PROGRAMMING IN JAVA

Lecture 11

Generics and the Java Collections Framework

Generic Classes and Methods, Lists, Sets, Maps, Queues, Iterators, Wildcards, Sorting, and Collection Algorithms

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain why generics are used in Java;
2. define and use generic classes;
3. define and use generic methods;
4. explain type parameters such as `T`, `E`, `K`, and `V`;
5. distinguish raw types from parameterized types;
6. use bounded type parameters;
7. use wildcard types;
8. explain the purpose of the Java Collections Framework;
9. use `List`, `Set`, `Map`, and `Queue`;
10. distinguish `ArrayList` from `LinkedList`;
11. use `HashSet` for unique values;
12. use `HashMap` for key-value storage;
13. traverse collections using enhanced for-loops and iterators;
14. sort objects using `Comparable` and `Comparator`;
15. choose an appropriate collection for a practical problem.

2. Why Collections Are Needed

Arrays are useful, but they have limitations.

An array has a fixed size:

```
int[] numbers = new int[5];
```

Once created, its length cannot change.

In many real applications we need collections that can:

- grow dynamically;
- remove elements;
- search efficiently;
- store unique values;
- map keys to values;
- process elements in insertion or priority order.

The Java Collections Framework provides standard interfaces and classes for these tasks.

3. What Is the Java Collections Framework?

The **Java Collections Framework** is a set of interfaces, classes, and algorithms used to store and process groups of objects.

Important interfaces include:

- `Collection<E>`
- `List<E>`
- `Set<E>`
- `Queue<E>`
- `Deque<E>`
- `Map<K,V>`

Some important implementation classes are:

- `ArrayList<E>`
- `LinkedList<E>`
- `HashSet<E>`
- `TreeSet<E>`

- `HashMap<K,V>`
- `TreeMap<K,V>`
- `PriorityQueue<E>`
- `ArrayDeque<E>`

4. Why Generics Are Important

Before generics, collections stored objects as type `Object`.

This required explicit casting.

Example conceptually:

```
Object value = "Java";
String text = (String) value;
```

Generics allow the programmer to specify the expected element type.

```
List<String> names =
        new ArrayList<>();
```

Now only strings may be inserted through the generic API.

Generics provide:

- compile-time type safety;
- less explicit casting;
- clearer APIs;
- reusable algorithms;
- earlier detection of type errors.

5. Generic Type Parameters

Common type-parameter names include:

Symbol	Common meaning
`T`	Type
`E`	Element
`K`	Key
`V`	Value
`N`	Number

These are conventions rather than mandatory names.

6. Generic Classes

A generic class contains one or more type parameters.

General form:

```
class Box<T> {

    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

The same class can then work with different types.

Example 1. Generic Box

```
1  public class Main {
2
3      static class Box<T> {
4
5          private T value;
6
7          public void set(T value) {
8              this.value = value;
9          }
10
```

Example 1 (continued)

```
11          public T get() {
12              return value;
13          }
14      }
15
16      public static void main(String[] args) {
17
18          Box<Integer> numberBox =
19                  new Box<>();
20
21          numberBox.set(100);
22
23          Box<String> textBox =
24                  new Box<>();
25
26          textBox.set("Java");
27
28          System.out.println(
29                  numberBox.get()
30          );
31
32          System.out.println(
33                  textBox.get()
34          );
35      }
36  }
```

Expected output

```
100
Java
```

7. Generic Classes with Several Type Parameters

A class may have several type parameters.

```
class Pair<K, V> {

    private K key;
    private V value;
}
```

This pattern is common in key-value structures.

Example 2. Generic Pair

```
1  public class Main {
2
3      static class Pair<K, V> {
4
5          private K key;
6          private V value;
7
8          public Pair(
9                  K key,
10                 V value) {
11
12             this.key = key;
13             this.value = value;
14         }
15
16         public K getKey() {
17             return key;
18         }
19
20         public V getValue() {
21             return value;
22         }
23     }
24
25     public static void main(String[] args) {
26
27         Pair<Integer, String> student =
28                 new Pair<>(
29                         1001,
30                         "Aruzhan"
31                 );
```

Example 2 (continued)

```
32
33          System.out.println(
34                  student.getKey()
35          );
36
37          System.out.println(
38                  student.getValue()
39          );
40      }
41  }
```

Expected output

```
1001
Aruzhan
```

8. Generic Methods

A method can declare its own type parameter.

General form:

```
public static <T> void print(T value) {
    System.out.println(value);
}
```

The type parameter appears before the return type.

Example 3. Generic Method

```
1  public class Main {
2
3      public static <T>
4      void display(T value) {
5
6          System.out.println(value);
7      }
8
9      public static void main(String[] args) {
10
11         display(10);
12         display(3.14);
13         display("Java");
14         display(true);
15     }
16 }
```

Expected output

```
10
3.14
Java
true
```

9. Bounded Type Parameters

Sometimes a generic type should be restricted.

Example:

```
class Calculator<T extends Number> {
}
```

Now `T` must be `Number` or one of its subclasses.

Examples include:

- `Integer`
- `Double`
- `Long`
- `Float`

Example 4. Bounded Generic Method

```
1  public class Main {
2
3      public static <T extends Number>
4      double square(T value) {
5
6          double x =
7                  value.doubleValue();
8
9          return x * x;
10     }
11
12     public static void main(String[] args) {
13
14         System.out.println(
15                 square(5)
16         );
17
18         System.out.println(
19                 square(2.5)
20         );
21     }
22 }
```

Expected output

```
25.0
6.25
```

10. Raw Types

This code uses a raw type:

```
Box box = new Box();
```

It omits the type argument.

Raw types exist mainly for compatibility with older Java code.

They reduce type safety and should generally be avoided.

Prefer:

```
Box<String> box =
        new Box<>();
```

11. Wildcards

A wildcard is written using:

```
?
```

Example:

```
List<?> items
```

This means a list of some unknown type.

Wildcards are useful when writing flexible APIs.

12. Upper-Bounded Wildcards

Example:

```
List<? extends Number>
```

This may refer to:

- `List<Integer>`
- `List<Double>`
- `List<Long>`

It is useful when the method mainly reads numeric values.

Example 5. Sum with Wildcards

```
import java.util.List;

public class Main {

    public static double sum(
            List<? extends Number>
            values) {

        double result = 0.0;

        for (Number value : values) {
            result +=
                value.doubleValue();
        }

        return result;
    }

    public static void main(String[] args) {

        List<Integer> integers =
                List.of(1, 2, 3);

        List<Double> doubles =
                List.of(1.5, 2.5);

        System.out.println(
                sum(integers)
        );
```

Example 5 (continued)

```
30
31          System.out.println(
32                  sum(doubles)
33          );
34      }
35  }
```

Expected output

```
6.0
4.0
```

13. Lower-Bounded Wildcards

Example:

```
List<? super Integer>
```

This may refer to a list whose element type is `Integer` or one of its supertypes.

Lower bounds are useful when a method mainly writes values into a generic structure.

A commonly used guideline is:

Producer extends, consumer super.

Often abbreviated as PECS.

14. The `List` Interface

A `List<E>`:

- preserves element order;
- allows duplicate elements;
- supports indexed access.

Common implementations:

- `ArrayList<E>`

- `LinkedList<E>`

15. ArrayList

`ArrayList` is backed conceptually by a resizable array.

It is usually a good default list implementation.

Common operations:

```
list.add(value);
list.get(index);
list.set(index, value);
list.remove(index);
list.size();
list.contains(value);
```

Example 6. Working with ArrayList

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

public class Main {

    public static void main(String[] args) {

        List<String> students =
                new ArrayList<>();

        students.add("Aruzhan");
        students.add("Dias");
        students.add("Ali");

        System.out.println(
                students
        );

        students.remove("Dias");

        students.add("Dana");

        for (String student :
                students) {

            System.out.println(
                    student
            );
        }
    }
}
```

Expected output

```
[Aruzhan, Dias, Ali]
Aruzhan
Ali
Dana
```

16. `ArrayList` vs `LinkedList`

Property	ArrayList	LinkedList
Indexed access	fast	slower
Append	usually fast	fast
Insert/remove in middle	may shift elements	link changes but traversal required
Memory overhead	lower	higher
Typical default choice	Yes	only when linked-list behavior is useful

For most general-purpose list tasks, `ArrayList` is the usual starting choice.

17. The Set Interface

A `Set<E>` stores unique elements.

Duplicates are not retained according to the equality semantics of the set.

Common implementations:

- `HashSet<E>`
- `LinkedHashSet<E>`
- `TreeSet<E>`

Example 7. Removing Duplicates with `HashSet`

```
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) {

        Set<Integer> numbers =
                new HashSet<>();

        numbers.add(10);
        numbers.add(20);
        numbers.add(10);
        numbers.add(30);

        System.out.println(
                numbers
        );
    }
}
```

The exact order is not guaranteed.

One possible output is:

Illustration

```
[20, 10, 30]
```

The value `10` appears only once.

18. `HashSet`, `LinkedHashSet`, and `TreeSet`

Implementation	Main characteristic
`HashSet`	unique elements, no iteration order guarantee
`LinkedHashSet`	unique elements, preserves insertion order
`TreeSet`	unique elements, sorted order

19. The `Map` Interface

A `Map<K,V>` stores key-value pairs.

Each key maps to a value.

Typical operations:

```
map.put(key, value);
map.get(key);
map.remove(key);
map.containsKey(key);
map.size();
```

Common implementations:

- `HashMap<K,V>`
- `LinkedHashMap<K,V>`
- `TreeMap<K,V>`

Note that `Map` is part of the Collections Framework but does not extend `Collection`.

Example 8. HashMap

```
import java.util.HashMap;
import java.util.Map;

public class Main {

    public static void main(String[] args) {

        Map<Integer, String> students =
                new HashMap<>();

        students.put(1001, "Aruzhan");
        students.put(1002, "Dias");
        students.put(1003, "Ali");

        System.out.println(
                students.get(1002)
        );

        System.out.println(
                students.containsKey(
                        1001
                )
        );
    }
}
```

Expected output

```
Dias
true
```

20. Iterating Through a Map

A useful approach is `entrySet()`.

```
for (Map.Entry<Integer, String>
        entry : students.entrySet()) {

    System.out.println(
            entry.getKey() +
            " -> " +
```

```
            entry.getValue()
    );
}
```

Example 9. Word Frequency Counter

```
import java.util.HashMap;
import java.util.Map;

public class Main {

    public static void main(String[] args) {

        String[] words = {
            "java",
            "code",
            "java",
            "class",
            "code",
            "java"
        };

        Map<String, Integer> frequency =
                new HashMap<>();

        for (String word : words) {

            frequency.put(
                word,
                frequency.getOrDefault(
                    word, 0
                ) + 1
            );
        }

        System.out.println(
                frequency
        );
    }
}
```

One possible output is:

Illustration

```
{java=3, code=2, class=1}
```

21. Queues

A queue commonly processes elements in first-in, first-out order.

Important methods include:

- `offer()`
- `poll()`
- `peek()`

A common queue implementation is `LinkedList`.

Example 10. Queue

```
1  import java.util.LinkedList;
2  import java.util.Queue;
3
4  public class Main {
5
6      public static void main(String[] args) {
7
8          Queue<String> queue =
9                  new LinkedList<>();
10
11         queue.offer("A");
12         queue.offer("B");
13         queue.offer("C");
14
15         System.out.println(
16                 queue.poll()
17         );
18
19         System.out.println(
20                 queue.peek()
21         );
22
23         System.out.println(
24                 queue
25         );
26     }
27 }
```

Expected output

```
A
B
```

```
[B, C]
```

22. Deque and Stack Behavior

A `Deque<E>` supports insertion and removal at both ends.

`ArrayDeque<E>` is often preferred for stack or queue behavior.

For stack-style operations:

```
stack.push(value);
stack.pop();
stack.peek();
```

Example 11. Stack with `ArrayDeque`

```
1  import java.util.ArrayDeque;
2  import java.util.Deque;
3
4  public class Main {
5
6      public static void main(String[] args) {
7
8          Deque<String> stack =
9                  new ArrayDeque<>();
10
11         stack.push("First");
12         stack.push("Second");
13         stack.push("Third");
14
15         System.out.println(
16                 stack.pop()
17         );
18
19         System.out.println(
20                 stack.peek()
21         );
22     }
23 }
```

Expected output

```
Third
Second
```

23. Priority Queues

A `PriorityQueue<E>` retrieves elements according to priority rather than strict insertion order.

By default, natural ordering is used.

Example 12. Priority Queue

```
1  import java.util.PriorityQueue;
2  import java.util.Queue;
3
4  public class Main {
5
6      public static void main(String[] args) {
7
8          Queue<Integer> queue =
9                  new PriorityQueue<>();
10
11         queue.offer(40);
12         queue.offer(10);
13         queue.offer(30);
14         queue.offer(20);
15
16         while (!queue.isEmpty()) {
17
18             System.out.println(
19                     queue.poll()
20             );
21         }
22     }
23 }
```

Expected output

```
10
20
30
40
```

24. Iterators

An `Iterator<E>` traverses a collection element by element.

Important methods:

- `hasNext()`
- `next()`
- `remove()`

Example:

```
Iterator<String> iterator =
        names.iterator();

while (iterator.hasNext()) {
    System.out.println(
            iterator.next()
    );
}
```

Example 13. Safe Removal with an Iterator

```
1  import java.util.ArrayList;
2  import java.util.Iterator;
3  import java.util.List;
4
5  public class Main {
6
7      public static void main(String[] args) {
8
9          List<Integer> numbers =
10                 new ArrayList<>(
11                     List.of(
12                         1, 2, 3, 4, 5, 6
13                     )
14                 );
15
16         Iterator<Integer> iterator =
17                 numbers.iterator();
18
19         while (iterator.hasNext()) {
20
21             int value =
```

Example 13 (continued)

```
                    iterator.next();

            if (value % 2 == 0) {
                iterator.remove();
            }
        }

        System.out.println(
                numbers
        );
    }
}
```

Expected output

```
[1, 3, 5]
```

25. Why Not Remove Directly During Enhanced For-Loop?

Modifying some collections structurally while iterating using an enhanced for-loop may cause:

Illustration

`ConcurrentModificationException`

An iterator's `remove()` method is one correct approach for certain in-place removals.

26. Sorting Collections

Lists can be sorted using:

```
Collections.sort(list);
```

or:

```
list.sort(comparator);
```

Objects may define natural ordering with `Comparable`.

Alternative orderings can be defined with `Comparator`.

Example 14. Sorting Students

```
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;

public class Main {

    static class Student {

        private String name;
        private double gpa;

        public Student(
                String name,
                double gpa) {

            this.name = name;
            this.gpa = gpa;
        }

        public String getName() {
            return name;
        }

        public double getGpa() {
            return gpa;
        }

        @Override
        public String toString() {
```

Example 14 (continued)

```
30              return name +
31                      " (" + gpa + ")";
32          }
33      }
34
35      public static void main(String[] args) {
36
37          List<Student> students =
38                  new ArrayList<>();
39
40          students.add(
41              new Student("Dias", 3.5)
42          );
43
44          students.add(
45              new Student("Aruzhan", 3.9)
46          );
47
48          students.add(
49              new Student("Ali", 3.7)
50          );
51
52          students.sort(
53              Comparator.comparingDouble(
54                  Student::getGpa
55              ).reversed()
56          );
57
58          for (Student s : students) {
59              System.out.println(s);
60          }
61      }
62  }
```

Expected output

```
Aruzhan (3.9)
Ali (3.7)
Dias (3.5)
```

27. Useful Collection Algorithms

The `Collections` utility class provides methods such as:

- `sort()`
- `reverse()`
- `shuffle()`
- `min()`
- `max()`
- `frequency()`
- `binarySearch()`

Example 15. Collections Utility Methods

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<Integer> values =
                new ArrayList<>(
                    List.of(
                        5, 2, 8, 2, 9
                    )
                );

        Collections.sort(values);

        System.out.println(values);

        System.out.println(
                "Min = " +
                Collections.min(values)
        );

        System.out.println(
                "Max = " +
                Collections.max(values)
        );

        System.out.println(
```

Example 15 (continued)

```
                "Frequency of 2 = " +
                Collections.frequency(
                    values, 2
                )
        );
    }
}
```

Expected output

```
[2, 2, 5, 8, 9]
Min = 2
Max = 9
Frequency of 2 = 2
```

28. Immutable Collection Factories

Modern Java provides convenient factory methods:

```
List.of(...)
Set.of(...)
Map.of(...)
```

The returned collections are unmodifiable.

Example:

```
List<String> names =
        List.of("A", "B", "C");
```

Attempting structural modification causes an exception.

If a modifiable copy is needed:

```
List<String> modifiable =
        new ArrayList<>(names);
```

29. Equality and Hash-Based Collections

Hash-based collections such as `HashSet` and `HashMap` rely heavily on:

- `equals()`
- `hashCode()`

If custom objects are used as keys or set elements, these methods should be implemented consistently when logical equality is needed.

Rule:

> If two objects are equal according to `equals()`, they must have the same `hashCode()`.

30. Choosing the Right Collection

Need	Typical choice
Ordered sequence with duplicates	`ArrayList`
Frequent double-ended operations	`ArrayDeque`
Unique elements	`HashSet`
Unique elements preserving insertion order	`LinkedHashSet`
Sorted unique elements	`TreeSet`
Key-value lookup	`HashMap`
Key-value insertion order	`LinkedHashMap`
Sorted keys	`TreeMap`
FIFO processing	`Queue` / `ArrayDeque`
Priority-based processing	`PriorityQueue`

Example 16. In-Memory Student Repository

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  public class Main {
5
6      static class Student {
7
8          private int id;
9          private String name;
```

Example 16 (continued)

```
10          private double gpa;
11
12          public Student(
13                  int id,
14                  String name,
15                  double gpa) {
16
17              this.id = id;
18              this.name = name;
19              this.gpa = gpa;
20          }
21
22          public int getId() {
23              return id;
24          }
25
26          @Override
27          public String toString() {
28              return id + " " +
29                      name + " " +
30                      gpa;
31          }
32      }
33
34      static class StudentRepository {
35
36          private Map<Integer, Student>
37                  data =
38                  new HashMap<>();
39
40          public void add(Student student) {
41
42              data.put(
43                  student.getId(),
44                  student
45              );
46          }
47
48          public Student findById(int id) {
49
50              return data.get(id);
51          }
52
53          public void remove(int id) {
54
```

Example 16 (continued)

```
            data.remove(id);
        }

        public void displayAll() {

            for (Student student :
                    data.values()) {

                System.out.println(
                        student
                );
            }
        }
    }

    public static void main(String[] args) {

        StudentRepository repository =
                new StudentRepository();

        repository.add(
            new Student(
                1, "Aruzhan", 3.9
            )
        );

        repository.add(
            new Student(
                2, "Dias", 3.5
            )
        );

        repository.displayAll();

        System.out.println(
            "Found: " +
            repository.findById(1)
        );
    }
}
```

One possible output:

Illustration

```
1 Aruzhan 3.9
2 Dias 3.5
Found: 1 Aruzhan 3.9
```

Example 17. Grouping Students by Course

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Main {

    public static void main(String[] args) {

        Map<String, List<String>>
                groups =
                new HashMap<>();

        groups.computeIfAbsent(
            "Java",
            key -> new ArrayList<>()
        ).add("Aruzhan");

        groups.computeIfAbsent(
            "Java",
            key -> new ArrayList<>()
        ).add("Dias");

        groups.computeIfAbsent(
            "Databases",
            key -> new ArrayList<>()
        ).add("Ali");

        System.out.println(groups);
    }
}
```

One possible output:

Illustration

```
{Java=[Aruzhan, Dias], Databases=[Ali]}
```

31. Common Mistakes

31.1 Mistake 1: Using Raw Types

Avoid:

```
List list = new ArrayList();
```

Prefer:

```
List<String> list =
        new ArrayList<>();
```

31.2 Mistake 2: Choosing a Collection by Habit

Do not always use `ArrayList` or `HashMap`. Choose according to the required behavior.

31.3 Mistake 3: Assuming `HashSet` or `HashMap` Has Stable Order

Hash-based implementations do not guarantee insertion order.

31.4 Mistake 4: Modifying an Unmodifiable Collection

Collections returned by `List.of()`, `Set.of()`, and `Map.of()` cannot be modified structurally.

31.5 Mistake 5: Inconsistent `equals()` and `hashCode()`

This can cause surprising behavior in hash-based collections.

31.6 Mistake 6: Removing Directly During Iteration

Use iterator removal or another safe strategy where appropriate.

32. Design Guidelines

When using generics and collections:

1. prefer parameterized types over raw types;
2. choose the interface type for variables when practical;
3. choose collection implementations based on required behavior;
4. prefer `ArrayList` as a general default list;
5. use sets when uniqueness matters;
6. use maps for key-based lookup;
7. use queues when processing order matters;
8. use generic methods for reusable algorithms;
9. use wildcards to increase API flexibility;
10. implement `equals()` and `hashCode()` consistently;
11. avoid depending on unspecified iteration order;
12. use meaningful comparators for sorting.

33. Summary

In this lecture, we learned that:

- generics provide compile-time type safety;
- generic classes and methods work with multiple types;
- bounded type parameters restrict accepted types;
- wildcards increase API flexibility;
- the Collections Framework provides reusable data structures;
- `List` stores ordered values and allows duplicates;
- `Set` stores unique values;
- `Map` stores key-value pairs;
- queues support ordered task processing;
- iterators provide controlled collection traversal;
- collection sorting can use natural or custom ordering;
- `equals()` and `hashCode()` matter for hash-based collections;
- choosing the correct collection improves clarity and efficiency.

34. Review Questions

1. What are generics?
2. Why are generics useful?
3. What is type safety?
4. What is a generic class?
5. What is a generic method?
6. What do the conventional symbols `T`, `E`, `K`, and `V` mean?
7. What is a bounded type parameter?
8. What is a wildcard?
9. What does `? extends Number` mean?
10. What does `? super Integer` mean?
11. What is PECS?
12. What is a raw type?
13. Why should raw types usually be avoided?
14. What is the Java Collections Framework?
15. What is the difference between `List` and `Set`?
16. What is the difference between `Set` and `Map`?
17. What is an `ArrayList`?
18. What is a `LinkedList`?
19. When is `ArrayList` usually preferred?
20. What is a `HashSet`?
21. What is a `TreeSet`?
22. What is a `HashMap`?
23. What is a `TreeMap`?
24. What is a queue?
25. What is a deque?
26. What is a priority queue?
27. What is an iterator?
28. Why can direct removal during iteration be problematic?
29. What does `Collections.sort()` do?
30. How can a list be sorted with a `Comparator`?
31. Why do `equals()` and `hashCode()` matter for hash-based collections?
32. What is an unmodifiable collection?
33. Which collection would you choose for unique student IDs?
34. Which collection would you choose for mapping IDs to students?
35. Which collection would you choose for FIFO processing?

35. Programming Exercises

Beginner Level

1. Create an `ArrayList<String>` containing five student names.
2. Add, remove, and replace elements in a list.
3. Print all elements using an enhanced for-loop.
4. Create a `HashSet<Integer>` and demonstrate duplicate removal.
5. Create a `HashMap<Integer,String>` for student IDs and names.
6. Retrieve a value from a map using a key.
7. Iterate through map entries.
8. Create a generic `Box<T>` class.
9. Create a generic `Pair<K,V>` class.
10. Create a generic method that prints any value.

Intermediate Level

1. Store `Student` objects in an `ArrayList`.
2. Sort students by GPA.
3. Sort students by name.
4. Count word frequencies using `HashMap`.
5. Remove duplicate words using `HashSet`.
6. Implement a FIFO queue.
7. Implement stack behavior using `ArrayDeque`.
8. Remove all even values from a list using an iterator.
9. Create a generic method for summing `List<? extends Number>`.
10. Create a generic maximum method using bounded type parameters.

Advanced Level

1. Build an in-memory Student Repository using `Map<Integer,Student>`.
2. Add CRUD operations to the repository.
3. Group students by course using `Map<String,List<Student»`.
4. Find the top three students by GPA.
5. Build a phone book using a map.
6. Create an inventory system using maps, lists, and sets.
7. Create a priority-based task manager using `PriorityQueue`.
8. Implement a generic repository class.
9. Design a method using wildcards to accept several numeric list types.
10. Build a mini University Management System using collections and generics.

36. Homework

Task: Develop an In-Memory Course Registration System

Create the following classes:

Illustration

```
Student
Course
Registration
StudentRepository
CourseRepository
RegistrationService
Main
```

Requirements:

1. Use `Map<Integer,Student>` to store students by ID.
2. Use `Map<String,Course>` to store courses by course code.
3. Use a suitable collection to store registrations.
4. Prevent duplicate student IDs.
5. Prevent duplicate course codes.
6. Prevent duplicate registrations.
7. Support:
   - adding students;
   - adding courses;
   - registering students for courses;

    - searching students by ID;
    - displaying all students in a course;
    - displaying all courses of a student.
8. Sort students by GPA.
9. Sort courses alphabetically.
10. Use at least one generic class or generic method.
11. Use at least one set to enforce uniqueness.
12. Use at least one map for efficient lookup.

Submission requirements:

- all Java source files;
- program output;
- explanation of why each collection type was selected;
- explanation of where generics improve type safety;
- explanation of how duplicate data is prevented;
- explanation of how sorting is implemented.

37. Key Terms

Generic
A class, interface, or method parameterized by one or more types.

Type parameter
A symbolic type such as `T`, `E`, `K`, or `V`.

Raw type
Use of a generic type without type arguments.

Bounded type parameter
A generic parameter restricted to a type hierarchy.

Wildcard
The symbol `?` representing an unknown generic type.

Collection
A Java abstraction for a group of elements.

List Ordered collection allowing duplicates.

Set Collection of unique elements.

Map
Key-value data structure.

Queue
Structure commonly used for ordered processing.

Deque
Double-ended queue.

Iterator
Object used to traverse a collection.

ArrayList
Resizable-array implementation of `List`.

HashSet
Hash-based implementation of `Set`.

HashMap
Hash-based implementation of `Map`.

PriorityQueue
Queue whose removal order is determined by priority.