Collection and Generic

B.Tech. (IT), Sem-6,
Applied Design Patterns and Application Frameworks (ADPAF)

Dharmsinh Desai University
Prof. H B Prajapati

Basic concepts of Data Structure

- A data structure is a container
  - It holds other data (primitive or user defined)
  - E.g., list, queue, stack, etc.
- Different types of data structures are optimized for certain types of operations.
  - Optimized for searching, sorting, …

Abstract Data Types

- Abstract Data Types (also known as ADTs) are descriptions of how a data type will work without implementation details.
- Description can be a formal, mathematical description
- In programming languages, ADT is represented by prototypes of functions.
- E.g., Java interfaces are a form of ADTs

Core operations in data structure

- A data structure should have 3 core operations.
  - A function to add data element
  - A function to remove added data element
  - A function to access data element.
- We can add other needed operations in a data structure.
- Two details are related to operations
  - Interface details
  - Implementation details

Data Structure

- A data structure is an implementation of an abstract data type.
- A data structure is created for organization of information in computer memory to allow efficient access of stored information, i.e., better algorithm efficiency.
- For example, a list is implemented using an array.

Data structures in Java

- Data structures are part of the Java Standard Library (the Collections Framework)
- Operation interfaces provided in Java interfaces
  - Description of an operation in Java interface includes
    - name of operation,
    - data type of each parameter, and
    - return type
  - Two main interfaces
    - Collection
    - Iterator
- Implementation of data structure is done in Java classes.
  - Java classes implement Java interfaces

Java Collection Framework

- Collections Framework was first introduced in Java 2 platform, Standard Edition, version 1.2.
- Single unit: The Collections Framework provides a well-designed set of interfaces and classes for storing and manipulating groups of data as a single unit, a collection.
- Provides a convenient API to many of the abstract data types
  - maps, sets, lists, trees, arrays, hashtables, etc.
- Java, being object oriented, provides data structure + algorithm
- Programmers can define higher level data abstractions
  - stacks, queues, and thread-safe collections.

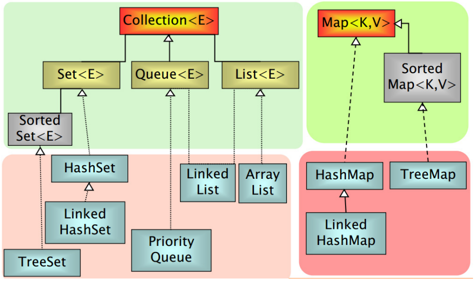
Understanding of the containers

- The Collection interface is a group of objects, with duplicates allowed.
- The Set interface extends Collection but forbids duplicates.
- The Queue interface extends Collection. Added objects have an order. Addition of object at the rear and removal from the front.
- The List interface extends Collection, allows duplicates, and introduces positional indexing.
- The Map interface extends neither Set nor Collection. It maps two things: key->value. But reverse is not defined, i.e., (value -> key)

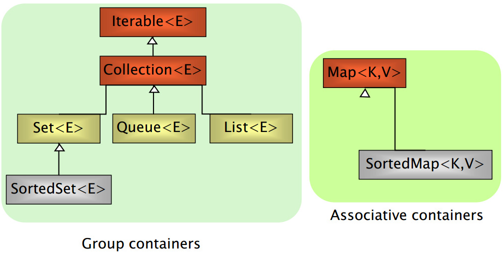
Interfaces

- Two types
  - Group containers
  - Associative containers
- There is no common interface between these two types. However there are methods that return Set views of Map objects.

Implementation



Interfaces



Collection

- Group of elements (references to objects)
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
  - T()
    - Create an empty collection
  - T(Collection c)
    - Create a collection from another collection

Widely used containers

- We will study the following container classes:
  - ArrayList
  - HashMap
  - PriorityQueue

Object as a data element of Collection

- Creating generic containers using the Object data type and polymorphism is relatively straight forward.
  - For example: private Object[] container; can hold multiple objects of any data type, as Object is the super parent.
- Problems using Object as a generic data type
  - Identify the type of object (Type checking) using instanceof operator.
  - Type cast the object to appropriate class (down casting).

ArrayList Class

- Implements the List interface and uses an array as its internal storage container
- It is a list, not an array.
- The internal array that actually stores the elements of the list is not visible outside of the ArrayList class.
- All actions on ArrayList objects are performed via the methods
- ArrayLists are generic.
  - They can hold objects of any type!

Collection before Java SE 5

- We could add any object in a collection

```
List myList = new ArrayList(10);
myList.add(“Sun");
myList.add(new Integer(108));
```

- But, when we access an item from the collection, we need to perform casting.

  ```
  String itemName = (String) myList.iterator().next();
  ```

Java Object to store any type of object

- Single parent class in Java
  - In Java, all classes can inherit from exactly one other class except Object which is at the top of the class hierarchy
- Object reference and an object of a class
  - Object reference variable can refer to an object of any Java object (Object is a super parent class)
  - It can allow polymorphism
- Thus, if the internal storage container is of type Object it can hold anything
  - Primitive values (int, float, double) are handled by *wrapping* them in wrapper objects.
  - int -> Integer, char -> Character, float -> Float, etc.

Collection before Java SE 5

- If you by mistake cast the object to wrong type, the program would successfully compile, but at runtime an exception would be thrown.
- We can use instanceof to avoid a blind cast

  ```
  Iterator li = myList.iterator();
  Object myObj = li.next();
  String item = null;
  if (myObj instanceof String) {
          item = (String) myObj;
  }
  ```

Generics

- From J2SE 5.0, Java provides compile-time type safety with the Java Collections framework through generics
- Generics allows us to specify the types of objects you want to store in a Collection at compile-time.
  - The "<>" characters are used to designate what type is to be stored.
  - If the wrong type of data is provided, a compile-time error is shown.
- Then, when we add and get items from the list, the list already knows what types of objects are supposed to be acted on.
- So we don't need to cast while accessing the object.

Example on traditional ArrayList

- We create an application GenericCollection

```
package genericcollection;

import java.util.ArrayList;
import java.util.List;
public class TraditionalCollection {
   public static void main(String[] args) {
      List list=new ArrayList();
      list.add("Sun");
      list.add(108);
      System.out.println("Value = "+(String)list.get(0));
      System.out.println("Value = "+(String)list.get(1));
   }
}
```

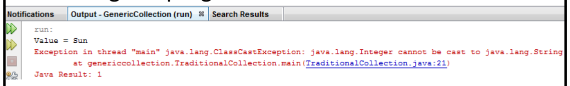
Generic collections

- From Java 5, all collection interfaces and classes have been redefined as Generics
- A generic collection can hold any object data type.
- Which type of object a particular collection will hold is specified when declaring an instance of a class that implements the Collection interface
- Use of generics lead to code that is
  - Having type safety at compile time
  - more compact
  - easier to understand
  - equally performing

Example on traditional ArrayList

Do we get any compile time error?

Running the program:



```
run:
Value = Sun
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.String
        at genericcollection.TraditionalCollection.main(TraditionalCollection.java:21)
Java Result: 1
```

We get ClassCastException, which is an error at runtime.

Methods in the Collection interface

Collection

```
public interface Collection{
      int size()
      boolean isEmpty()
      boolean contains(Object element)
      boolean containsAll(Collection c)
      boolean add(Object element)
      boolean addAll(Collection c)
      boolean remove(Object element)
      boolean removeAll(Collection c)
      void clear()
      Object[] toArray()
      Iterator iterator()
}
```

Generic Collection

```
public interface Collection<E>
{     public boolean add(E o)
            public boolean addAll(Collection<?
      extends E> c)
      public void clear()
      public boolean contains(Object o)
      public boolean containsAll(Collection<?> c)
      public boolean equals(Object o)
      public int hashCode()
      public boolean isEmpty()
      public Iterator<E> iterator()
      public boolean remove(Object o)
      public boolean removeAll(Collection<?> c)
      public boolean retainAll(Collection<?> c)
      public int size()
      public Object[] toArray()
      public <T> T[] toArray(T[] a)
}
```

Example on ArrayList (in and after J2SE 5.0)

- ArrayList<String> as a container

```
package genericcollection;

import java.util.ArrayList;
import java.util.List;

public class ArrayListString{
public static void main(String[] args) {
      List list=new ArrayList<String>();
      list.add("Sun");
      list.add(108);
      System.out.println("Value = "+list.get(0));
      System.out.println("Value = "+list.get(1));
   }
}
```

Example on ArrayList (in and after J2SE 5.0)

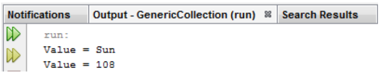
- Does the program get compiled successfully?
  - Any error? or
  - No error?

Example on ArrayList (in and after J2SE 5.0)

- Does the program get compiled successfully?
  - Any error?
  - No error?

Example on ArrayList (in and after J2SE 5.0)

- We get no error.
- The program gives following output

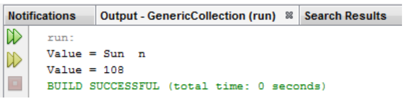


```
run:
Value = Sun
Value = 108
```

- But, why we do not get compile time error?
- What has happened?

Example on ArrayList (in and after J2SE 5.0)

- We get no error.
- The program gives following output



```
run:
Value = Sun  n
Value = 108
BUILD SUCCESSFUL (total time: 0 seconds)
```

- But, why we do not get compile time error?
- What has happened?

Example on ArrayList (in and after J2SE 5.0)

- ArrayList<Integer> as a container:

```
package genericcollection;
import java.util.ArrayList;
import java.util.List;
public class ArrayListInteger {
   public static void main(String[] args) {
      List list=new ArrayList<Integer>(5);
      list.add("Sun");
      list.add(108);
      System.out.println("Value = "+list.get(0));
      System.out.println("Value = "+list.get(1));
   }
}
```

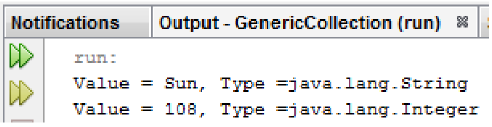
Let's see both the programs again

- ArrayList<Integer> as a container:

```
package genericcollection;
import java.util.ArrayList;
import java.util.List;
public class ArrayListInteger {
   public static void main(String[] args) {
      List list=new ArrayList<Integer>(5);
      list.add("Sun");
      list.add(108);
      System.out.println("Value = "+list.get(0)+", Type
     ="+list.get(0).getClass().getName());
      System.out.println("Value = "+list.get(1)+", Type
     ="+list.get(1).getClass().getName());
   }
}
```

Let’s see both the programs again

- ArrayList<Integer> as a container:
- Output



```
run:
Value = Sun, Type =java.lang.String
Value = 108, Type =java.lang.Integer
```

How to avoid non compatible object gets added in a specific type of container?

- ArrayList<Integer> as a container:

```
package genericcollection;
import java.util.ArrayList;
public class CorrectArralyListInteger {
   public static void main(String[] args) {
      ArrayList<Integer> list=new ArrayList<Integer>(5);
      list.add("Sun");
      list.add(108);
      System.out.println("Value = "+list.get(0)+", Type
     ="+list.get(0).getClass().getName());
      System.out.println("Value = "+list.get(1)+", Type
     ="+list.get(1).getClass().getName());
   }

}
```

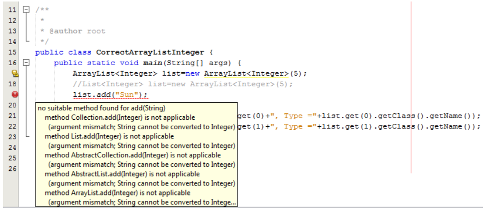
Let’s see both the programs again

- ArrayList<String> as a container:

```
package genericcollection;

import java.util.ArrayList;
import java.util.List;
public class ArrayListString {
public static void main(String[] args) {
      List list=new ArrayList<String>();
      list.add("Sun");
      list.add(108);
System.out.println("Value = "+list.get(0)+", Type ="+list.get(0).getClass().getName());
      System.out.println("Value = "+list.get(1)+", Type
     ="+list.get(1).getClass().getName());
   }
}
```

How to avoid non compatible object gets added in a specific type of container?

- ArrayList<Integer> as a container:

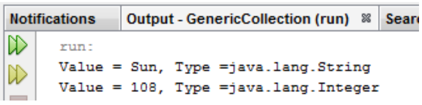


```
/**
 *
 * @author root
 */
public class CorrectArrayListInteger {
    public static void main(String[] args) {
        ArrayList<Integer> list=new ArrayList<Integer>(5);
        //List<Integer> list=new ArrayList<Integer>(5);
        list.add("Sun");
get(0)+", Type ="+list.get(0).getClass().getName());
get(1)+", Type ="+list.get(1).getClass().getName());
```

```
no suitable method found for add(String)
  method Collection.add(Integer) is not applicable
    (argument mismatch; String cannot be converted to Integer)
  method List.add(Integer) is not applicable
    (argument mismatch; String cannot be converted to Integer)
  method AbstractCollection.add(Integer) is not applicable
    (argument mismatch; String cannot be converted to Integer)
  method AbstractList.add(Integer) is not applicable
    (argument mismatch; String cannot be converted to Integer)
  method ArrayList.add(Integer) is not applicable
    (argument mismatch; String cannot be converted to Intege...
```

Let’s see both the programs again

- ArrayList<String> as a container:
- Output



```
run:
Value = Sun, Type =java.lang.String
Value = 108, Type =java.lang.Integer
```

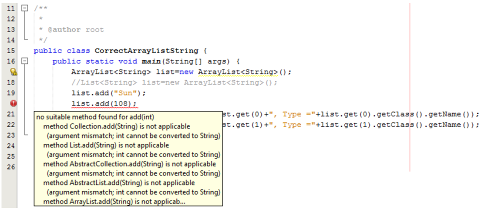
How to avoid non compatible object gets added in a specific type of container?

- ArrayList<String> as a container:

```
package genericcollection;
import java.util.ArrayList;
public class CorrectArralyListString {
   public static void main(String[] args) {
      ArrayList<String> list=new ArrayList<String>();
      list.add("Sun");
      list.add(108);
System.out.println("Value = "+list.get(0)+", Type
     ="+list.get(0).getClass().getName());
      System.out.println("Value = "+list.get(1)+", Type
     ="+list.get(1).getClass().getName());
   }

}
```

How to avoid non compatible object gets added in a specific type of container?

- ArrayList<String> as a container:



```java
/**
 *
 * @author root
 */
public class CorrectArrayListString {
    public static void main(String[] args) {
        ArrayList<String> list=new ArrayList<String>();
        //List<String> list=new ArrayList<String>();
        list.add("Sun");
        list.add(108);
```

```
no suitable method found for add(int)
    method Collection.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method List.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method AbstractCollection.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method AbstractList.add(String) is not applicable
      (argument mismatch; int cannot be converted to String)
    method ArrayList.add(String) is not applicab...
```

```java
st.get(0)+", Type ="+list.get(0).getClass().getName());
st.get(1)+", Type ="+list.get(1).getClass().getName());
```

How to avoid non compatible object gets added in a specific type of container?

- In both the programs, we explicitly reference specific container using a reference of the specific type
- Example 1:

  ArrayList<String> list=new ArrayList<String>();
  Instead of
  List list=new ArrayList<String>();
- Example 2:

  ArrayList<Integer> list=new ArrayList<Integer>(5);
  Instead of
  List list=new ArrayList<Integer>(5);

How to declare a specific type of container?

- In both the programs, we can drop data type in the constructors:
- Example 1:

  ArrayList<String> list=new ArrayList<>();
  Instead of
  ArrayList<String> list=new ArrayList<String>();
- Example 2:

  ArrayList<Integer> list=new ArrayList<>(5);
  Instead of
  ArrayList<Integer> list=new ArrayList<Integer>(5);

How to declare a specific type of container, but reference is of generic type?

- Suppose we want reference is of generalized type (List) and still the reference can point of objects of specialized type (ArrayList or LinkedList)?
- Example 1:

  List<String> list=new ArrayList<>();
  Instead of
  ArrayList<String> list=new ArrayList<>();
- Example 2:

  List<Integer> list=new ArrayList<>(5);
  Instead of
  ArrayList<Integer> list=new ArrayList<>(5);

Map

- A container that associates keys to values
  - E.g., Adhar ID → Person
  - Student ID → Student
- What are keys and values?
  - Keys and values must be objects (Not variables of primitive data type, e.g., int, float, double, etc.)
  - Keys must be unique
  - Only one value per key
    - E.g., one Student ID (key) cannot be assigned to two students (value)
- Following constructors are common to all collection implementers
  - T()
    - Creates an empty map
  - T(Map m)
    - Creates a map from another map

Map interface (Traditional)

- Traditional Map has the following major operations
  - Object put(Object key, Object value)
  - Object get(Object key)
  - Object remove(Object key)
  - boolean containsKey(Object key)
  - boolean containsValue(Object value)
  - public Set keySet()
  - public Collection values()
  - int size()
  - boolean isEmpty()
  - void clear()
- In Generic Map (J2SE 5.0), instead of Object type is Template (e.g., E)

Example on HashMap

```
package genericcollection;
import java.util.HashMap;
import java.util.Map;
public class MapExample {
   public static void main(String[] args) {
      String studentName;
      Integer id;
      Map<Integer, String> studentMap=new HashMap<>();
      studentMap.put(1, "Kisan");
      studentMap.put(2, "Radha");
      studentMap.put(3, "Ganga");
```

Can we invoke a method on the returned object?

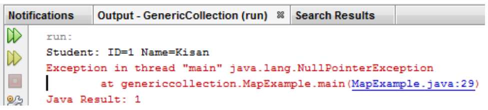
```
package genericcollection;
import java.util.HashMap;
import java.util.Map;
public class MapExample {
   public static void main(String[] args) {
      String studentName;
      Integer id;
      Map<Integer, String> studentMap=new HashMap<>();
      studentMap.put(1, "Kisan");
      studentMap.put(2, "Radha");
      studentMap.put(3, "Ganga");
      id=1;
      studentName=studentMap.get(id);
      System.out.println("Student: ID="+id+" Name="+studentName);
      id=4;
      studentName=studentMap.get(id);
      System.out.println("Student: ID="+id+" Name="+studentName.toUpperCase());
   }
}
```

Example on HashMap

```
       id=1;
      studentName=studentMap.get(id);
      System.out.println("Student: ID="+id+" Name="+studentName);
      id=4;
      studentName=studentMap.get(id);
      System.out.println("Student: ID="+id+" Name="+studentName);
   }
}
```

Can we invoke a method on the returned object?

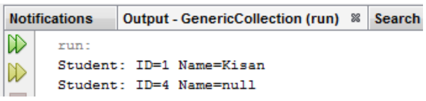
- No compile time error.
- But, at runtime, we get the following:



```
run:
Student: ID=1 Name=Kisan
Exception in thread "main" java.lang.NullPointerException
        at genericcollection.MapExample.main(MapExample.java:29)
Java Result: 1
```

Example on HashMap

- Output



```
run:
Student: ID=1 Name=Kisan
Student: ID=4 Name=null
```

Always check for returned object

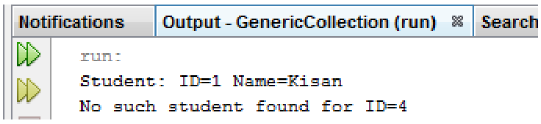
```
package genericcollection;
import java.util.HashMap;
import java.util.Map;
public class CorrectMapExample {
   public static void main(String[] args) {
      String studentName;
      Integer id;
      Map<Integer, String> studentMap=new HashMap<>();
      studentMap.put(1, "Kisan");
      studentMap.put(2, "Radha");
      studentMap.put(3, "Ganga");
```

Always check for returned object

```
id=1;
studentName=studentMap.get(id);
System.out.println("Student: ID="+id+" Name="+studentName);
id=4;
studentName=studentMap.get(id);
if(studentName!=null)
   System.out.println("Student: ID="+id+"
Name="+studentName.toUpperCase());
else
   System.out.println("No such student found for ID="+id);
  }

}
```

Always check for returned object

Running the program



```
run:
Student: ID=1 Name=Kisan
No such student found for ID=4
```

Queue

- Collection whose elements have an order
- Defines a head position where is the first element that can be accessed
- Important methods
  - add() (adds element into the queue)
  - peek() (Retrieves element, but does not remove)
  - poll() (No exception if element is absent)
  - remove() (Throws exception if element is absent)

Queue Implementations

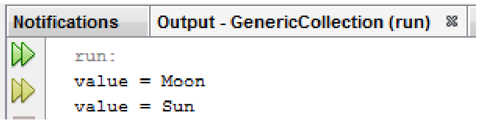
- LinkedList
  - head is the first element of the list
  - FIFO: Fist-In-First-Out
- PriorityQueue
  - head is the smallest element

Example on PriorityQueue

```
package genericcollection;
import java.util.PriorityQueue;
public class QueueTest {
   public static void main(String[] args) {
      PriorityQueue<String> queue=new PriorityQueue();
      queue.add("Sun");
      queue.add("Moon");
      System.out.println("value = "+queue.remove());
      System.out.println("value = "+queue.remove());
   }
}
```

Example on PriorityQueue

- Running the program



```
run:
value = Moon
value = Sun
```

Retrieving objects using Iterators

- A common operation with collections is to iterate over their elements
- Interface Iterator provides a transparent means to cycle through all elements of a Collection
- Keeps track of last visited element of the related collection
- Each time querying the current element, it moves on automatically.

Example on Iteration

```
package genericcollection;

public class Student {
   private int studentId;
   private String studentName;

   public int getStudentId() {
      return studentId;
   }

   public void setStudentId(int
    studentId) {
      this.studentId = studentId;
   }
```

```
public String getStudentName() {
      return studentName;
   }

   public void setStudentName(String
    studentName) {
      this.studentName =
    studentName;
   }
   public Student(int id, String name){
      studentId=id;
      studentName=name;
   }
}
```

How we get Iterator from a List

```
public interface List<E>{
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E>{
    E next();
    boolean hasNext();
    void remove();
}
```

- The remove() method is optionally supported by the underlying collection. When called, and supported, the element returned by the last next() call is removed.

Example on Iteration

```
package genericcollection;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class StudentIteration {
   public static void main(String[] args) {
      List<Student> studentList=new
     ArrayList<>(5);
      studentList.add(new
     Student(1,"Kisan"));
      studentList.add(new
     Student(2,"Radha"));
      studentList.add(new
     Student(3,"Ganga"));
      studentList.add(new
     Student(4,"Narmada"));
```

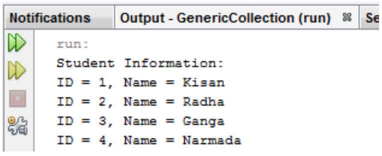
```
Iterator<Student>
     iterator=studentList.iterator();
      Student student;
      System.out.println("Student
     Information:");
      while(iterator.hasNext()){
         student=iterator.next();
         System.out.println("ID =
     "+student.getStudentId()+", Name =
     "+student.getStudentName());
      }
   }
}
```

Example on Iteration

- We store Student object in a collection
- We retrieve it using Iterator

Example on Iteration

- Running the program



```
run:
Student Information:
ID = 1, Name = Kisan
ID = 2, Name = Radha
ID = 3, Name = Ganga
ID = 4, Name = Narmada
```

Example on Iteration using foreach loop

- If we use foreach loop, we do not need to get iterator explicitly
- Syntax
  - If we have object of type X stored in a Collection of type Xlist, then we can write foreach loop as follows:

```
for(X xobj: XListObj){
   //access xobj
}
```

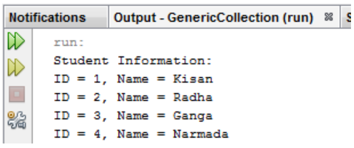
Example on Iteration using foreach loop

```
package genericcollection;

import java.util.ArrayList;
import java.util.List;
public class StudentIteration {
   public static void main(String[] args) {
      List<Student> studentList=new
     ArrayList<>(5);
      studentList.add(new
     Student(1,"Kisan"));
      studentList.add(new
     Student(2,"Radha"));
      studentList.add(new
     Student(3,"Ganga"));
      studentList.add(new
     Student(4,"Narmada"));
```

```
System.out.println("Student
     Information:");
      for(Student student:studentList){
         System.out.println("ID =
     "+student.getStudentId()+", Name =
     "+student.getStudentName());
      }
   }
}
```

Example on Iteration using foreach loop

- We get the same output



```
run:
Student Information:
ID = 1, Name = Kisan
ID = 2, Name = Radha
ID = 3, Name = Ganga
ID = 4, Name = Narmada
```

Iterating a Map

- We create a Map object of the following mapping
  - studentId (Integer) -> studentName (String)
- We can get all values (studentNames) using
  - map.values()

Example on Iterating a Map

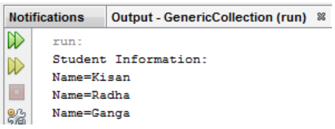
```
package genericcollection;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
public class StudentMapIteration {
   public static void main(String[] args) {
      Map<Integer, String> studentMap=new HashMap<>();
      studentMap.put(1, "Kisan");
      studentMap.put(2, "Radha");
      studentMap.put(3, "Ganga");
```

Example on Iterating a Map

```
      Collection<String> students=studentMap.values();
      System.out.println("Student Information:");
      for(String stName:students){
         System.out.println("Name="+stName);
      }
   }
}
```

Example on Iterating a Map

- We get the following output



```
run:
Student Information:
Name=Kisan
Name=Radha
Name=Ganga
```

Iterating a Map using Key

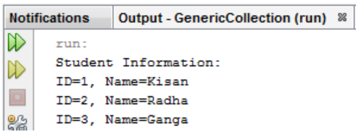
```
Collection<Integer> stKeys=studentMap.keySet();
      String stName;
      System.out.println("Student Information:");
      for(Integer stKey:stKeys){
         stName=studentMap.get(stKey);
         System.out.println("ID="+stKey+", Name="+stName);
      }
   }
}
```

Iterating a Map using Key

- Suppose, we do not know keys?
- We create a Map object of the following mapping
  - studentId (Integer) -> studentName (String)
- We can get all keys (studentId) using
  - map.keySet()

Iterating a Map using Key

- Running the program



```
run:
Student Information:
ID=1, Name=Kisan
ID=2, Name=Radha
ID=3, Name=Ganga
```

Iterating a Map using Key

```
package genericcollection;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
public class StudentKeyIteration {
   public static void main(String[] args) {
      Map<Integer, String> studentMap=new HashMap<>();
      studentMap.put(1, "Kisan");
      studentMap.put(2, "Radha");
      studentMap.put(3, "Ganga");
```

Storing diverse objects in Generic Collection

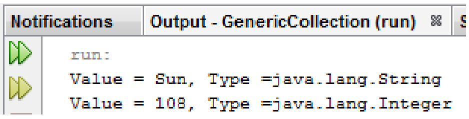
- We want to store diverse objects in a collection object created using generic feature.

Example: Storing diverse objects in Generic Collection

```
package genericcollection;

import java.util.ArrayList;
import java.util.List;
public class DifferentObjsInGeneric {
    public static void main(String[] args) {
        List<Object> list=new ArrayList<>();
        list.add("Sun");
        list.add(108);
        System.out.println("Value = "+list.get(0)+", Type
    ="+list.get(0).getClass().getName());
        System.out.println("Value = "+list.get(1)+", Type
    ="+list.get(1).getClass().getName());
    }
}
```

Example: Storing diverse objects in Generic Collection

- Output



```
run:
Value = Sun, Type =java.lang.String
Value = 108, Type =java.lang.Integer
```

References

- CS 307 Fundamentals of Computer Science, Topic 12 ADTS, Data Structures, Java Collections and Generic Data Structures
- Java Collection Framework, SoftEng, March 2009, http://softeng.polito.it
- Java Generics by Billy B. L. Lim