12-1

Generics & Comparable Comparator

Objectives

After completing this lesson, you should be able to:

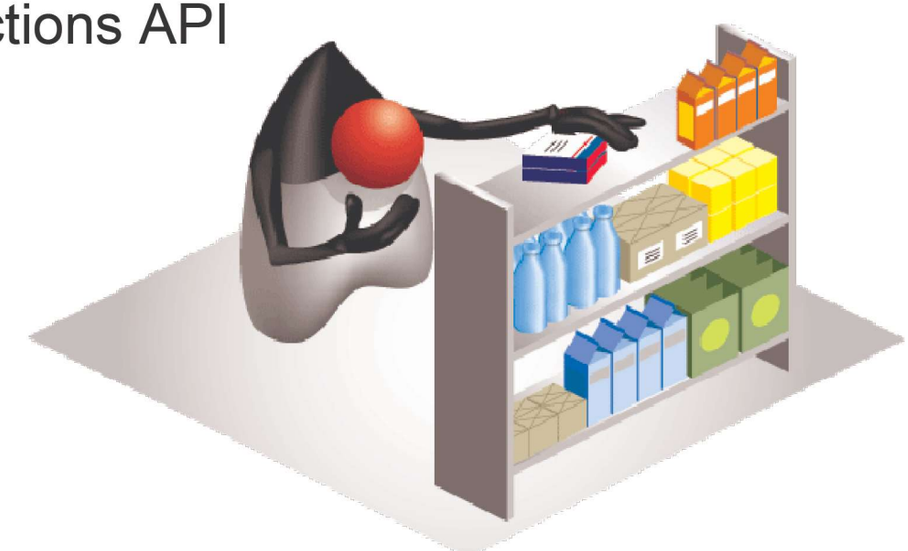
- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Order collections

Topics

- Generics
  - Generics with Type Inference Diamond
- Collections
  - Collection Types
  - `List` Interface
  - `ArrayList` Implementation
  - Autoboxing and Unboxing
  - `Set` Interface
  - `Map` Interface
  - `Deque` Interface
  - Ordering Collections
  - `Comparable` Interface
  - `Comparator` Interface

Generics

- Provide flexible type safety to your code
- Move many common errors from run time to compile time
- Provide cleaner, easier-to-write code
- Reduce the need for casting with collections
- Are used heavily in the Java Collections API



Simple Cache Class Without Generics

```
public class CacheString {
  private String message;
    public void add(String message){
        this.message = message;
    }

    public String get(){
        return this.message;
    }
}
```

```
public class CacheShirt {
    private Shirt shirt;

public void add(Shirt shirt){
       this.shirt = shirt;
              }

     public Shirt get(){
        return this.shirt;
              }
            }
```

Generic Cache Class

```
1   public class CacheAny <T>{
2
3       private T t;
4
5       public void add(T t){
6           this.t = t;
7       }
8
9       public T get(){
10          return this.t;
11      }
12  }
```

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
1   public static void main(String args[]){
2      CacheString myMessage = new CacheString(); // Type
3      CacheShirt myShirt = new CacheShirt();     // Type
4
5      //Generics
6      CacheAny<String> myGenericMessage = new CacheAny<String>();
7      CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();
8
9      myMessage.add("Save this for me"); // Type
10     myGenericMessage.add("Save this for me"); // Generic
11
12  }
```

Generics with Type Inference Diamond

- Syntax:
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

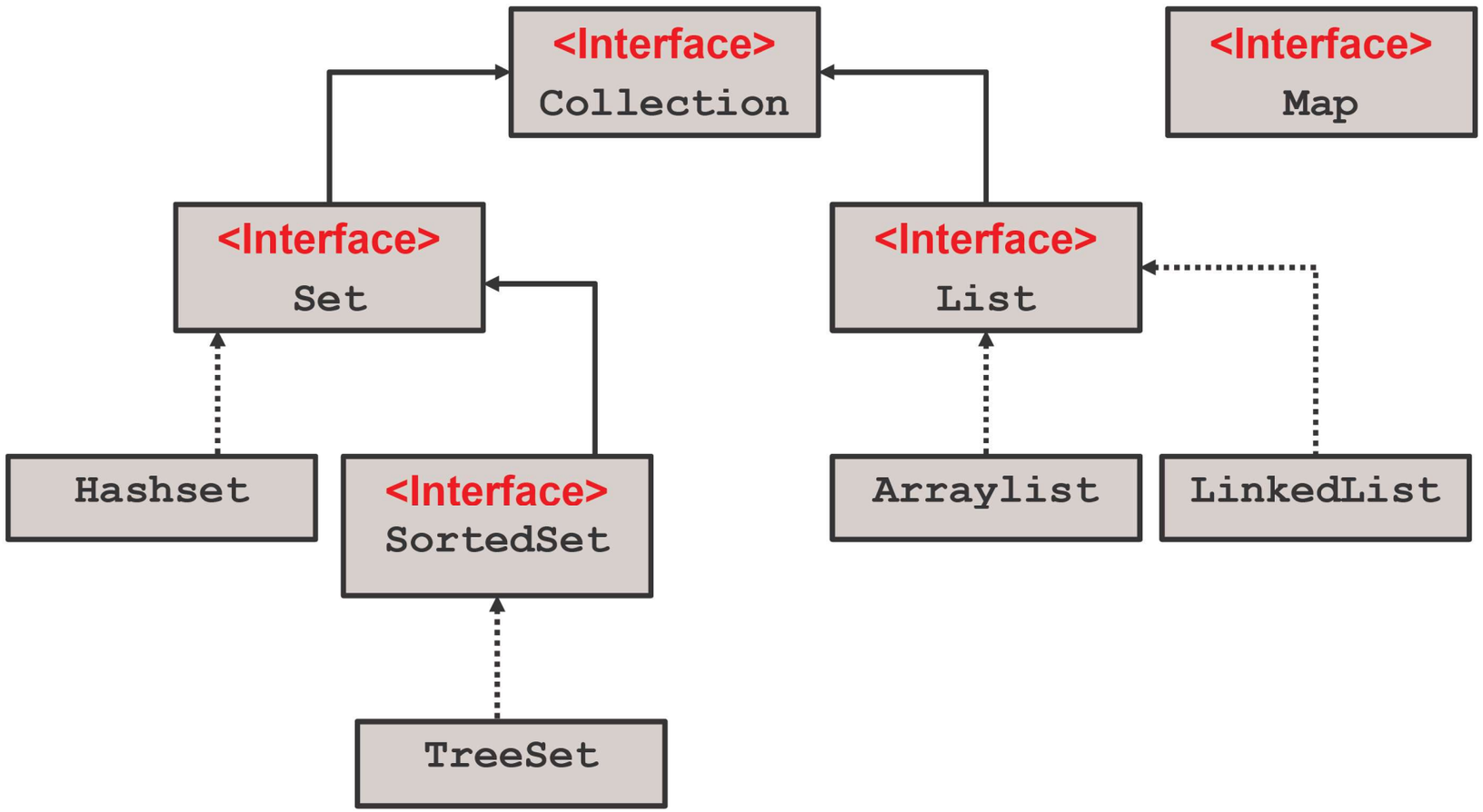
```
//Generics
CacheAny<String> myMessage = new CacheAny<>();
}
```

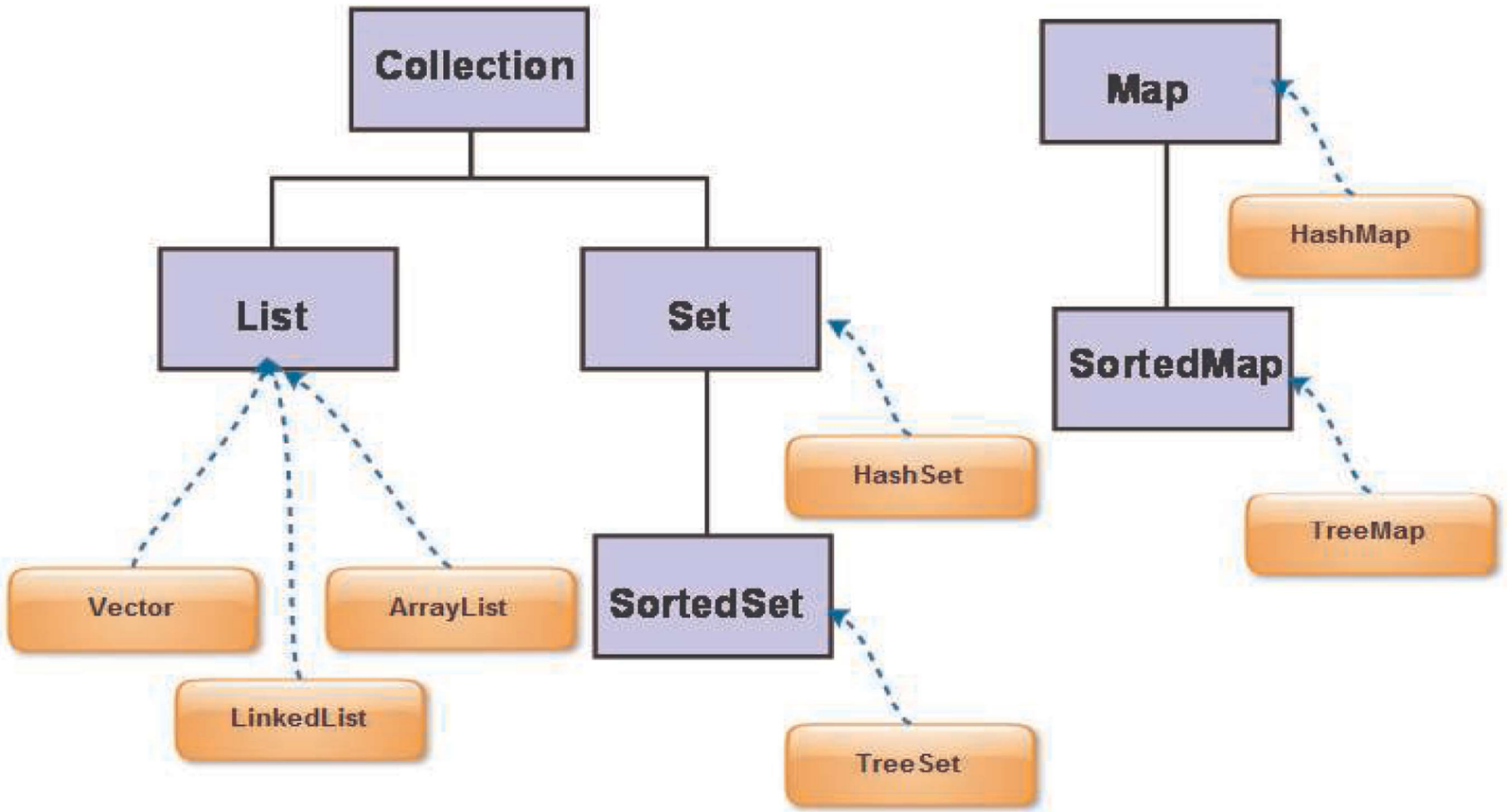
Java Collections Framework

Java Collections Framework is an API architecture for managing a group of objects that can be manipulated independently of their internal implementation. It is:

- Found in the `java.util` package
- Defined by six core interfaces and some implementation classes:
  - `Collection` interface: A generic group of elements
  - `Set` interface: A group of unique elements
  - `List` interface: An ordered group of elements
  - `Map` interface: A group of unique keys and their values
  - `SortedSet` and `SortedMap` for a sorted `Set` and `Map`

Collection Types




Collection
List
Set
Map
HashMap
SortedMap
HashSet
TreeMap
Vector
ArrayList
SortedSet
LinkedList
TreeSet

Collection Interfaces and Implementation

Interface	Implementation		
List	ArrayList	LinkedList	
Set	TreeSet	HashSet	LinkedHashSet
Map	HashMap	HashTable	TreeMap
Deque	ArrayDeque		

Collections Framework Components

The Java Collections Framework is a set of interfaces and classes used to store and manipulate groups of data as a single unit.

- Core interfaces are the interfaces used to manipulate collections and to pass them from one method to another.
- Implementations are the actual data objects used to store collections; the data structures implement the core collection interface.
- Algorithms are pieces of reusable functionality provided by the JDK.

`List` Interface

- `List` defines generic list behavior.
  - Is an ordered collection of elements
- `List` behaviors include:
  - Adding elements at a specific index
  - Getting an element based on an index
  - Removing an element based on an index
  - Overwriting an element based on an index
  - Getting the size of the list
- `List` allows duplicate elements.

ArrayList

- Is an implementation of the `List` interface
  - The list automatically grows if elements exceed initial size.
- Has a numeric index
  - Elements are accessed by index.
  - Elements can be inserted based on index.
  - Elements can be overwritten.
- Allows duplicate items

```
List<Integer> partList = new ArrayList<>(3);
partList.add(new Integer(1111));
partList.add(new Integer(2222));
partList.add(new Integer(3333));
partList.add(new Integer(4444)); // ArrayList auto grows
System.out.println("First Part: " + partList.get(0)); //
First item
partList.add(0, new Integer(5555)); // Insert an item by
index
```

Autoboxing and Unboxing

- Simplifies syntax
- Produces cleaner, easier-to-read code

```
1   public class AutoBox {
2      public static void main(String[] args){
3          Integer intObject = new Integer(1);
4          int intPrimitive = 2;
5
6          Integer tempInteger;
7          int tempPrimitive;
8
9          tempInteger = new Integer(intPrimitive);
10         tempPrimitive = intObject.intValue();
11
12         tempInteger = intPrimitive; // Auto box
13         tempPrimitive = intObject;  // Auto unbox
```

ArrayList Without Generics

```
1  public class OldStyleArrayList {
2    public static void main(String args[]){
3      List partList = new ArrayList(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Oops a string!");
9
10     Iterator elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = (Integer)(elements.next()); // error?
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

Java example using syntax prior to Java 1.5

Runtime error: `ClassCastException`

Generic ArrayList

```
1  public class GenericArrayList {
2    public static void main(String args[]) {
3      List<Integer> partList = new ArrayList<>(3);
4
5      partList.add(new Integer(1111));
6      partList.add(new Integer(2222));
7      partList.add(new Integer(3333));
8      partList.add("Bad Data");  // compiler error now
9
10     Iterator<Integer> elements = partList.iterator();
11     while (elements.hasNext()) {
12       Integer partNumberObject = elements.next();
13       int partNumber = partNumberObject.intValue();
14
15       System.out.println("Part number: " + partNumber);
16     }
17   }
18 }
```

No cast required.

Generic `ArrayList`: Iteration and Boxing

```
for (Integer partNumberObj:partList){
      int partNumber = partNumberObj; // Demos auto unboxing
      System.out.println("Part number: " + partNumber);
}
```

- The enhanced `for` loop, or `for-each` loop, provides cleaner code.
- No casting is done because of autoboxing and unboxing.

`Iterator` Interface

The `Iterator` interface, which is part of Java Collections Framework, can be used to process a series of objects. The `java.util.Iterator` interface:

- Implements an object-oriented approach for accessing elements in a collection
- Replaces the `java.util.Enumeration` approach
- Contains the following methods:
  - `hasNext()` returns `true` if more elements exist.
  - `next()` returns the next `Object`, if any.
  - `remove()` removes the last element returned.

`Set` Interface

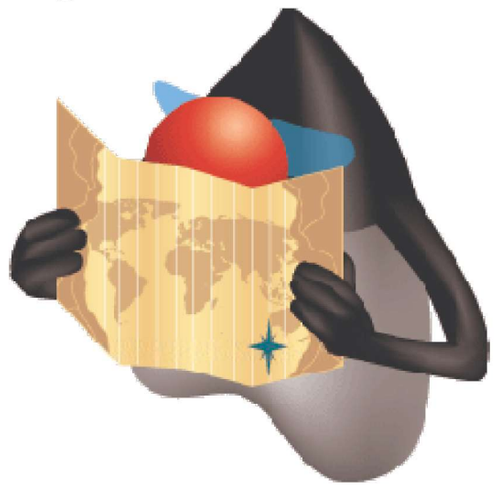
- A `Set` is an interface that contains only unique elements.
- A `Set` has no index.
- Duplicate elements are not allowed.
- You can iterate through elements to access them.
- `TreeSet` provides sorted implementation.

TreeSet: Implementation of Set

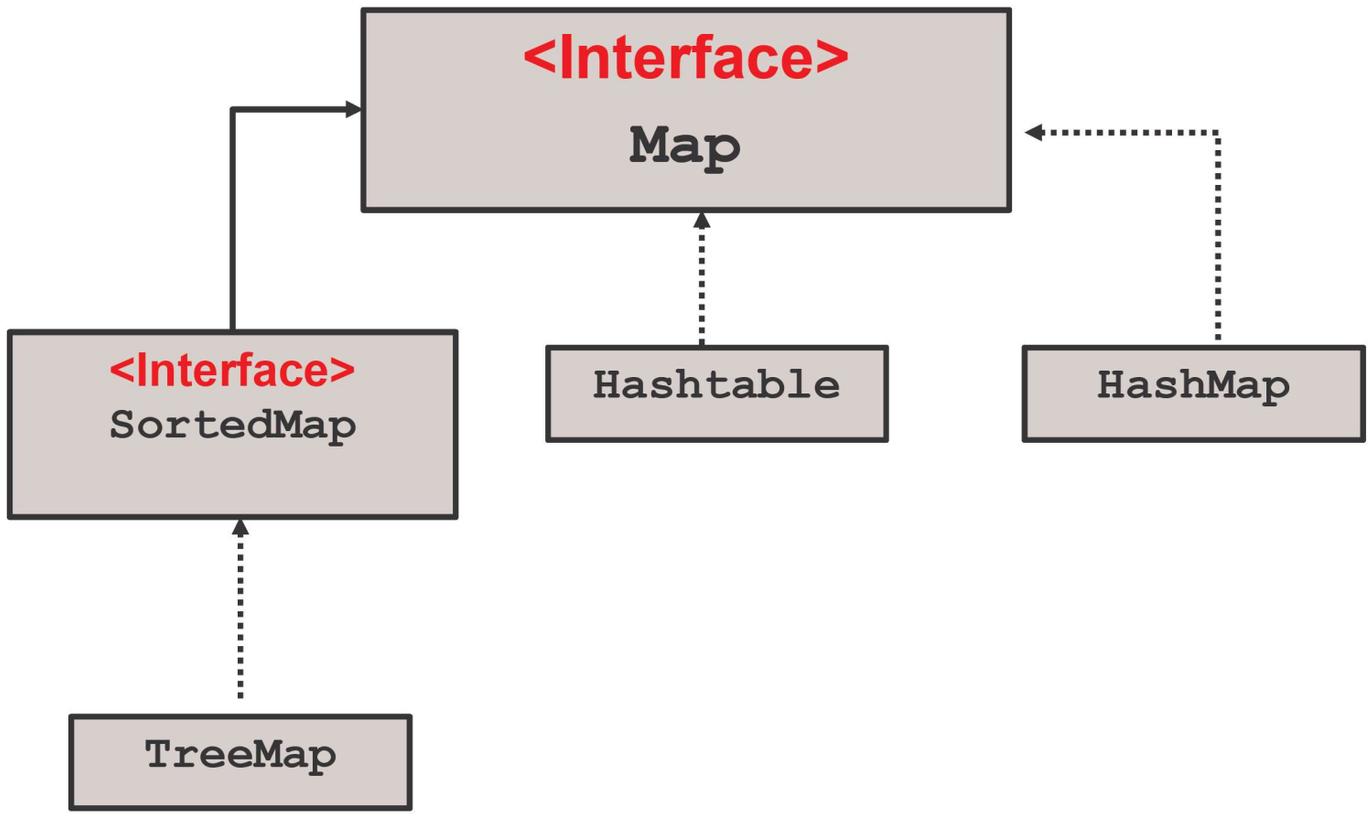
```
1   public class SetExample {
2      public static void main(String[] args){
3          Set<String> set = new TreeSet<>();
4
5          set.add("one");
6          set.add("two");
7          set.add("three");
8          set.add("three"); // not added, only unique
9
10         for (String item:set){
11             System.out.println("Item: " + item);
12         }
13     }
14  }
```

- A collection that stores multiple key-value pairs
  - Key: Unique identifier for each element in a collection
  - Value: A value stored in the element associated with the key
- Called “associative arrays” in other languages

Key	Value
101	Blue Shirt
102	Black Shirt
103	Gray Shirt



Map Types

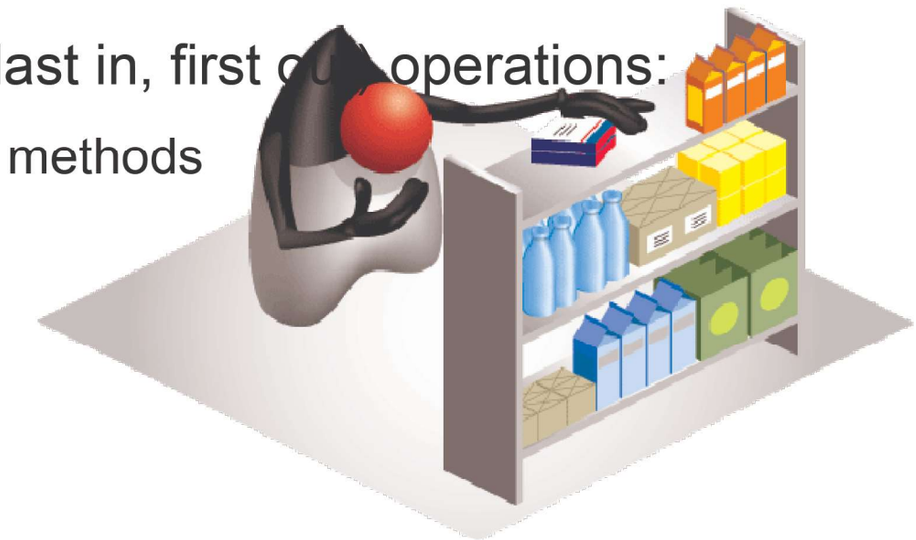

<Interface>
Map
<Interface>
SortedMap
Hashtable
HashMap
TreeMap

TreeMap: Implementation of Map

```
1   public class MapExample {
2      public static void main(String[] args){
3          Map <String, String> partList = new TreeMap<>();
4          partList.put("S001", "Blue Polo Shirt");
5          partList.put("S002", "Black Polo Shirt");
6          partList.put("H001", "Duke Hat");
7
8          partList.put("S002", "Black T-Shirt"); // Overwrite value
9          Set<String> keys = partList.keySet();
10
11         System.out.println("=== Part List ===");
12         for (String key:keys){
13             System.out.println("Part#: " + key + " " +
14                                partList.get(key));
15         }
16     }
17  }
```

A collection that can be used as a stack or a queue

- It means a "double-ended queue" (and is pronounced "deck").
- A queue provides FIFO (first in, first out) operations:
  - `add(e)` and `remove()` methods
- A stack provides LIFO (last in, first out) operations:
  - `push(e)` and `pop()` methods



Stack with `Deque`: Example

```
1   public class TestStack {
2      public static void main(String[] args){
3          Deque<String> stack = new ArrayDeque<>();
4          stack.push("one");
5          stack.push("two");
6          stack.push("three");
7
8          int size = stack.size() - 1;
9          while (size >= 0 ) {
10             System.out.println(stack.pop());
11             size--;
12         }
13     }
14  }
```

Ordering Collections

- The `Comparable` and `Comparator` interfaces are used to sort collections.
  - Both are implemented by using generics.
- Using the `Comparable` interface:
  - Overrides the `compareTo` method
  - Provides only one sort option
- The `Comparator` interface:
  - Is implemented by using the `compare` method
  - Enables you to create multiple `Comparator` classes
  - Enables you to create and use numerous sorting options

Comparable: Example

```
1   public class ComparableStudent implements
    Comparable<ComparableStudent>{
2       private String name; private long id = 0; private double gpa = 0.0;
3
4       public ComparableStudent(String name, long id, double gpa){
5          // Additional code here
6       }
7       public String getName(){ return this.name; }
8          // Additional code here
9
10      public int compareTo(ComparableStudent s){
11          int result = this.name.compareTo(s.getName());
12          if (result > 0) { return 1; }
13          else if (result < 0){ return -1; }
14          else { return 0; }
15      }
16  }
```

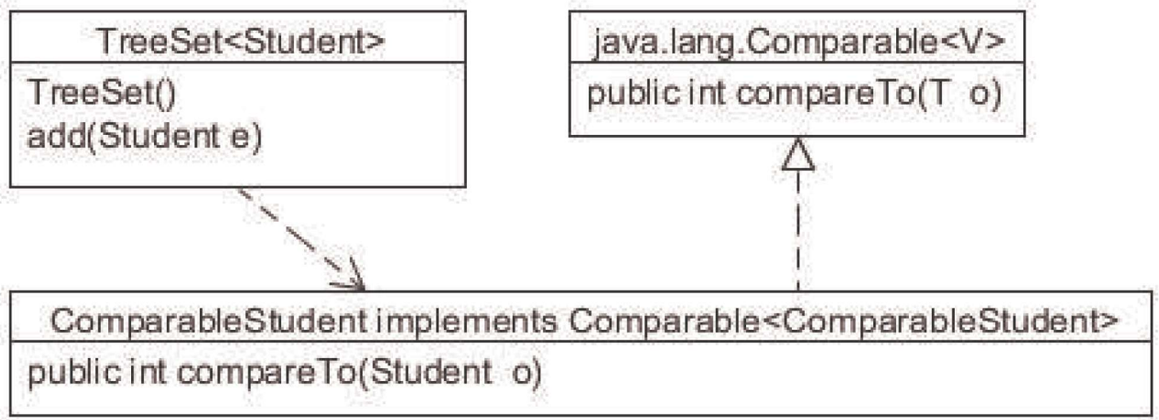
Comparable Test: Example

```
public class TestComparable {
  public static void main(String[] args){
    Set<ComparableStudent> studentList = new TreeSet<>();

    studentList.add(new ComparableStudent("Thomas Jefferson", 1111, 3.8));
    studentList.add(new ComparableStudent("John Adams", 2222, 3.9));
    studentList.add(new ComparableStudent("George Washington", 3333, 3.4));

    for(ComparableStudent student:studentList){
      System.out.println(student);
    }
  }
}
```

`Comparable` Interface

Using the `Comparable` interface:

- Overrides the `compareTo` method
- Provides only one sort option



Sorting logic is inside of the Student class.
Benefit: Sorting can leverage private fields to determine order.
Drawback: Only one ordering behavior is possible.

`Comparator` Interface

- Is implemented by using the `compare` method
- Enables you to create multiple `Comparator` classes
- Enables you to create and use numerous sorting options

Comparator: Example

```
public class StudentSortName implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        int result = s1.getName().compareTo(s2.getName());
        if (result != 0) { return result; }
        else {
            return 0;  // Or do more comparing
        }
    }
}
```

```
public class StudentSortGpa implements Comparator<Student>{
    public int compare(Student s1, Student s2){
        if (s1.getGpa() < s2.getGpa()) { return 1; }
        else if (s1.getGpa() > s2.getGpa()) { return -1; }
        else { return 0; }
    }
}
```

Here the compare logic is reversed and results in descending order.

Comparator Test: Example

```
1     public class TestComparator {
2       public static void main(String[] args){
3           List<Student> studentList = new ArrayList<>(3);
4           Comparator<Student> sortName = new StudentSortName();
5           Comparator<Student> sortGpa = new StudentSortGpa();
6
7           // Initialize list here
8           Collections.sort(studentList, sortName);
9           for(Student student:studentList){
10              System.out.println(student);
11          }
12          Collections.sort(studentList, sortGpa);
13          for(Student student:studentList){
14              System.out.println(student);
15          }
16      }
17  }
```

Summary

In this lesson, you should have learned how to:

- Create a custom generic class
- Use the type inference diamond to create an object
- Create a collection without using generics
- Create a collection by using generics
- Order collections