UNIVERSITY OF MISKOLC
FACULTY OF MECHANICAL ENGINEERING
AND INFORMATICS

Java programming

GEIAL31A-B2a

Data structures in Java

Tamás Tompa, PhD
assistant professor
Department of Information Technology

Based on materials prepared by Miklós Szűcs

Miskolc, 2026

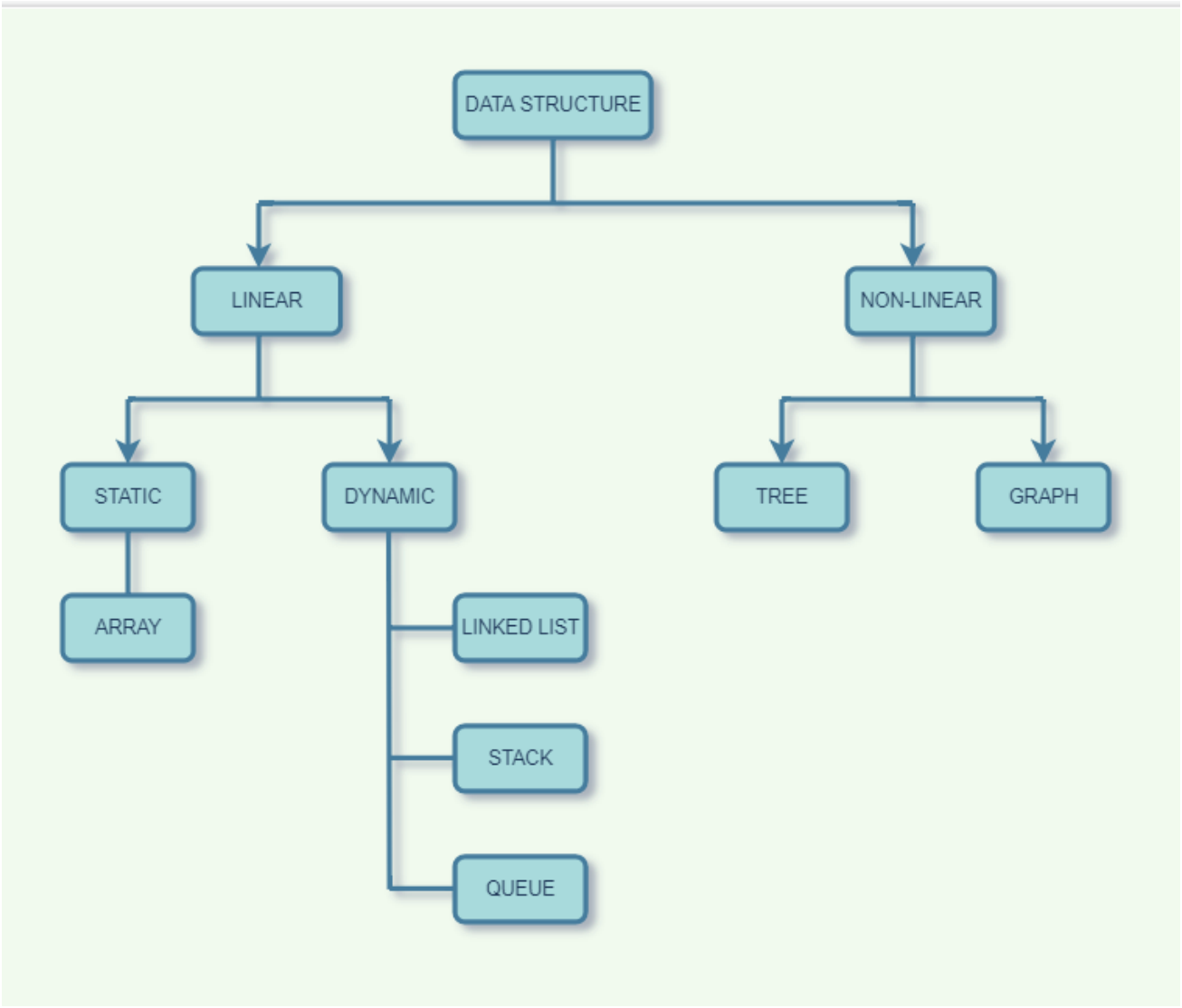
Definition

- Data structures are fundamental to any programming language
- A data structure is **defined as a format for arranging, processing, accessing, and storing data**
- Data structures are the combination of both simple and complex forms, all of which are made to **organise data for a certain use**
- Java includes many other data structures
  - arrays
  - collections (list, map, set, etc.)

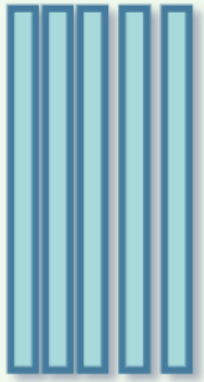
Types

- There are two types of data structures:
  - **primitive data structures**
    - byte, short, int, float, char, boolean, long, double
  - **non-primitive data structures**
    - **linear data structures**
      - the elements arranged in a linear fashion
      - each element is connected to one other element only
      - arrays, stack, queue, linked list
    - **non-linear data structures**
      - the elements arranged in a non-linear fashion
      - each element is connected to n-other elements
      - trees, heap, hash

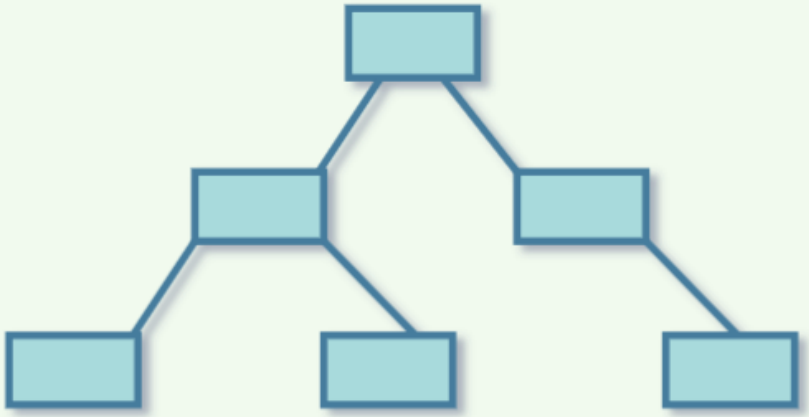
Types


DATA STRUCTURE
LINEAR
NON-LINEAR
STATIC
DYNAMIC
TREE
GRAPH
ARRAY
LINKED LIST
STACK
QUEUE

Types



LINEAR DATA STRUCTURE



NON- LINEAR DATA STRUCTURE

Arrays

- Arrays are used to **store multiple values in a single variable**
  - instead of declaring separate variables for each value
- To declare an array, define the variable type with **square brackets** [ ]
- Can be **accessed** an array element **by** referring to the **index** number
- Java uses static arrays, which means that **the number of elements is fixed** and cannot be changed
- **Array indexes start at 0**
  - the index must be an integer between 0 and (numberOfElements − 1)

Arrays - examples

```
String[] cars;
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

```
int[] myNum = {10, 20, 30, 40};
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
System.out.println(cars[0]);
// Outputs Volvo
```

Index	Element
0	Volvo
1	BMW
2	Ford
3	Mazda

Arrays - Instantiation

- `new type[number of elements];`
- **The number of elements in an array cannot be changed later**
- Accessing array elements: `arrayVariable[index]`
- In the case of reference arrays, each array element is a reference
- When the array is created, the referenced objects are not created

```
String[] cars = new String[4]; // size is 4

cars[0] = "Volvo";
cars[1] = "BMW";
cars[2] = "Ford";
cars[3] = "Mazda";

System.out.println(cars[0]); // Outputs Volvo
```

Arrays - Loop Through

- Loop through the array elements with the **for loop**, and use the **length property** to specify how many times the loop should run

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (int i = 0; i < cars.length; i++) {
  System.out.println(cars[i]);
}
```

- Or can be used foreach loop

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

for (String items : cars) {
  System.out.println(items);
}
```

Multidimensional arrays

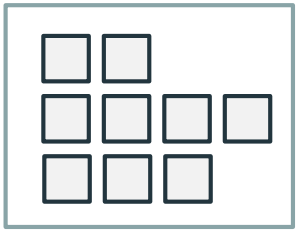
- A multidimensional array is **an array that contains other arrays**
- Java supports multidimensional arrays with an arbitrary number of dimensions
- **Two-dimensional array:** write each row inside its own curly braces:

```
int[][] a = new int[5][2];
a[0][0] = 12;
```

- **Jagged array** (array with rows of different lengths)

```
int[][] a = new int[3][];
a[0] = new int[2];
a[1] = new int[4];
a[2] = new int[3];
```

a =



Multidimensional arrays

```
int[][] myNumbers = { {1, 4, 2}, {3, 6, 8} };
```

	COLUMN 0	COLUMN 1	COLUMN 2
ROW 0	1	4	2
ROW 1	3	6	8

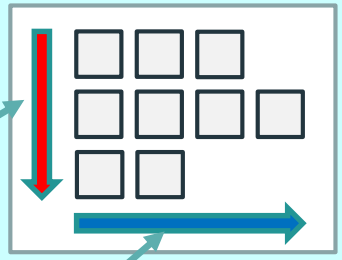
- **Access elements**
  - To access an element of a two-dimensional array, need **two indexes**
    - the **row**
    - the **column**

```
System.out.println(myNumbers[1][2]);
// Outputs 8
```

Multidimensional arrays

```
public class ArrayP2 {
   public static void main(String args[]) {
      int t [][] = {
1.          {0,0,0},
            {0,0,0,0},
            {0,0}
      };
2.    for (int i=0; i<t.length; i++)
3.       for (int j=0; j<t[i].length; j++)
4.          t[i][j] = (int)(Math.random()*9)+1;

5.    for(int[] x: t){
6.       System.out.println();
         for(int y: x)
7.          System.out.print(y+" ");
8.    }
   }
}
```



```
6 2 5
1 7 9 2
9 2
```

Multidimensional arrays

1. Creating a matrix instance with initial values
2. `t.length`: number of rows
3. `t[i].length`: length of the given row (number of elements)
4. Accessing a matrix element using its indices
5. Enhanced for loop over a matrix. first, a row is assigned to an array,
7. then the elements of that array are assigned to individual variables
6. Printing an empty line to improve readability
8. Printing the value of a matrix element

Enums

- An enum is a **special "class"** that represents a **group of constants** (unchangeable variables, like final variables)
- To create an enum, use the enum keyword and separate the constants with a comma
- Superclass: `java.lang.Enum`
- Enum constants do not require explicit values, they are **referenced by their names** only

```
enum Level
{
  LOW,
  MEDIUM,
  HIGH
}
```

```
Level myVar = Level.MEDIUM;
```

Enums

- Inside a class

```
public class Main {
  enum Level {
    LOW,
    MEDIUM,
    HIGH
  }

  public static void main(String[]
args) {
    Level myVar = Level.MEDIUM;
    System.out.println(myVar);
  }
}
```

Enums

- An enum can also have a **constructor** just like a class
- The constructor is called automatically when the constants are created
- For example, each constant in the enum has a value (a string) that is set through the constructor

Enums

```
enum Level {
  LOW("Low level"),
  MEDIUM("Medium level"),
  HIGH("High level");

  private String description;

  private Level(String
description) {
    this.description =
description;
  }

  public String
getDescription() {
    return description;
  }
}
```

```
public class Main {
  public static void main(String[]
args) {
    Level myVar = Level.MEDIUM;

System.out.println(myVar.getDescriptio
n());
  }
}
```

Enums

- Can be loop through the constants and print their values using the `values()` method:

```
for (Level myVar : Level.values()) {
  System.out.println(myVar + ": " + myVar.getDescription());
}
```

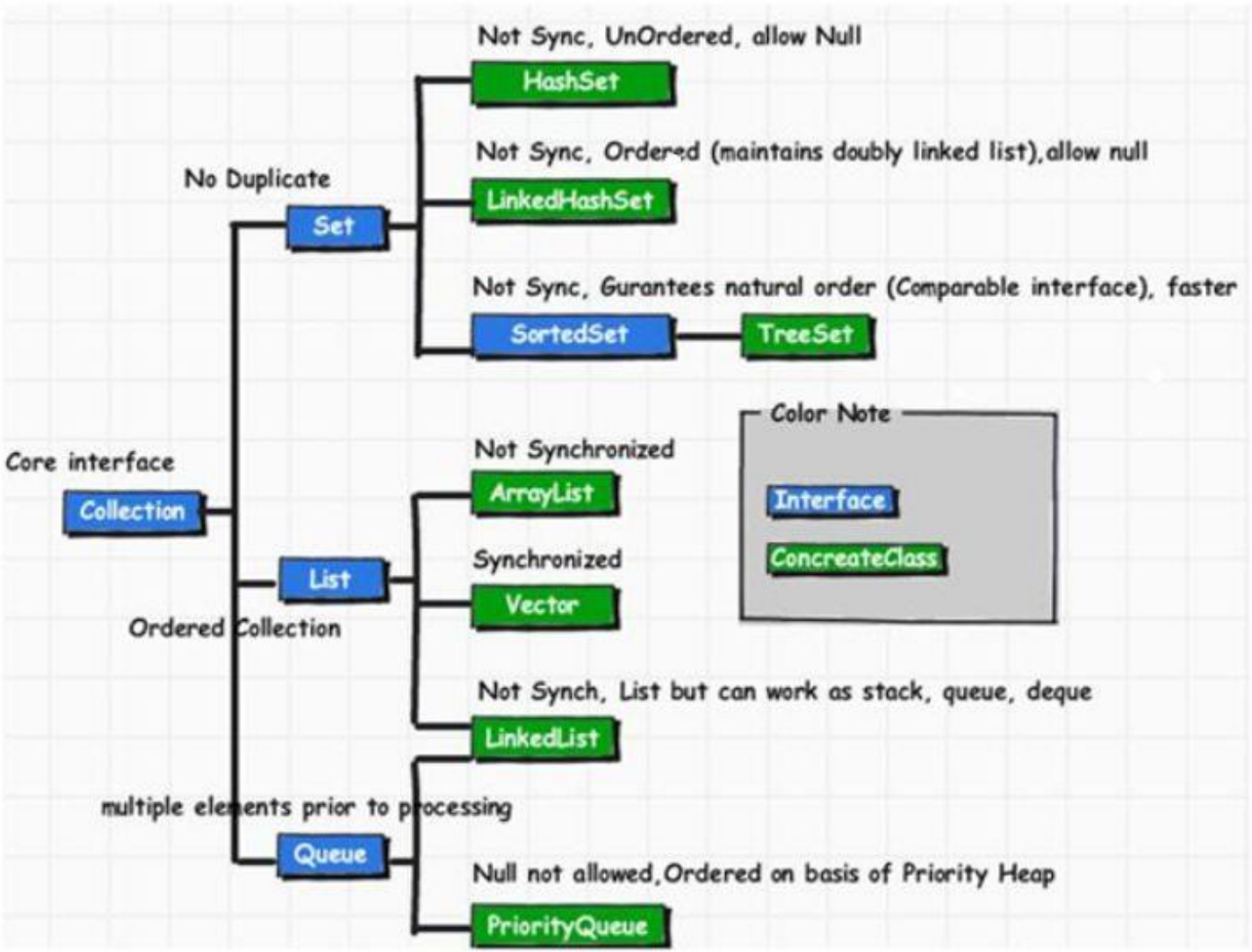
```
LOW: Low level
MEDIUM: Medium level
HIGH: High level
```

Collections framework

- **Limitations of arrays**
  - **fixed size**, the size cannot be increased dynamically
  - if the **size** is chosen **too large**, it results in **wasted memory**
  - modifying an array:
    - problems: inserting an element, deleting an element
    - **searching**: all elements must be checked (**linear search**)

- **Solution**
  - **Collections**: **dynamically resizable data structures**
  - Framework: in the `java.util` package
  - **provides a set of interfaces** (List, Set, Map) and a set of classes (ArrayList, HashSet, HashMap, etc.) that implement those interfaces

Collections framework



ArrayList

- An ArrayList is like a **resizable array**
- It is part of the `java.util` package and implements the List interface
- The difference between a built-in array and an ArrayList, is that the size of an array cannot be modified
- The list **must be instantiated**
- **Only wrapper (reference) classes can be used** as the element type.Integer, Double, Character, Long, Boolean,
- A list **can store only reference types**, therefore primitive types cannot be stored directly

```
import java.util.ArrayList; // Import the ArrayList class

ArrayList<String> cars = new ArrayList<String>();
```

ArrayList - add elements

- o Add elements to an ArrayList, **use the `add()` method**:

```
ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
}
```

- Also add an element at a specified position **by referring to the index number**:

```
ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");

    cars.add(0, "Mazda"); // Insert element at the beginning of the list (0)
```

ArrayList - operations

- **Acces an element**
  - to access an element in the ArrayList, **use the `get()` method** and refer to the index number:

```
cars.get(0);  // Get the first element
```

- **Change an element**
  - to modify an element, **use the `set()` method** and refer to the index number:

```
cars.set(0, "Opel");
```

- **Remove an element**
  - to remove an element, **use the `remove()` method** and refer to the index number:

```
cars.remove(0);
```

ArrayList - operations

- **Remove all elements**
  - to remove all the elements **use the `clear()` method**:

```
cars.clear();
```

- **Size of ArrayList**
  - to find out how many elements an ArrayList have, **use the `size()` method**:

```
cars.size();
```

ArrayList - iterate

- Loop through the elements of an ArrayList with a **for loop**, and use the `size()` method to specify how many times the loop should run:

```
ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

for (int i = 0; i < cars.size(); i++) {
      System.out.println(cars.get(i));
    }
}
```

ArrayList - iterate

- Loop through the elements of an ArrayList with a for loop, can be used the **for-each loop**:

```
for (String item : cars) {
   System.out.println(item);
}
```

ArrayList - other types

- To use other types, such as int, you must specify an equivalent **wrapper class**
  - Integer for int, Boolean for boolean, Character for char, Double for double, etc:

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>();
    myNumbers.add(10);
    myNumbers.add(15);
    myNumbers.add(20);
    myNumbers.add(25);

    for (int i : myNumbers) {
      System.out.println(i);
    }
```

ArrayList - sort

- The **`sort()` method** for sorting lists alphabetically or numerically:

```
ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");

    Collections.sort(cars);  // Sort cars

    for (String i : cars) {
      System.out.println(i);
    }
```

Set

- The Set is used to store a collection of unique elements
- Unlike a List, a Set does not allow duplicates, and it does not preserve the order of elements
- Common classes that implement Set:
  - HashSet - fast and unordered
  - TreeSet - sorted set
  - LinkedHashSet - ordered by insertion

List	Set
Allows duplicates	Does not allow duplicates
Maintains order	Does not guarantee order
Access by index	No index-based access

Hashset

- **Every element is unique**

```
import java.util.HashSet; // Import the HashSet class

HashSet<String> cars = new HashSet<String>();
```

```
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("BMW");  // Duplicate
cars.add("Mazda");
System.out.println(cars);
```

```
[Volvo, Mazda, Ford, BMW]
```

Hashset

- **Check if an element exists**

```
cars.contains("Mazda");
```

```
true
```

- **Remove an element**

```
cars.remove(„Volvo");
```

```
[Mazda, Ford, BMW]
```

- **Remove all elements**

```
cars.clear();
```

```
[]
```

- **Size**

```
cars.size();
```

```
4
```

TreeSet

- It stores **unique elements in sorted order**
  - the elements will be sorted automatically
  - the duplicated elements will only appear once

```
TreeSet<String> cars = new TreeSet<>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("BMW");  // Duplicate
    cars.add("Mazda");

    System.out.println(cars);
```

```
[BMW, Ford, Mazda, Volvo]
```

Map

- It is used to **store key-value pairs**
  - each **key** must be **unique**, but **values** can be **duplicated**
- Interface
- Common classes that implement Map:
  - HashMap - fast and unordered
  - TreeMap - sorted by key
  - LinkedHashMap - ordered by insertion

Method	Descrition
put()	Adds or updates a key-value pair
get()	Returns the value for a given key
remove()	Removes the key and its value
containsKey()	Checks if the map contains the key
keySet()	Returns a set of all keys

HashMap

- It stores items in **key/value pairs**, where each key maps to a specific value
- It implements the Map interface
- Instead of accessing elements by an index, can be used a key to retrieve its associated value
- A HashMap can store many different combinations:
  - String keys and Integer values
  - String keys and String values

```
import java.util.HashMap; // Import the HashMap class

HashMap<String, String> capitalCities = new HashMap<>();
```

HashMap - add items

```
HashMap<String, String> capitalCities = new HashMap<String, String>();

// Add keys and values (Country, City)
capitalCities.put("England", "London");
capitalCities.put("India", "New Dehli");
capitalCities.put("Austria", "Wien");
capitalCities.put("Norway", "Oslo");
capitalCities.put("Norway", "Oslo"); // Duplicate
capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);
```

```
{Austria=Wien, USA=Washington DC, Norway=Oslo, England=London, India=New Dehli}
```

HashMap - operations

- **Access an item**

```
capitalCities.get("England");
```

```
London
```

- **Remove an item**

```
capitalCities.remove("England");
```

```
{Austria=Wien, USA=Washington DC, Norway=Oslo, India=New Dehli}
```

- **Remove all item**

```
capitalCities.clear();
```

TreeMap

- It stores **key/value pairs in sorted order by key**
- It implements the Map interface

```
import java.util.TreeMap; // Import the TreeMap class

TreeMap<String, String> capitalCities = new TreeMap<>();

capitalCities.put("England", "London");
capitalCities.put("India", "New Dehli");
capitalCities.put("Austria", "Wien");
capitalCities.put("Norway", "Oslo");
capitalCities.put("Norway", "Oslo"); // Duplicate
capitalCities.put("USA", "Washington DC");

System.out.println(capitalCities);
```

```
{Austria=Wien, England=London, India=New Dehli, Norway=Oslo, USA=Washington DC}
```

Iterator

- An Iterator is **an object that can be used to loop through collections**
- It is called an "iterator" because "iterating" is the technical term for looping
- To use an Iterator, you must import it from the `java.util` package

```
ArrayList<String> cars = new ArrayList<String>();
cars.add("Volvo");
cars.add("BMW");
cars.add("Ford");
cars.add("Mazda");

Iterator<String> it = cars.iterator();

// Print the first item
System.out.println(it.next());
```

Summerize

Interface	Common classes	Descrition
List	ArrayList, LinkedList	Ordered collection that allows duplicates
Set	HashSet, TreeSet, LinkedHashSet	Collection of unique elements
Map	HashMap, TreeMap, LinkedHashMap	Stores key-value pairs with unique keys

Equals() and hashCode()

- **Equals()**
  - The `equals(Object o)` method is used to **check logical (content) equality**
  - The default implementation in Object compares references (==)
  - It is usually overridden to define when two objects should be considered equal
  - Typical rule: compare the significant fields of the class
  - Example: Two Book objects are equal if their author and title are the same

Equals() and hashCode()

- **hashCode()**
  - The `hashCode()` method **returns an integer hash value for an object**
  - It is used by hash-based collections:
    - HashSet
    - HashMap
    - HashTable
  - The hash code is used to speed up searching, not to decide equality by itself

Equals() and hashCode()

- Java defines a strict rule
- **If `a.equals(b) == true` then `a.hashCode() == b.hashCode()` must also be true**
- **The reverse is not required**
  - same `hashCode()` ≠ guaranteed equality
- Whenever `equals()` is overridden, `hashCode()` must also be overridden
- `hashCode()`: fast filtering (which bucket?)
- `equals()`: exact comparison (are they really equal?)

Sorting objects

- **Comparable and Comparator interfaces** are used for sorting objects
- The main difference between Comparable and Comparator:
  - **Comparable**: it is used to define the **natural ordering** of the objects within the class
    - implemented inside the class (`class Book implements Comparable<Book>`)
    - `compareTo()` method
    - `Arrays.sort(array)` (no comparator needed)
  - **Comparator**: it is used to define **custom sorting logic externally**
    - implemented in a separate class or nested class
    - `compare()` method
    - `Arrays.sort(array, comparator)`
    - `Arrays.sort(array, new Employee.NameSorter());`

Sorting objects

- **Java8 comparator**
  - Java 8 introduced a more simple way to write comparators using **lambda expressions**
  - one criterion, ascending

```
Arrays.sort(array, Comparator.comparing(Person::getAge));
```

  - one criterion, descending

```
Comparator<Employee> compByAge = Comparator.comparing(Person::getAge);
Arrays.sort(array, compByAge.reversed());
```

  - multiple criteria (tie-breakers)

```
Arrays.sort(array, Comparator.comparing(Employee::getSalary)
    .thenComparing(Person::getAge)
    .thenComparing(Person::getName));
```

Sorting objects

- **Summerize**
  - **Comparable**
    - natural ordering inside the class
  - **Comparator**
    - alternative ordering outside the class
  - **Java 8 Comparator**
    - clean, chainable comparators using comparing(...), thenComparing(...), reversed()

Thank you for your attention!