Collection

Java's Object-Oriented Programming (OOP) provides a robust framework known as **Collections**. This framework is essentially an architecture that allows the storage and manipulation of a group of objects.

What is a Collection?

In Java, a Collection is considered a single unit of objects. It provides a unified interface for manipulating and accessing these objects.

Components of the Collection Framework

The Collection framework in Java includes various interfaces and classes:

- **Interfaces**: These include Set, List, Queue, Deque, and more.
- **Classes**: Examples are ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, and TreeSet.

These interfaces and classes provide a standardized architecture for storing and manipulating a group of objects.

Operations on Collections

Java Collections can perform all operations that you would typically perform on data. These operations include:

- **Searching**: You can search for an object within a collection.
- **Sorting**: Collections can be sorted in ascending or descending order.
- **Insertion**: You can insert an object at any position within the collection.
- **Manipulation**: Collections allow you to update the value of an object.

In conclusion, the Collection framework in Java's OOP is a powerful tool for managing groups of objects. Its various classes and interfaces provide a wide range of functionalities for data manipulation.

Use of Comparator Anonymous Class

```
import java.util.*;

public class Main {
    public static void main(String[] args) {

        Comparator<Integer> comp = new Comparator<Integer>() {
            public int compare(Integer a, Integer b) {
                if(a%10 > b%10) {
                    return 1;
                } else if(a%10 < b%10) {
                    return -1;
                } else {
                    return 0;
                }
            }
        };

        List<Integer> nums = new ArrayList<>();
        nums.add(13);
        nums.add(57);
        nums.add(21);
        nums.add(79);
        nums.add(45);

        System.out.println(nums);

        Collections.sort(nums, comp);

        System.out.println(nums);
    }
}


/*Output

[13, 57, 21, 79, 45]
[21, 13, 45, 57, 79]
*/
```

In Java, an anonymous class is a type of inner class (a class defined within another class) that does not have a name. It's used when you need to use a local class only once. They are

declared and instantiated in a single expression at the point of use. Anonymous classes can either extend an existing class or implement an interface.

The provided Java code demonstrates the use of an anonymous class. Here's a breakdown of what's happening:

```
Comparator<Integer> comp = new Comparator<Integer>() {
    public int compare(Integer a, Integer b) {
        if(a%10 > b%10) {
            return 1;
        } else if(a%10 < b%10) {
            return -1;
        } else {
            return 0;
        }
    }
};
```

In this part of the code, an anonymous class is created that implements the `Comparator<Integer>` interface. The `compare` method is overridden to provide a custom comparison logic. This comparator compares the last digit (remainder of division by 10) of two integers.

```
List<Integer> nums = new ArrayList<>();
nums.add(13);
nums.add(57);
nums.add(21);
nums.add(79);
nums.add(45);

System.out.println(nums);

Collections.sort(nums, comp);

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

Here, a list of integers is created and populated with values. The list is then printed to the console, sorted using the custom comparator, and printed again. The `Collections.sort(nums, comp)` line sorts the `nums` list based on the comparison logic defined in the anonymous class. As a result, the integers in the list are sorted according to their last digit.

In conclusion, anonymous classes in Java provide a concise and powerful way to create and use classes that are needed only in a single location. They are particularly useful when you need to customize the behavior of a method in a superclass or interface, as demonstrated in the provided code.

Using Comparator in Named Class

Let's create a person class and we will do the same sorting with the person's age.

```
public class person {
    public String name;
    public int age;

    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Now we have,

```
import java.util.*;

public class Main {
    public static void main(String[] args) {

        Comparator<person> comp = new Comparator<person>() {
            public int compare(person p1, person p2) {
                if(p1.age > p2.age) {
                    return 1;
                } else if(p1.age < p2.age) {
                    return -1;
                } else {
                    return 0;
                }
            }
        };

        List<person> pers = new ArrayList<>();
        pers.add(new person("rahim", 20));
        pers.add(new person("karim", 10));
        pers.add(new person("Jodu", 60));
        pers.add(new person("Modhu", 40));
        pers.add(new person("Josim", 30));
```

```
        Collections.sort(pers, comp);

        for(person p : pers) {
            System.out.println(p.name + " " + p.age);
        }

    }
}
```

This Java code demonstrates how to sort a list of custom objects. In this case, the custom object is a `person` with the property's `name` and `age`.

```
Comparator<person> comp = new Comparator<person>() {
    public int compare(person p1, person p2) {
        if(p1.age > p2.age) {
            return 1;
        } else if(p1.age < p2.age) {
            return -1;
        } else {
            return 0;
        }
    }
};
```

This part of the code creates a `Comparator` for `person` objects. The `compare` method is overridden to provide a custom comparison logic based on the `age` property of the `person` objects. If `p1`'s age is greater than `p2`'s age, it returns `1`. If `p1`'s age is less than `p2`'s age, it returns `-1`. If both ages are equal, it returns `0`.

```
List<person> pers = new ArrayList<>();
pers.add(new person("rahim", 20));
pers.add(new person("karim", 10));
pers.add(new person("Jodu", 60));
pers.add(new person("Modhu", 40));
pers.add(new person("Josim", 30));
```

Here, a list of `person` objects is created and populated with five `person` objects.

```
Collections.sort(pers, comp);
```

This line sorts the `pers` list using the custom comparator `comp`. As a result, the `person` objects in the list are sorted in ascending order of their `age`.

```
for(person p : pers) {
    System.out.println(p.name + " " + p.age);
}
```

Finally, this part of the code prints the name and age of each `person` in the sorted list.

In conclusion, this code demonstrates how to create a custom comparator for a custom object and use it to sort a list of those objects in Java.

Implementing Comparable Interface

To get rid of the anonymous class declaration we have. We have to implement `comparable` interface in the person class. Then we have to implement `compareTo` method. In that method, we have to write the same logit we were writing in the anonymous class.

```
public class person implements Comparable<person> {
    public String name;
    public int age;

    public person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int compareTo(person that) {
        if(this.age > that.age) {
            return 1;
        } else if(this.age < that.age) {
            return -1;
        } else {
            return 0;
        }
    }
}
```

Here is a step-wise explanation what is happening,

1. A class named `person` is created that implements the `Comparable` interface. This interface allows objects of the `person` class to be compared to each other.
2. The `person` class has two properties: `name` and `age`.
3. A constructor is defined for the `person` class that takes `name` and `age` as parameters. This constructor is used to create new instances of the `person` class.
4. The `compareTo` method is overridden in the `person` class. This method is used by the `Comparable` interface to compare `person` objects based on their `age`.
5. If the `age` of the current `person` object is greater than the `age` of the `person` object it's being compared to, the method returns `1`.
6. If the `age` of the current `person` object is less than the `age` of the `person` object it's being compared to, the method returns `-1`.
7. If the `age` of both `person` objects is equal, the method returns `0`.

This code allows a list of `person` objects to be sorted by their `age` in ascending order. When the list is sorted, `person` objects with a lower `age` will come before `person` objects with a higher `age`. If two `person` objects have the same `age`, their order relative to each other will remain unchanged.

Now the main function looks like,

```
import java.util.*;

public class Main {
    public static void main(String[] args) {

        List<person> pers = new ArrayList<>();
        //add 5 person
        pers.add(new person("rahim", 20));
        pers.add(new person("karim", 10));
        pers.add(new person("Jodu", 60));
        pers.add(new person("Modhu", 40));
        pers.add(new person("Josim", 30));

        Collections.sort(pers);

        for(person p : pers) {
            System.out.println(p.name + " " + p.age);
        }

    }
}
```

The person class is sortable, searchable and divisible in different categories based on the age property.