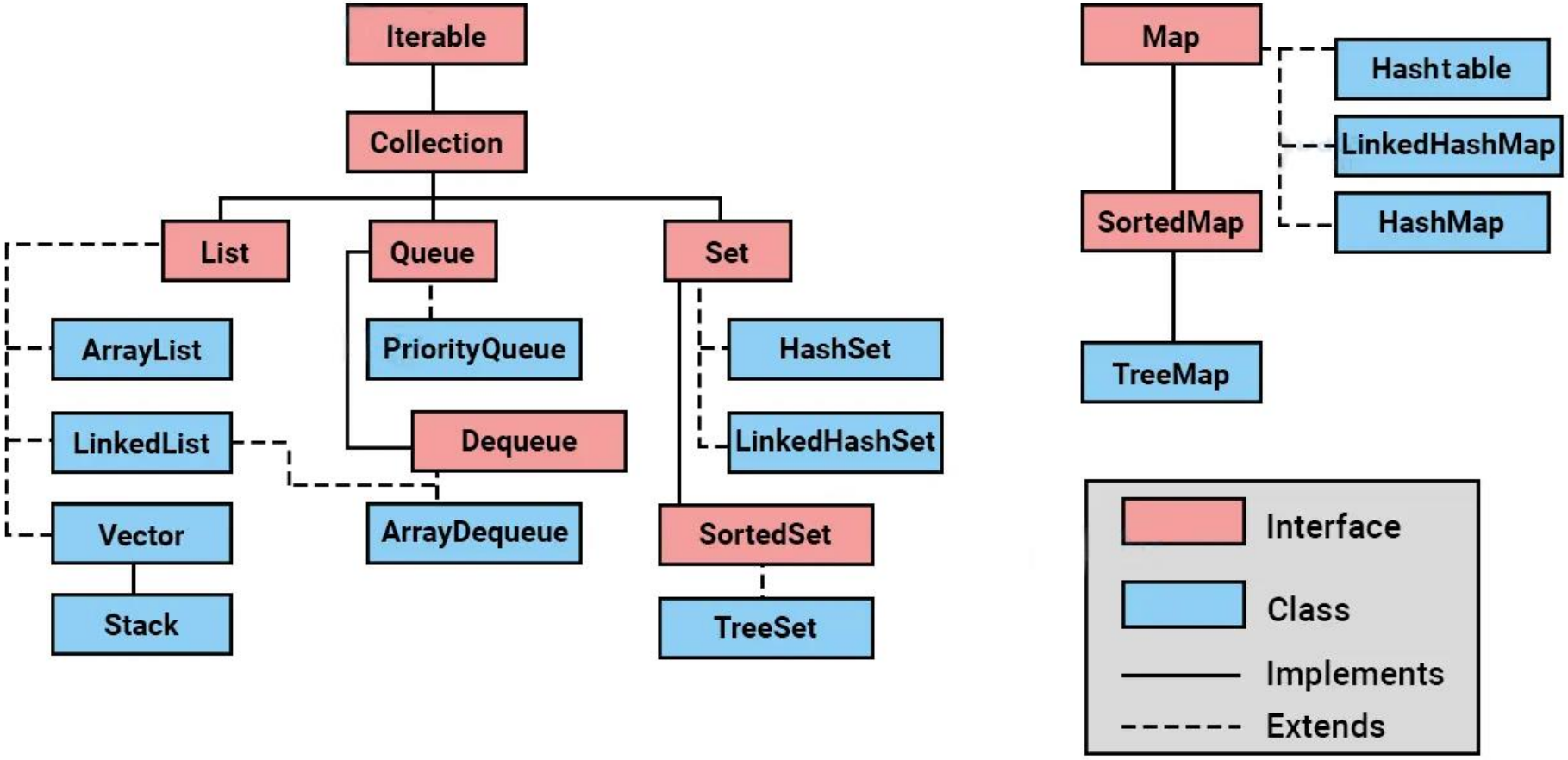
# Collections

- Collections
- Lambda expressions
- Streams

# Overview

- Collection is a framework in the `java.util` package that provides a unified architecture to store and manipulate a group of objects



# Explanation

- `Iterable`: something that one can move along in one direction, and can the target of the for-each loop
  - `Collection`: represents a group of objects, known as its elements
    - `Set`: a collection that cannot contain duplicate elements
    - `List`: an ordered collection that may contain duplicate elements
    - `Queue`: a collection used to hold multiple elements waiting for processing
- `Map`: an object that maps keys to values, cannot contain duplicate keys, and each key can map to at most one value

# `Iterable<T>` Interface

- This interface contains only a few abstract methods:

| Method | Description |
|---|---|
| `Iterator iterator()` | Returns an iterator |
| `void forEach(Consumer<? super T> action)` | Performs the given action for each element of the iterable |

- An `Iterator` is an object that allows to traverse through a collection and to remove elements from the collection selectively, if desired

# `Collection<E>` Interface

| Method | Description |
|---|---|
| `boolean add(E e)` | Inserts an element in this collection |
| `boolean addAll(Collection<? extends E> c)` | Inserts the specified collection elements in the invoking collection |
| `boolean remove(Object e)` | Deletes an element from the collection |
| `boolean removeAll(Collection<?> c)` | Deletes all the elements of the specified collection from the invoking collection |
| `boolean removeIf(Predicate<? super E> filter)` | Deletes all the elements of the collection that satisfy the specified predicate |
| `void clear()` | Removes the total number of elements from the collection |
| `int size()` | Returns the total number of elements in the collection |
| `boolean isEmpty()` | Checks if collection is empty |
| `Object[] toArray()`<br>`<T> T[] toArray(T[] a)` | Converts collection into array |

# Lists

# `List<E>` Interface

| Method | Description |
|---|---|
| `void sort(Comparator<? super E> c)` | Sorts the elements of the list on the basis of specified comparator |
| `int indexOf(Object o)`<br>`int lastIndexOf(Object o)` | Returns the index in this list of the first/last occurrence of the specified element, or -1 if the list does not contain this element |
| `E get(int index)` | Returns the element at the specified position in this list |
| `E set(int index, E element)` | Replaces the specified element in the list, present at the specified position |
| `List<E> subList(int fromIndex, int toIndex)` | Fetches all the elements lies within the given range |
| `ListIterator<E> listIterator([int start])` | Returns a list iterator over the elements in this list optionally starting at the specified position |

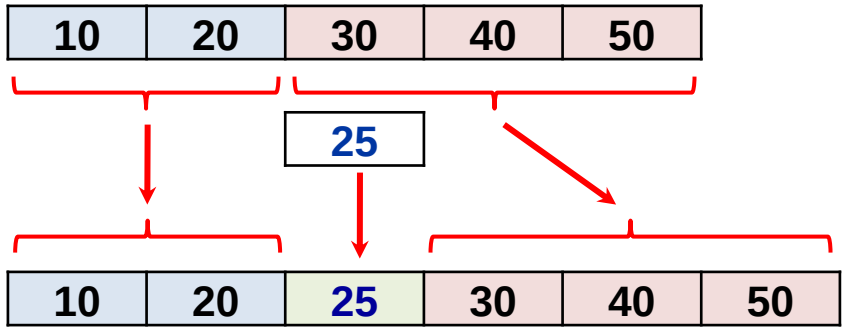
- `ListIterator` (which extends `Iterator`) allows traverse the list in either direction, while `Iterator` only allows to go in forward direction

# List Types

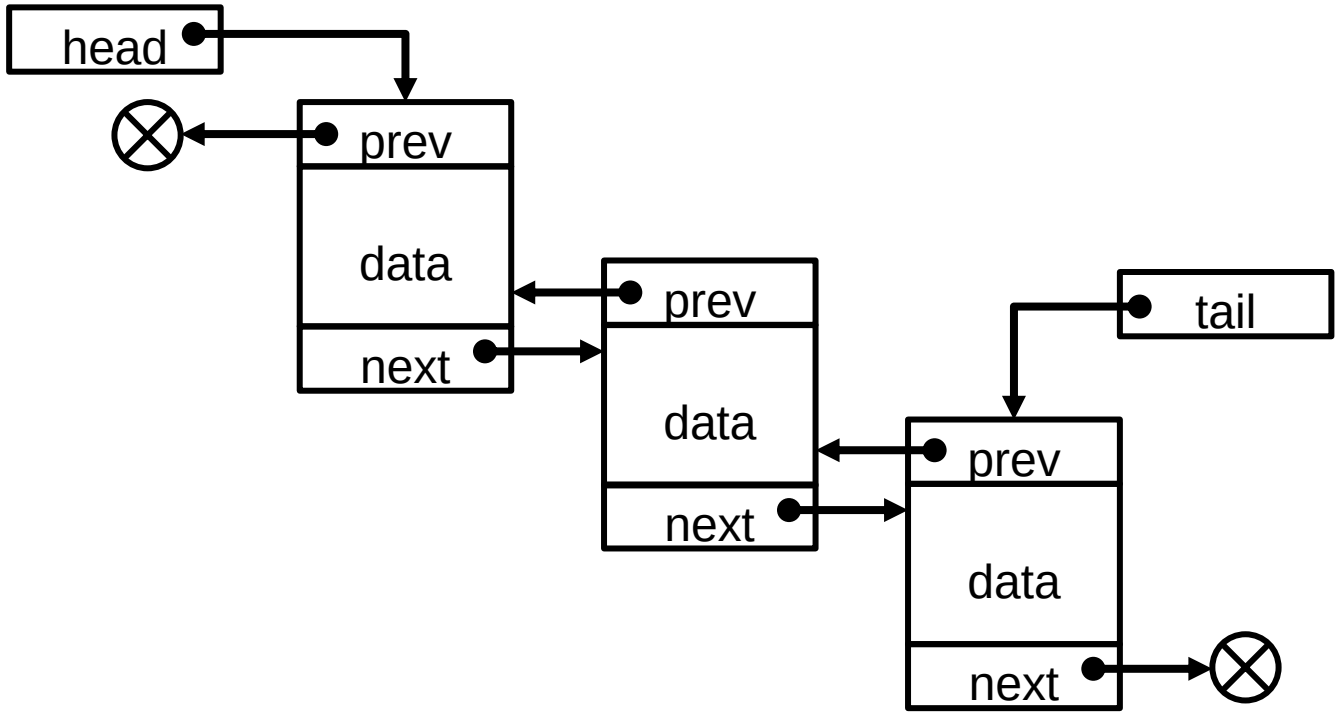
- `ArrayList`: uses dynamic arrays to store its elements and maintains insertion order
  - Random access property: fast at finding elements by their index, but slow at adding and removing elements
- `Vector`: is legacy and similar to `ArrayList`, but is synchronized (thread safe), while the other is not
- `LinkedList`: uses linked list data structure as it's internal implementation to store elements
  - Sequential access property: slow at finding elements by their index, but fast at adding and removing elements

# Random vs Sequential Access

- Random access




- Sequential access



# Example: Create and Iterate a List

```
// create a list of strings
List<String> list = new ArrayList<String>();

// add elements in the list
list.add("Mango");
list.add("Apple");
list.add("Banana");
list.add("Grapes");

// iterate the list element using for-each loop
for (String fruit : list)
    System.out.println(fruit);
```

# Example: Convert Array to List

```
// create an array
String[] array = { "Java", "Python", "PHP", "C++" };

// convert the array to list
List<String> list = new ArrayList<String>();
for (String lang : array)
    list.add(lang);

// print both
System.out.println("Array: " + Arrays.toString(array));
System.out.println("List: " + list);
```

# Maps

# Introduction

- Maps are usually used to implement:
  - Lookup tables
  - Dictionaries
  - Associative key-value pairs
- Types:
  - `HashMap`: non-synchronized, allowing one null key and values can be null
  - `LinkedHashMap`: similar to `HashMap` but preserve the order of insertion
  - `Hashtable`: synchronized, not allowing null key or value, faster than `HashMap`
  - `TreeMap`: no null key, but allowing null values, keys are ordered

# `Map<K,V>` Interface

| Method | Description |
|---|---|
| `void clear()` | Removes all of the mappings |
| `boolean containsKey(Object key)`<br>`boolean containsValue(Object value)` | Returns true if this map contains a mapping for the specified key/value |
| `void forEach(BiConsumer<? super K,? super V> action)` | Performs the given action for each entry in this map until all entries have been processed |
| `V get(Object key)`<br>`V getOrDefault(Object key, V defaultValue)` | Returns the value to which the specified key is mapped, or `null`/`defaultValue` if this map contains no mapping for the key |
| `V put(K key, V value)` | Associates the specified value with the specified key |
| `V remove(Object key, [Object value])` | Removes the mapping for a key if it is present |
| `boolean isEmpty()` | Returns true if this map contains no key-value mappings |
| `int size()` | Returns the number of key-value mappings |

# Example

```
Map<String, Integer> map = new HashMap<String, Integer>();


String[] keys = {"C++", "Java", "Python", "JavaScript", "Pascal"};
int[] values = {1985, 1996, 1991, 1997, 1979};
for (int i = 0; i < keys.length; i++)
    map.put(keys[i], values[i]);


System.out.println("Java was released in " + map.get("Java"));
System.out.print("Languages: " + map.keySet());
```

# Exercises

1. Write a program to manage a list of students, allowing the user to: add, remove, search by name
   - Use the `Student` class from the last exercise
   - Use `HashMap` for the search operation
2. Implement multiple hash tables for a student list so that we can find an item using different properties

# Lambda Expressions

# Introductory Example

- What is not good with this code?
  - ```
    class SortExample {
        static class StringComparator implements Comparator<String> {
            @Override
            public int compare(String s1, String s2) {
                return s1.compareTo(s2);
            }
        }

        public static void main(String args[]) {
            String[] values = {"Java", "C++", "Python", "JavaScript", "PHP"};

            List<String> list = new LinkedList<>();
            for (int i = 0; i < values.length; i++)
                list.add(values[i].toUpperCase());

            list.sort(new StringComparator());

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

# Lambda Expressions

- Lambda expressions were added in Java 8
  - Short blocks of code which take in parameters and return a value
  - Similar to methods, but they do not need a name and they can be implemented right in the body of a method
  - Make code more readable and maintainable
- Syntax:
  - Simple forms (single statement):
    - `parameter -> expression`
    - `(parameter1, parameter2) -> expression`
  - Complete form:
    - ```
      (parameter1, parameter2) -> {
          statement1;
          statement2;
      }
      ```
      ```

# Rewritten Code

```
import java.util.*;
import java.util.stream.*;

class SortExample {
    public static void main(String args[]) {
        String[] values = {"Java", "C++", "Python", "JavaScript", "PHP"};

        List<String> list = Arrays.stream(values)
            .map(value -> value.toUpperCase())
            .collect(Collectors.toList());

        list.sort((s1, s2) -> s1.compareTo(s2));

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

# Possibility to Access Local Variables from a LE

```
import java.util.*;
import java.util.stream.*;

class SortExample {
    public static void main(String args[]) {
        String[] values = {"Java", "C++", "Python", "JavaScript", "PHP"};

        List<String> list = Arrays.stream(values)
            .map(value -> value.toUpperCase())
            .collect(Collectors.toList());

        boolean ascendant = false;
        list.sort((s1, s2) -> ascendant ? s1.compareTo(s2) : s2.compareTo(s1));


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

# Lambda Expression as an Object

- A lambda expression is an object, and can be stored in a variable:
  - ```
    Comparator<String> comparator = (a, b) -> a.compareTo(b);
    list1.sort(comparator);
    list2.sort(comparator);
    ```
    ```

# Your Own Method with LE?

- Use a parameter with a single-method interface as its type:
  - ```
    import java.util.stream.*;

    class SortExample {
        interface DoubleFunction {
            double eval(double x);
        }

        static double findMax(DoubleFunction func, Double[] xa) {
            return Stream.of(xa).map(func::eval).max(Double::compare).get();
        }

        public static void main(String args[]) {
            double a = 1.3, b = -6.3, c = 4.4;
            DoubleFunction func = x -> a*x*x + b*x + c;

            Double[] xa = {1.2, 5.4, 2.4, 7.1};
            double ymax = findMax(func, xa);
            System.out.println(ymax);
        }
    }
    ```
    ```

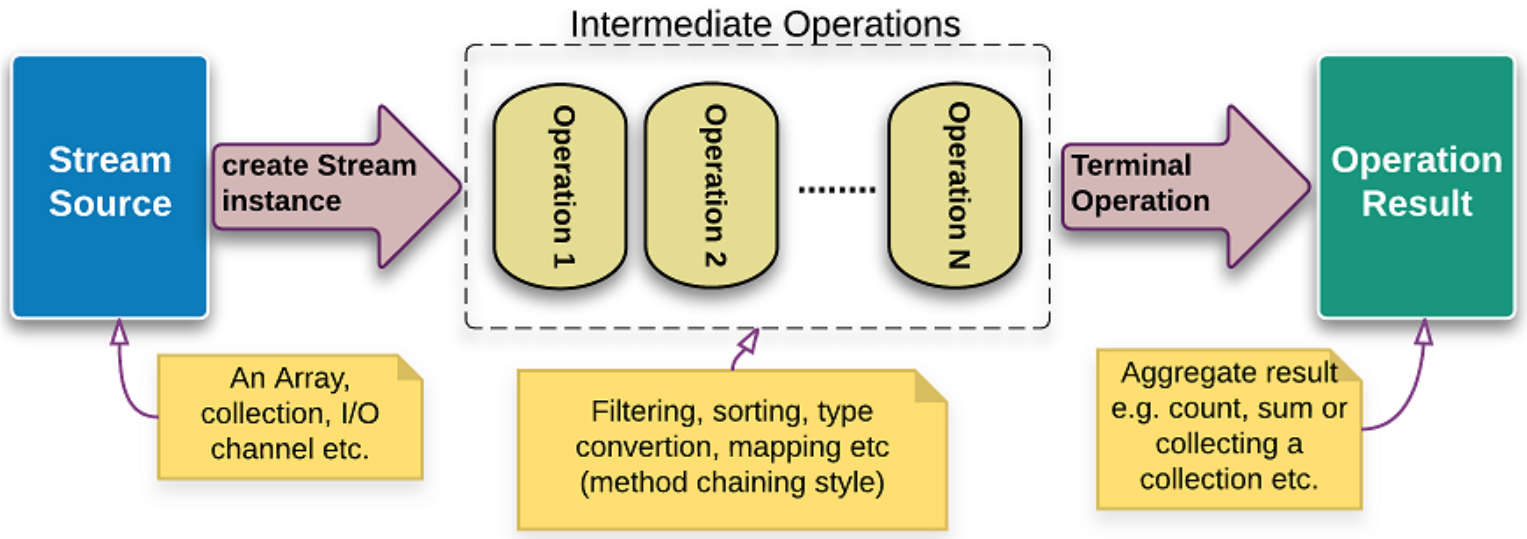
# How LE Works

- These codes are equivalent:
  - ```
    double a = 1.3, b = -6.3, c = 4.4;
    DoubleFunction func = x -> a*x*x + b*x + c;
    ```
  - ```
    double a = 1.3, b = -6.3, c = 4.4;
    DoubleFunction func = new DoubleFunction() {
        @Override
        public double eval(double x) {
            return a*x*x + b*x + c;
        }
    };
    ```

# Streams

# Introduction

- Streams are wrappers around a data source, allowing to operate with that data source and making bulk processing convenient and fast
  - Are from the `java.util.stream` package, and should not be confused with the I/O streams
  - Do not store data and it never modifies the underlying data source
  - Intermediate operations are lazy: computation on the source data is only performed when the terminal operation is initiated, and source elements are consumed only as needed
- Stream pipeline:



# Creating a Stream

- Using `Collection` interface's `stream()` method:
  - ```
    List<String> list = ...
    Stream<String> stream = list.stream();
    ```
- Using `Arrays` class's static `stream()` method:
  - ```
    String[] strs = ...
    Stream<String> stream = Arrays.stream(strs);
    ```
- Using `Stream` class's static `of()` method:
  - ```
    String[] strs = ...
    Stream stream = Stream.of(strs);
    ```
- Using `Stream.builder()`:
  - ```
    Stream.Builder<String> streamBuilder = Stream.builder();
    streamBuilder.accept("C++");
    streamBuilder.accept("Java");
    //...
    Stream<String> stream = streamBuilder.build();
    ```

# Intermediate Operations

- These methods usually accept functional interfaces as parameters and always return a new stream

| Method | Description |
|---|---|
| `<R> Stream<R> map(Function<? super T,? extends R> mapper)` | Returns a stream consisting of the results of applying the given function to the elements |
| `Stream<T> filter(Predicate<? super T> predicate)` | Returns a stream consisting of the elements of this stream that match the given predicate |
| `Stream<T> distinct()` | Returns a stream consisting of the distinct elements |
| `Stream<T> limit(long maxSize)` | Returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length |
| `Stream<T> sorted([Comparator<? super T> comparator])` | Returns a stream consisting of the elements of this stream but sorted |

# Terminal Operations

- A terminal operation is a method that produces some result, which can be a value or a new collection

| Method | Description |
|---|---|
| `long count()` | Returns the count of elements |
| `boolean allMatch/anyMatch/noneMatch (Predicate<? super T> predicate)` | Returns whether all/any/no elements of this stream match the provided predicate |
| `<R,A> R collect(Collector<? super T,A,R> collector)` | Performs a mutable reduction operation on the elements using a `Collector` |
| `Object[] toArray()` | Returns an array containing the elements |
| `void forEach(Consumer<? super T> action)` | Performs an action for each element |
| `T reduce(T identity, BinaryOperator<T> accumulator)` | Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value |

# Example

- Find the summation of the 5 smallest distinct even elements of a given array of integers
  - ```
    List<Integer> list = List.of(6, 2, 5, 1, 9, 20, 4, 6, 3, 8);

    int sum = list.stream()
        .sorted()
        .filter(a -> a % 2 == 0)
        .limit(5)
        .reduce(0, (a, b) -> a + b);

    System.out.println(sum);
    ```
    ```

# Example

- Convert a list of integers to another list consisting of their squared values that smaller than 30, sorted in descendant order
  - ```
    List<Integer> list = List.of(6, 2, 5, 1, 9, 20, 4, 6, 3, 8);

    List<Integer> result = list.stream()
            .map(a -> a * a)
            .filter(a -> a < 30)
            .sorted((a, b) -> b - a)
            .collect(Collectors.toList());

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

# Example

- Get a sorted list of all files in a given folder
  - ```
    List<String> allFileNames =
        Arrays.stream(new File("D:\\").listFiles())
            .filter(File::isFile)
            .map(File::getName)
            .sorted()
            .collect(Collectors.toList());

    System.out.println(allFileNames);
    ```
    ```

# Exercise

- Rewrite the student management program to use lambda expressions and streams