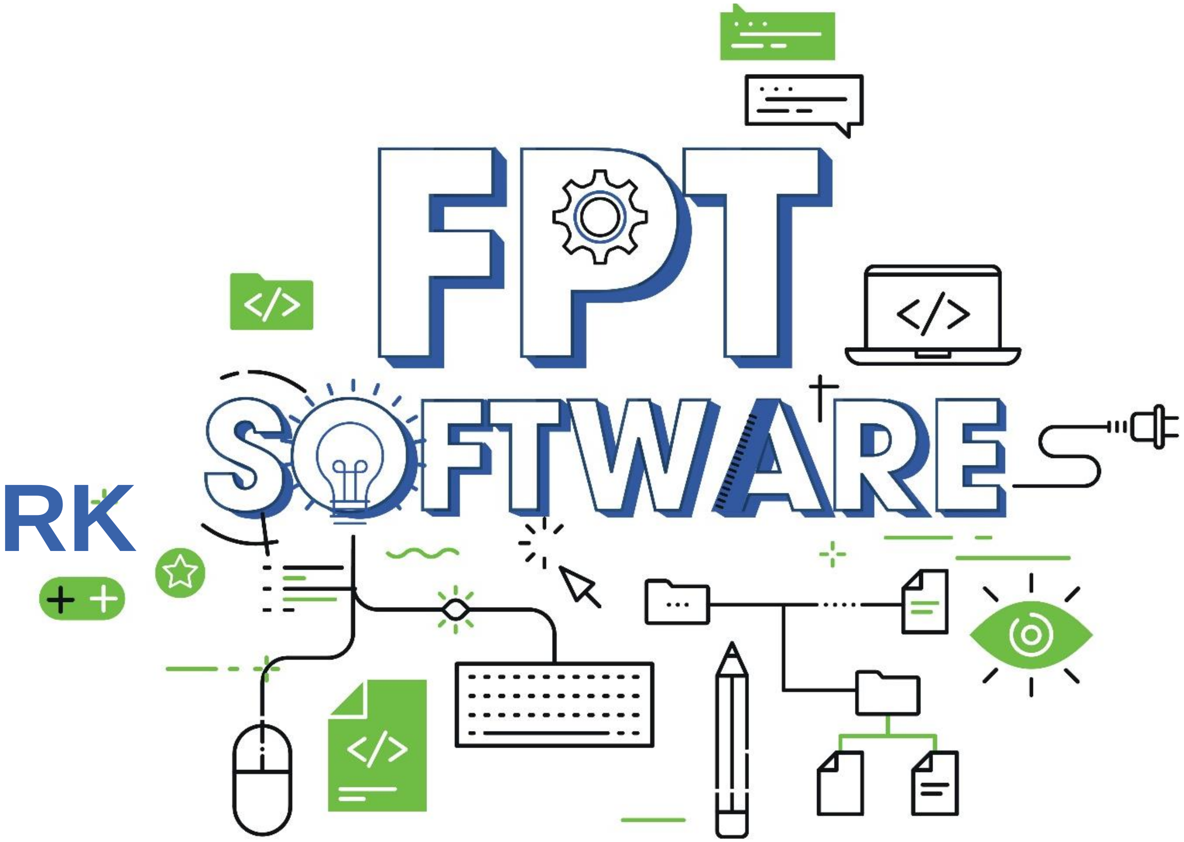
JAVA

COLLECTIONS FRAMEWORK

Set Interface



Agenda

1. Set Interface Overview
2. HashSet Class
3. TreeSet Class
4. Q & A

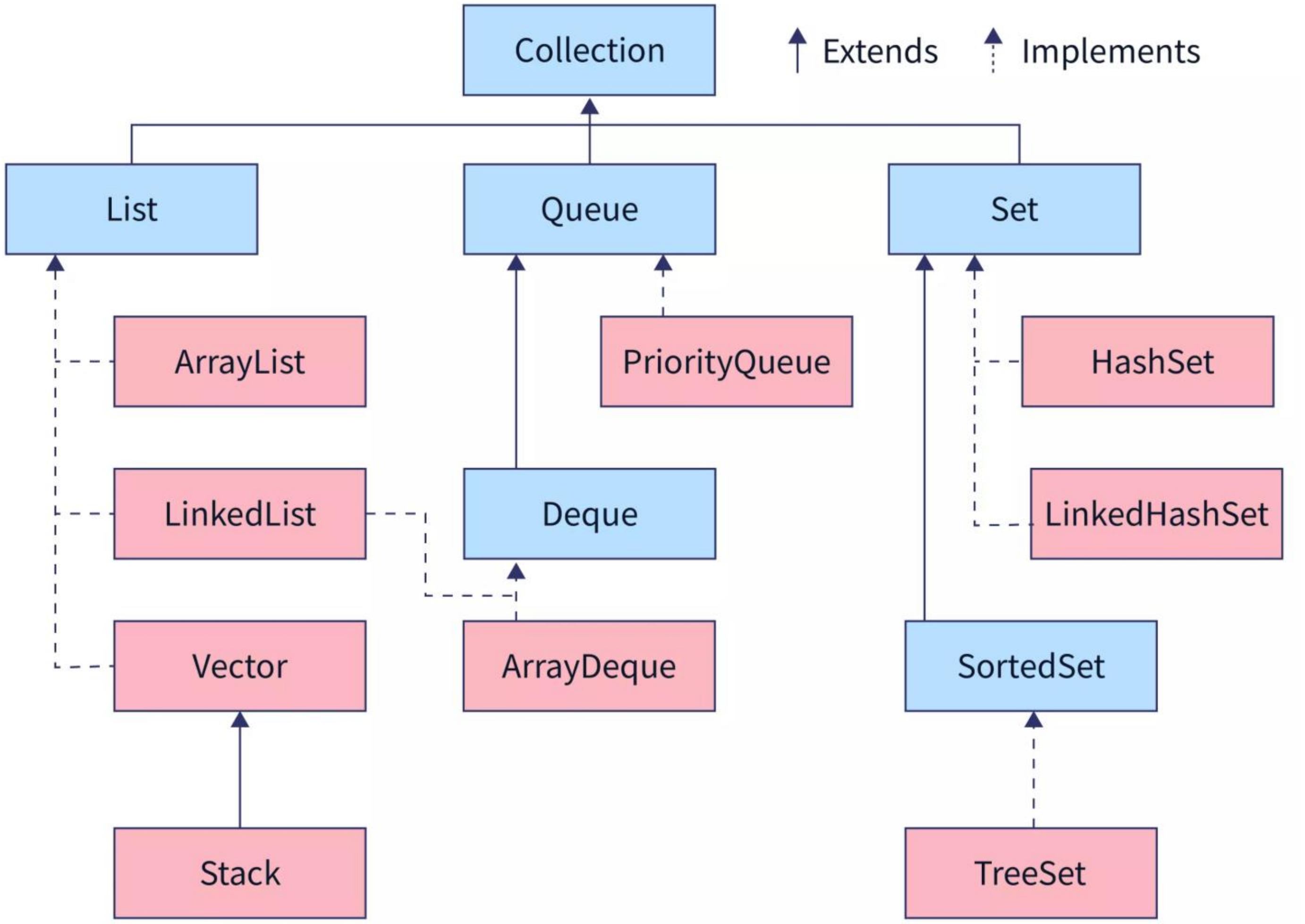
Lesson Objectives

- ***Understand*** *the characteristics of HashSet and TreeSet.*
- ***Understand*** *the difference between Set and other collection types.*
- ***Perform*** *common operations such as adding, removing, and accessing elements in an HashSet.*
- ***Perform*** *common operations such as adding, removing, and accessing elements in an TreeSet.*

Section 4

Set Interface

Hierarchy of Collections Framework - review



Set Interface

- **Set** interface in Java is present in *java.util* package.
  - ✓ It extends the Collection interface.
  - ✓ It represents the unordered set of elements which doesn't allow us to store the duplicate items.
  - ✓ We can store at most one null value in Set.
  - ✓ Set is implemented by **HashSet**, **LinkedHashSet**, and **TreeSet**.
- **Set can be instantiated as:**

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

Section 2

HashSet class

HashSet class

- Java **HashSet** class is used to create a collection that uses a hash table for storage.
  - ✓ It inherits the AbstractSet class and implements Set interface.
- The **important points** about Java HashSet class are:
  - ✓ HashSet stores the elements by using a mechanism called hashing.
  - ✓ HashSet contains unique elements only.
  - ✓ HashSet allows null value.
  - ✓ HashSet class is non synchronized.
  - ✓ HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
  - ✓ HashSet is the best approach for search operations.
  - ✓ The initial default capacity of HashSet is 16, and the load factor is 0.75.

Methods of Java HashSet class

Method	Description
boolean add(E e)	It is used to add the specified element to this set if it is not already present.
void clear()	It is used to remove all of the elements from the set.
object clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean isEmpty()	It is used to return true if this set contains no elements.
Iterator<E> iterator()	It is used to return an iterator over the elements in this set.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
int size()	It is used to return the number of elements in the set.
Spliterator<E> spliterator()	It is used to create a late-binding and fail-fast Spliterator over the elements in the set.

HashSet Example

```java
public class HashSetTest {

  public static void main(String[] args) {

    // Creating HashSet and adding elements
    HashSet<String> set = new HashSet<>();
    set.add("One");
    set.add("Two");
    set.add("Three");
    set.add("Three"); // ignoring duplicate elements
    set.add("Four");
    set.add("Five");
    Iterator<String> i = set.iterator();
    while (i.hasNext()) {
      System.out.println(i.next());
    }
  }

}
```

Output:

```
Five
One
Four
Two
Three
```

Store user-defined class objects

- Give **Course** class:

```
public class Course {
  private String courseCode;
  private String courseTitle;
  private int numOfCredits;

  public Course() {

  }

 public Course(String courseCode, String courseTitle, int numOfCredits) {
    super();
    this.courseCode = courseCode;
    this.courseTitle = courseTitle;
    this.numOfCredits = numOfCredits;
 }
  // getter, setter methods
}
```

Store user-defined class objects

- **Consider the following snippet code**:

```java
public class HashSetDemo {

  public static void main(String[] args) {
    HashSet<Course> courses = new HashSet<>();

    Course c1 = new Course("J001", "Java SE Programming Essentials", 10);
    Course c2 = new Course("S004", "SQL", 5);
    Course c3 = new Course("F003", "Front End Essentials", 10);
    Course c4 = new Course("S004", "SQL", 5);
    courses.add(c1);
    courses.add(c2);
    courses.add(c3);
    courses.add(c4);

    System.out.println("Size of set: " + courses.size());

  }
}
```

Output:

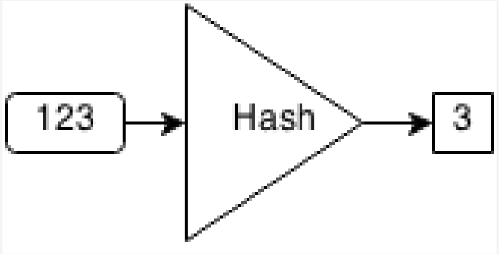
```
Size of set: 4
```

Why? c2 and c4 are the same values.

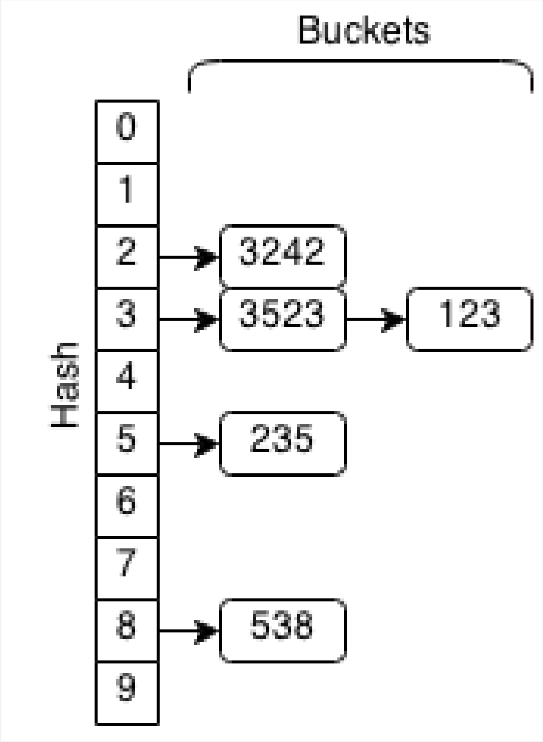
Set doesn't allow us to store the duplicate items.

Working of HashSet in Java

- **HashSet** uses hashing algorithms to **store**, **remove**, and **retrieve** its elements.
  - ✓ When an object is added to the Set, its **hash code** is used to choose a “bucket” into which to place the object.
  - ✓ Example: **`set.add (123)`**;

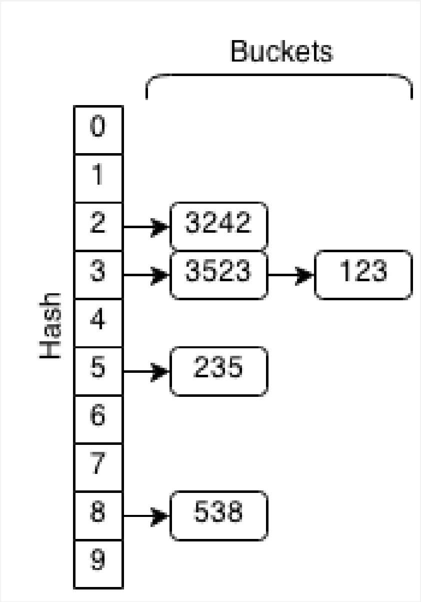


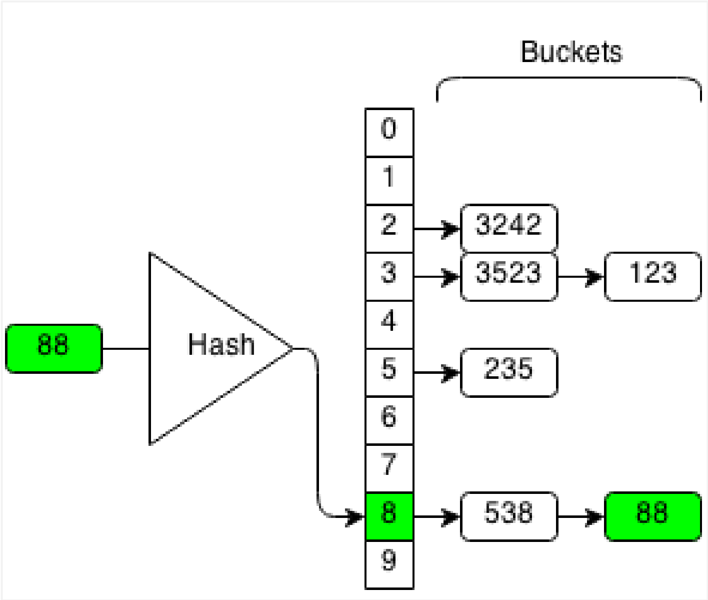
- For every element in a hash set, the hash is computed and elements with the same hash are grouped together. This group of similar hashes is **called a bucket** and they are usually stored as linked lists.



Working of HashSet in Java

- If have same hashcode then the **equals()** method will be called.
  - ✓ return **true**: the call leaves the set unchanged and returns false.
  - ✓ Return **false**: grouped together of similar hashes is **called a bucket** and they are usually stored as linked lists.
- **Example**, if we want to insert **88** in the following hash set:
  - ✓ We compute the hash of 88 which is 8, and we insert it to the end of the bucket with hash 8.





equals() and hashCode()

- Modify **Course** class, you must overriding `equals()` and `hashCode()` methods:

```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((courseCode == null) ? 0 : courseCode.hashCode());
    result = prime * result + ((courseTitle == null) ? 0 : courseTitle.hashCode());
    result = prime * result + numOfCredits;
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
    return true;
    if (obj == null)
    return false;
    if (getClass() != obj.getClass())
    return false;
    Course other = (Course) obj;
    return Objects.equals(courseCode, other.courseCode)
            && Objects.equals(courseTitle, other.courseTitle)
            && numOfCredits == other.numOfCredits;
}
```

Review Result

- **Consider the following snippet code**:

```java
public class HashSetDemo {

  public static void main(String[] args) {
    HashSet<Course> courses = new HashSet<>();

    Course c1 = new Course("J001", "Java SE Programming Essentials", 10);
    Course c2 = new Course("S004", "SQL", 5);
    Course c3 = new Course("F003", "Front End Essentials", 10);
    Course c4 = new Course("S004", "SQL", 5);
    courses.add(c1);
    courses.add(c2);
    courses.add(c3);
    courses.add(c4);

    System.out.println("Size of set: " + courses.size());

  }
}
```

Output:

```
Size of set: 3
```

Has only 1 element with value ***`("S004", "SQL", 5) in HastSet`***

HashSet

- Demo!

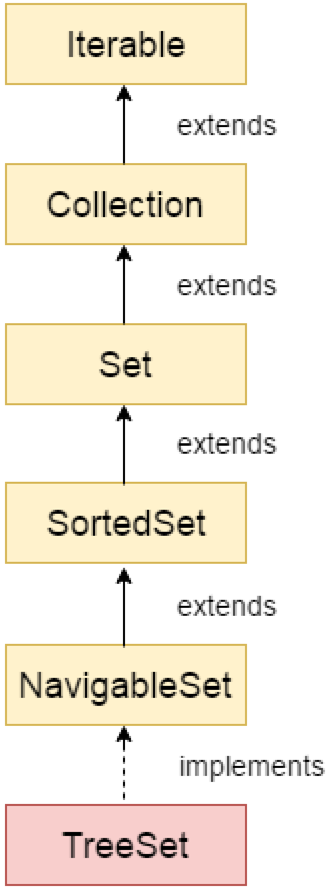
Section 3

TreeSet class

TreeSet

- Java **TreeSet** class implements the Set interface that **uses a tree for storage**. It inherits *AbstractSet* class and implements the *NavigableSet* interface. The objects of the TreeSet class are stored in *ascending order*.
- The **important points** about Java TreeSet class are:
  - ✓ Java TreeSet class contains unique elements only like HashSet.
  - ✓ Java TreeSet class access and retrieval times are quiet fast.
  - ✓ Java TreeSet class doesn't allow null element.
  - ✓ Java TreeSet class is non synchronized.
  - ✓ Java TreeSet class maintains ascending order.
- **Constructors:**

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
TreeSet(Collection<? extends E> c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator<? super E> comparator)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet<E> s)	It is used to build a TreeSet that contains the elements of the given SortedSet.



Methods of Java TreeSet class

Method	Description
boolean **add**(E e)	It is used to add the specified element to this set if it is not already present.
boolean **addAll**(Collection<? extends E> c)	It is used to add all of the elements in the specified collection to this set.
E **ceiling**(E e)	It returns the equal or closest greatest element of the specified element from the set, or null there is no such element.
Iterator **descendingIterator**()	It is used iterate the elements in descending order.
NavigableSet **descendingSet**()	It returns the elements in reverse order.
E **floor**(E e)	It returns the equal or closest least element of the specified element from the set, or null there is no such element.
SortedSet **headSet**(E toElement)	It returns the group of elements that are less than the specified element.
NavigableSet **headSet**(E toElement, boolean inclusive)	It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element.
E **higher**(E e)	It returns the closest greatest element of the specified element from the set, or null there is no such element.

Methods of Java TreeSet class

Method	Description
E **lower**(E e)	It returns the closest least element of the specified element from the set, or null there is no such element.
E **pollFirst**()	It is used to retrieve and remove the lowest(first) element.
E **pollLast**()	It is used to retrieve and remove the highest(last) element.
SortedSet **tailSet**(E fromElement)	It returns a set of elements that are greater than or equal to the specified element.
NavigableSet **tailSet**(E fromElement, boolean inclusive)	It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element.
E **first**()	It returns the first (lowest) element currently in this sorted set.
E **last**()	It returns the last (highest) element currently in this sorted set.
int **size**()	It returns the number of elements in this set.

TreeSet Example

```java
public class TreeSetTest {
  public static void main(String[] args) {
    // Creating and adding elements
    TreeSet<String> al = new TreeSet<String>();
    al.add("Ravi");
    al.add("Vijay");
    al.add("Ravi");
    al.add("Ajay");
    // Traversing elements
    Iterator<String> itr = al.iterator();
    while (itr.hasNext()) {
       System.out.println(itr.next());
    }
  }
}
```

Output:

```
Ajay
Ravi
Vijay
```

*String and Integer both implement the **Comparable** interface in Java!*

By default, the objects or elements of the TreeSet are stored according to the natural ordering in ascending order.

TreeSet Example

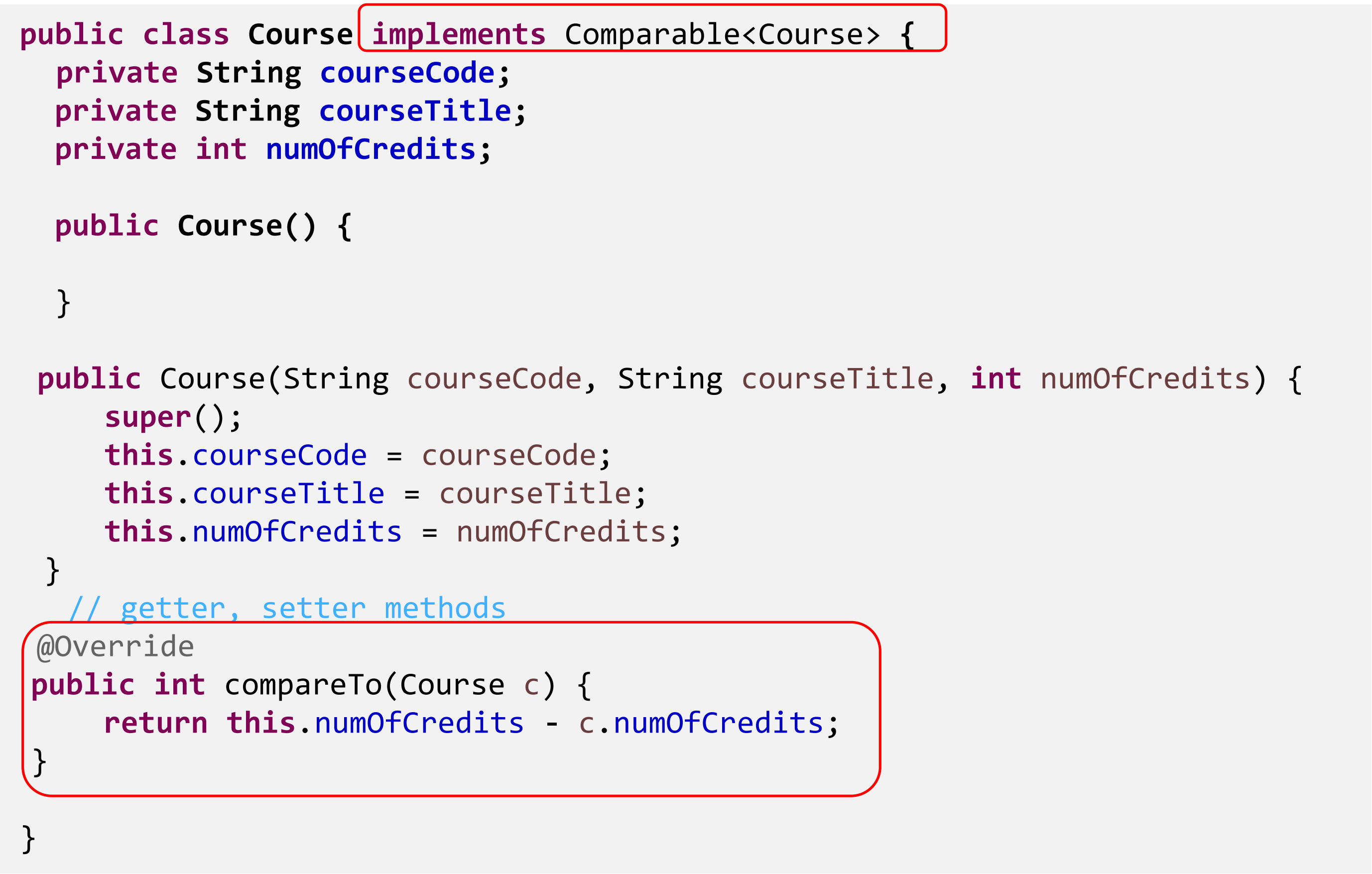
```java
public class TreeSetTest2 {

  public static void main(String[] args) {
    TreeSet<Integer> set = new TreeSet<Integer>();
    set.add(24);
    set.add(66);
    set.add(12);
    set.add(15);
    System.out.println("Initial  Set: " + set);

    System.out.println("Reverse Set: " + set.descendingSet());

    System.out.println("Highest Value: " + set.pollFirst());
    System.out.println("Lowest Value: " + set.pollLast());
  }
}
```

Output:

```
Initial  Set: [12, 15, 24, 66]
Reverse Set: [66, 24, 15, 12]
Highest Value: 12
Lowest Value: 66
```

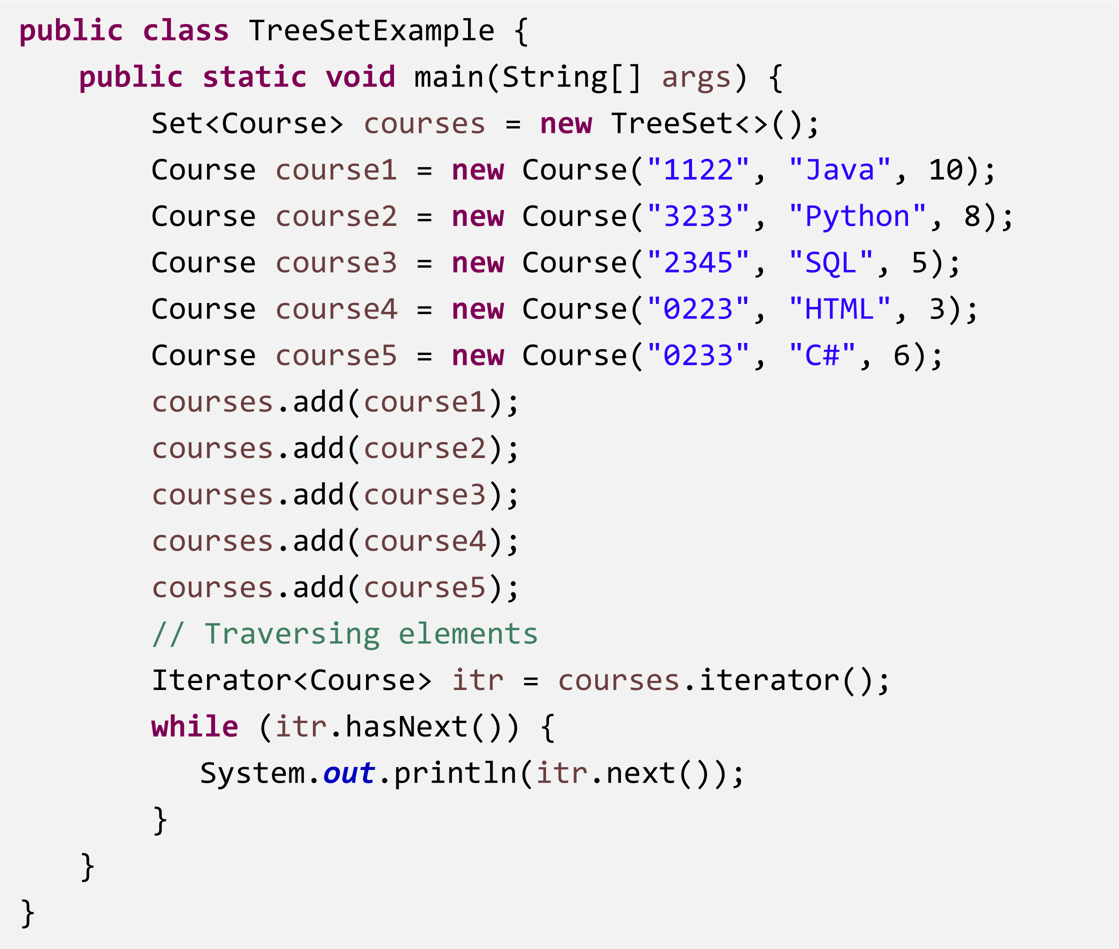
Java TreeSet Example: Course class

- Let's see a TreeSet example where we are *adding courses* to the set and *printing all* the courses.
- **String** and **Wrapper** classes are *Comparable by default*
- *To add **user-defined objects** in TreeSet, you need to implement the Comparable interface.*



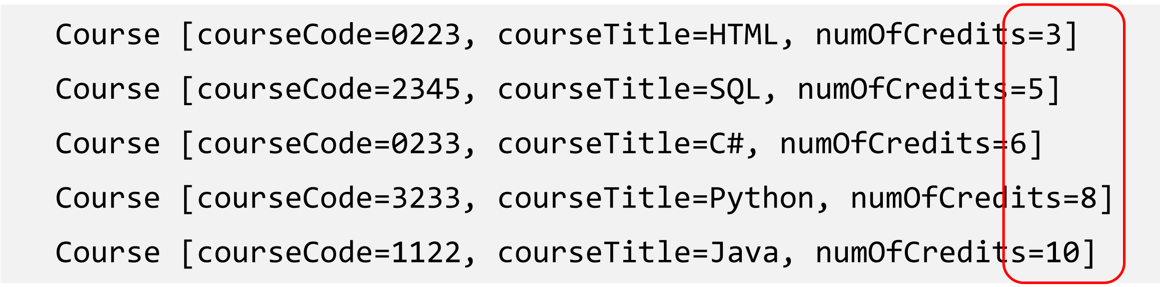
```
public class Course implements Comparable<Course> {
  private String courseCode;
  private String courseTitle;
  private int numOfCredits;

  public Course() {

  }

 public Course(String courseCode, String courseTitle, int numOfCredits) {
     super();
     this.courseCode = courseCode;
     this.courseTitle = courseTitle;
     this.numOfCredits = numOfCredits;
 }
   // getter, setter methods
 @Override
 public int compareTo(Course c) {
     return this.numOfCredits - c.numOfCredits;
 }

}
```

Java TreeSet Example: Course class



```java
public class TreeSetExample {
    public static void main(String[] args) {
        Set<Course> courses = new TreeSet<>();
        Course course1 = new Course("1122", "Java", 10);
        Course course2 = new Course("3233", "Python", 8);
        Course course3 = new Course("2345", "SQL", 5);
        Course course4 = new Course("0223", "HTML", 3);
        Course course5 = new Course("0233", "C#", 6);
        courses.add(course1);
        courses.add(course2);
        courses.add(course3);
        courses.add(course4);
        courses.add(course5);
        // Traversing elements
        Iterator<Course> itr = courses.iterator();
        while (itr.hasNext()) {
           System.out.println(itr.next());
        }
    }
}
```

Output:



```
Course [courseCode=0223, courseTitle=HTML, numOfCredits=3]
Course [courseCode=2345, courseTitle=SQL, numOfCredits=5]
Course [courseCode=0233, courseTitle=C#, numOfCredits=6]
Course [courseCode=3233, courseTitle=Python, numOfCredits=8]
Course [courseCode=1122, courseTitle=Java, numOfCredits=10]
```

If we add an object of the class that is not implementing the Comparable interface, the ClassCastException is raised.

TreeSet Comparator

- The **TitleComparator** class implements the `Comparator` interface.

```
public class TitleComparator implements Comparator<Course> {
  @Override
  public int compare(Course c1, Course c2) {
      return c1.getCourseTitle().compareTo(c2.getCourseTitle());
  }
}
```

- Create **TreeSet** with **Comparator**:

```
Set<Course> courses = new TreeSet<>(new NameComparator());
```

- **Output**:

```
Course [courseCode=0233, courseTitle=C#, numOfCredits=6]
Course [courseCode=0223, courseTitle=HTML, numOfCredits=3]
Course [courseCode=1122, courseTitle=Java, numOfCredits=10]
Course [courseCode=3233, courseTitle=Python, numOfCredits=8]
Course [courseCode=2345, courseTitle=SQL, numOfCredits=5]
```

Difference between Hashset and Treeset in Java

Parameters	HashSet	TreeSet
Ordering or Sorting	It does not provide a guarantee to sort the data.	It provides a guarantee to sort the data. *The sorting depends on the supplied Comparator.*
Null Objects	In HashSet, **only an element** can be null.	It does not allow null elements.
Comparison	It uses **hashCode()** or **equals()** method for comparison.	It uses **compare()** or **compareTo()** method for comparison.
Performance	It is **faster** than TreeSet.	It is **slower** in comparison to HashSet.
Implementation	Internally it uses **HashMap** to store its elements.	Internally it uses **TreeMap** to store its elements.
Data Structure	HashSet is backed up by a hash table.	TreeSet is backed up by a Red-black Tree.
Values Stored	It allows only **heterogeneous** values.	It allows only **homogeneous** values.

TreeSet

- Demo!

Agenda

1 Set Interface Overview

2 HashSet Class

3 TreeSet Class

4 Q & A

Software

CODE YOUR LIFE

THANK YOU!