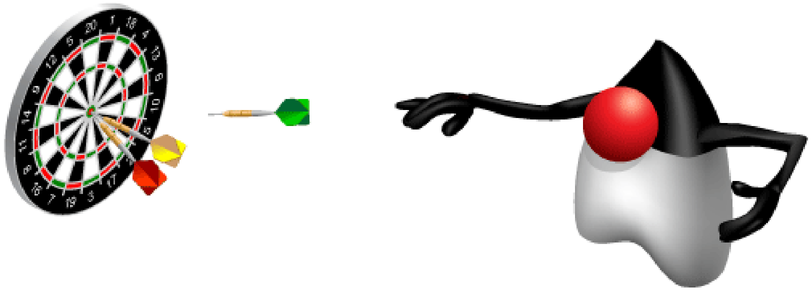
7

Generics and Collections

Objectives

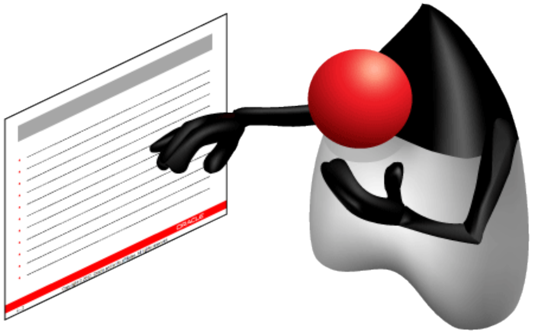
After completing this lesson, you should be able to:

- 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
- Implement an `ArrayList`
- Implement a `TreeSet`
- Implement a `HashMap`
- Implement a `Deque`
- 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

```
public class CacheAny <T>{

   private T t;

   public void add(T t){
       this.t = t;
   }

   public T get(){
        return this.t;
    }
 }
```

Generics in Action

Compare the type-restricted objects to their generic alternatives.

```
public static void main(String args[]){
   CacheString myMessage = new CacheString(); // Type
   CacheShirt myShirt = new CacheShirt();     // Type

   //Generics
   CacheAny<String> myGenericMessage = new CacheAny<String>();
   CacheAny<Shirt> myGenericShirt = new CacheAny<Shirt>();

   myMessage.add("Save this for me"); // Type
    myGenericMessage.add("Save this for me"); // Generic

 }
```

Generics with Type Inference Diamond

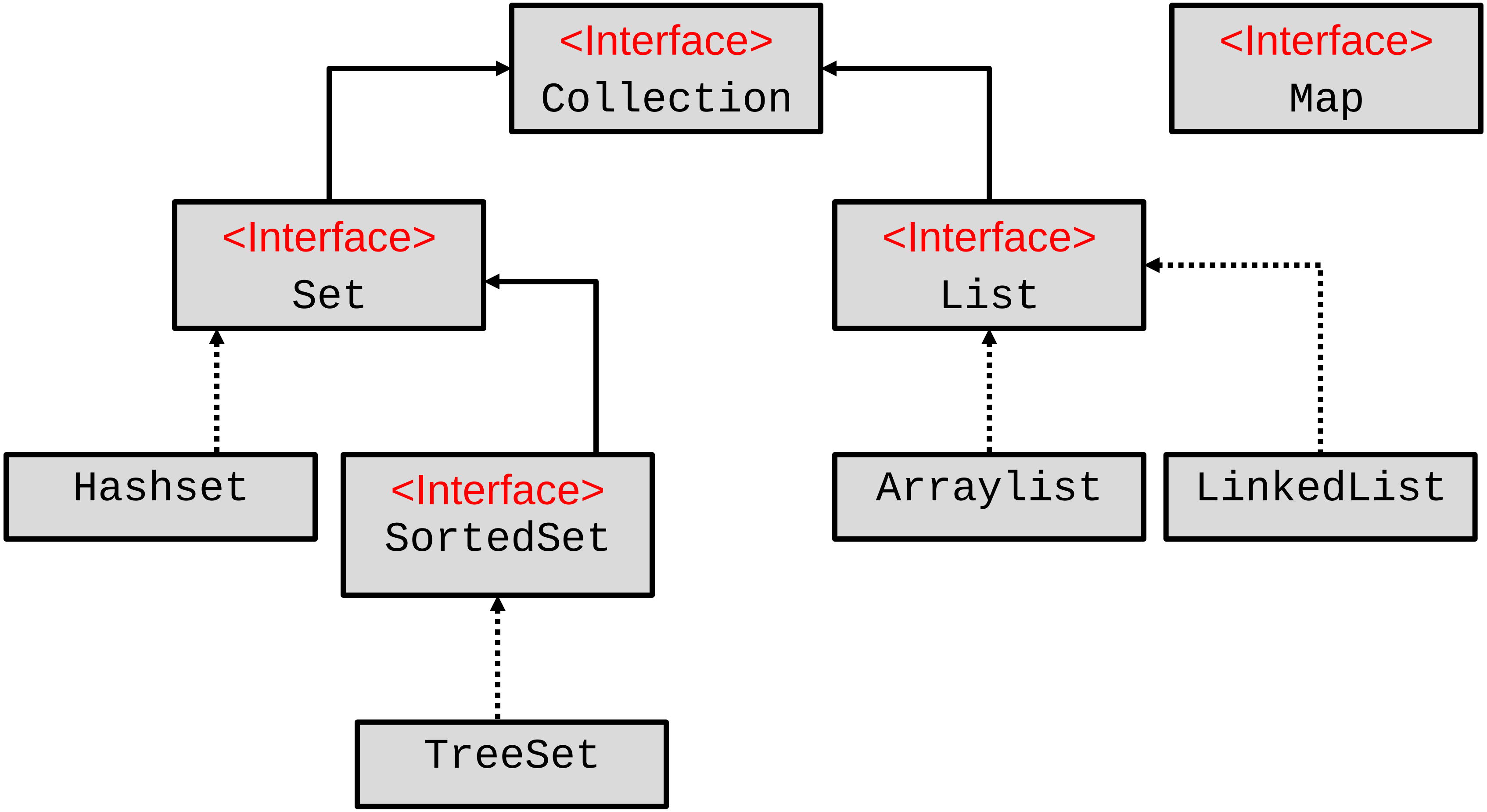
- Syntax:
  - There is no need to repeat types on the right side of the statement.
  - Angle brackets indicate that type parameters are mirrored.
- Simplifies generic declarations
- Saves typing

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

Collections

- A collection is a single object designed to manage a group of objects.
  - Objects in a collection are called *elements.*
  - *Primitives are not allowed in a collection.*
- Various collection types implement many common data structures:
  - Stack, queue, dynamic array, hash
- The Collections API relies heavily on generics for its implementation.

Collection Types



<Interface>
Collection

<Interface>
Map

<Interface>
Set

<Interface>
List

Hashset

<Interface>
SortedSet

Arraylist

LinkedList

TreeSet

Collection Interfaces and Implementation

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

`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

```
public class AutoBox {
   public static void main(String[] args){
       Integer intObject = new Integer(1);
       int intPrimitive = 2;

       Integer tempInteger;
       int tempPrimitive;

       tempInteger = new Integer(intPrimitive);
        tempPrimitive = intObject.intValue();

        tempInteger = intPrimitive; // Auto box
        tempPrimitive = intObject;  // Auto unbox
```

ArrayList Without Generics

```
public class OldStyleArrayList {
  public static void main(String args[]){
    List partList = new ArrayList(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Oops a string!");

    Iterator elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = (Integer)(elements.next()); // error?
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

Java example using syntax prior to Java 1.5

Runtime error: `ClassCastException`

Generic ArrayList

```java
public class GenericArrayList {
  public static void main(String args[]) {
    List<Integer> partList = new ArrayList<>(3);

    partList.add(new Integer(1111));
    partList.add(new Integer(2222));
    partList.add(new Integer(3333));
    partList.add("Bad Data");  // compiler error now

    Iterator<Integer> elements = partList.iterator();
    while (elements.hasNext()) {
      Integer partNumberObject = elements.next();
      int partNumber = partNumberObject.intValue();

      System.out.println("Part number: " + partNumber);
    }
  }
}
```

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.

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.

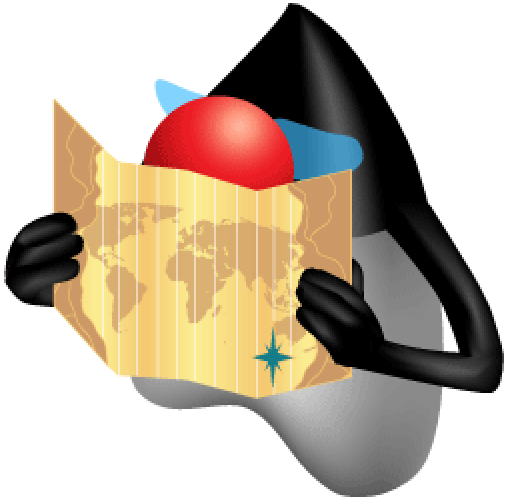
TreeSet: Implementation of Set

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

       set.add("one");
       set.add("two");
       set.add("three");
       set.add("three"); // not added, only unique

        for (String item:set){
            System.out.println("Item: " + item);
        }
    }
 }
```

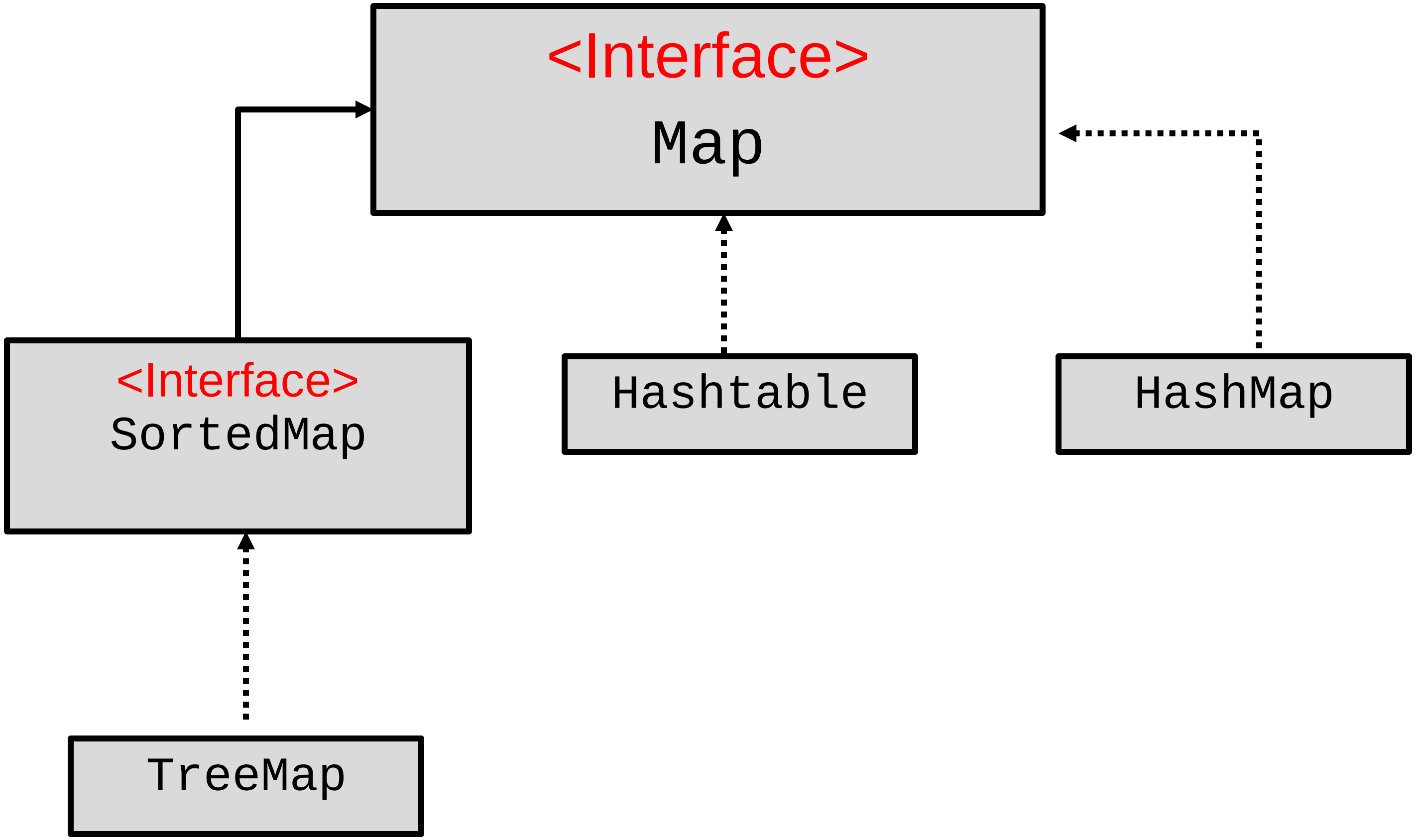
Map Interface

- 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



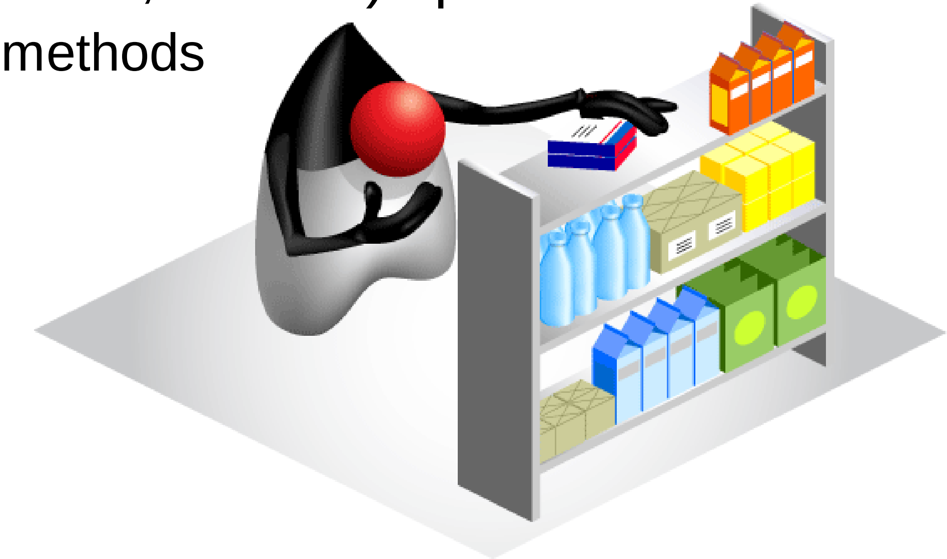
TreeMap: Implementation of Map

```
public class MapExample {
   public static void main(String[] args){
       Map <String, String> partList = new TreeMap<>();
       partList.put(“S001", "Blue Polo Shirt");
       partList.put(“S002", "Black Polo Shirt");
       partList.put(“H001", "Duke Hat");

       partList.put(“S002", "Black T-Shirt"); // Overwrite value
       Set<String> keys = partList.keySet();

        System.out.println("=== Part List ===");
        for (String key:keys){
            System.out.println("Part#: " + key + " " +
                                partList.get(key));
        }
    }
 }
```

Deque Interface

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

```java
public class TestStack {
   public static void main(String[] args){
       Deque<String> stack = new ArrayDeque<>();
       stack.push("one");
       stack.push("two");
       stack.push("three");

       int size = stack.size() - 1;
       while (size >= 0 ) {
            System.out.println(stack.pop());
            size--;
        }
    }
}
```

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

```
public class ComparableStudent implements Comparable<ComparableStudent>{
   private String name; private long id = 0; private double gpa = 0.0;

   public ComparableStudent(String name, long id, double gpa){
      // Additional code here
   }
   public String getName(){ return this.name; }
      // Additional code here

    public int compareTo(ComparableStudent s){
        int result = this.name.compareTo(s.getName());
        if (result > 0) { return 1; }
        else if (result < 0){ return -1; }
        else { return 0; }
    }
 }
```

`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);
    }
  }
}
```

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

```java
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
        }
    }
}
```

```java
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.

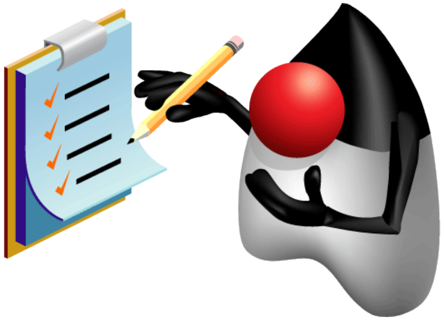
`Comparator` Test: Example

```java
public class TestComparator {
  public static void main(String[] args){
      List<Student> studentList = new ArrayList<>(3);
      Comparator<Student> sortName = new StudentSortName();
      Comparator<Student> sortGpa = new StudentSortGpa();

      // Initialize list here

      Collections.sort(studentList, sortName);
       for(Student student:studentList){
           System.out.println(student);
       }

       Collections.sort(studentList, sortGpa);
       for(Student student:studentList){
           System.out.println(student);
       }
   }
}
```

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
- Implement an `ArrayList`
- Implement a `Set`
- Implement a `HashMap`
- Implement a `Deque`
- Order collections



Practice 7-1 Overview: Counting Part Numbers by Using a `HashMap`

This practice covers the following topics:

- Creating a map to store a part number and count
- Creating a map to store a part number and description
- Processing the list of parts and producing a report

Practice 7-2 Overview: Implementing Stack by Using a Deque Object

This practice covers using the Deque object to implement a stack.

Quiz

Which of the following is *not* a conventional abbreviation for use with generics?

a. T: Table

b. E: Element

c. K: Key

d. V: Value

Quiz

Which interface would you use to create multiple sort options for a collection?

a. Comparable

b. Comparison

c. Comparator

d. Comparinator