# Advanced Programming Methods

Iuliana Bocicor
*maria.bocicor@ubbcluj.ro*

Babes-Bolyai University

2024

# Overview

# Packages I

- Are used to group classes and interfaces.
- Benefits: avoid name conflicts, write a better maintainable code.
- Built-in packages, e.g.: *java.lang*, *java.util*, *java.sql*, *java.time*, *java.time.format* and many others.
- User-defined packages:
  - defined by the instruction package.
  - package must be the first instruction in the file.
  - a corresponding folder (having the exact same name) is created and all classes and interfaces are saved in this folder.
- If a *.java* file does not contain the instruction package, all the file classes are part of an unnamed default package.

# Packages II

- To use class **ClassName** declared in package **PackageName**:

1.
```
        // ...
public static void main(String args[]){
    PackageName.ClassName obj =
                            new PackageName.ClassName();
}
```

2.
```
        import PackageName.*  OR
        import PackageName.ClassName
        // ...
public static void main(String args[]){
    ClassName obj = new ClassName();
}
```

# Packages III

- From a package we can import one or several classes, or all classes and interface contained.
- A file may contain many import instructions.
- These must be at the beginning of the file, before class declarations.
- The package *java.lang* is implicitly imported by the compiler.

# Nested classes I

- A class declared within another class is called a *nested class*.
- They allow the logical grouping of related classes.

# Nested classes II

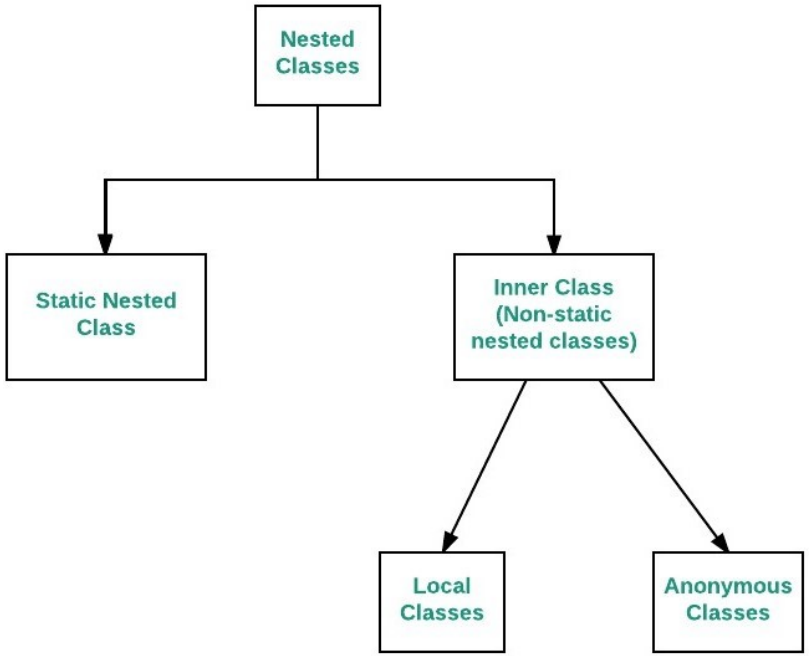


Figure: Figure source: Nested Classes in Java

# Nested classes III

- A nested class cannot exist independently of its outer class.
- A nested class can access all members of its outer class.
- A nested class is a member of its enclosing class. As a member it can be declared private, public, protected, or package private (default).
- To instantiate a nested class we need an instance of the outer class (unless the inner class is static).
- To access the current outer object (class **Outer**) from within the inner class: Outer.this.

## Example

nested.NestedClasses.

# Introduction to Generics

- What type of objects does the *LinkedList* in the previous example (NestedClasses.java) contain?
- **Problem**: we need to downcast each time when we refer an element in the list.

```
String elem = (String)iterator.element();
```

# Generic types I

- A *generic type* is a generic class or interface that is parameterized over types.
- They are similar to C++ templates.
- Type parameter naming conventions:
  - E - Element (used by Java Collections Framework).
  - K - Key.
  - V - Value.
  - N - Number.
  - T - Type.
- In a generic class we cannot have generic static data members. Why not?

# Generic types II

- Defining a generic class:

```
class ClassName<T1, T2, ..., Tn> { /* ... */ }
```

- Invoking and instantiating a generic type:

```
ClassName<Integer> = new ClassName<Integer>();
```

- Primitive types (int, byte, char, float, double, ...) cannot be used when instantiating generic types.

## Example

generics.GenericLinkedList. generics.GenericStack.

# Generic methods

- Methods that introduce their own type parameters.
- The can be static or non-static.
- The list of type parameters, inside angle brackets, must appear before the method's return type.

## Example

generics.GenericMethods.

# Type erasure I

- As opposed to C++, Java does not create a new class for each instance of a generic class.
- The Java compiler applies **type erasure**:
  - all type parameters in generic types are replaced with their bounds or Object if the type parameters are unbounded;
  - extra type casts are being generated if necessary to preserve type safety.
- The bytecode after compilation contains only normal classes, interfaces and methods (no new types are produced).
- Casting is applied at compile time, ensuring type safety.

# Type erasure II

```
class Stack<E>
{
    // ...

    E top()
    {
        // ...
    }
}
```

- The code will be replaced by (default) Object:

# Type erasure III

```
class Stack
{
    // ...

    Object top()
    {
        // ...
    }
}
```

# Bounded Type Parameters

- Allow restricting the types that can be used as type arguments in a parameterized type.
- To specify an upper bound / upper bounds the parameter's name must be followed by the extends keyword:

```
class Calculator<T extends Number>
class Calculator<T extends Number & Comparable>
```

- Any variable of type T can call methods from the classes or interfaces specified as upper bounds.

## Example

generics.BoundedTypeParameters.

# Generic classes and subtyping

```
class Base { /* ... */ }
class Derived extends Base { /* ... */ }
```

- **But**: List<Derived> **is not** a subtype of List<Base> (see Explanation).
- There is no relation between the two types.

```
public static void printCollection
                        (List<Person> persons)
```

- The method above cannot be used for a list of students (List<Student>, where *Student* is a subclass of *Person*).

# Wildcards

- The question mark (?) is known as the wildcard in generic programming.
- It represents an unknown type.
- Upper bounded wildcards:

```
public static void printCollection
                (LinkedList<? extends Person>)
```

- Lower bounded wildcards:

```
public static void printCollection
                (LinkedList<? super Student>)
```

## Example

generics.Wildcards.

# Java Collections Framework

- A collection is an object that represents a group of objects (such as the classic Vector class).
- A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
- Advantages:
  - Code reuse.
  - Programming effort reduction.
  - Performance increasing.
  - Polymorphic algorithms.
  - Use of generic types.

# Hierarchy

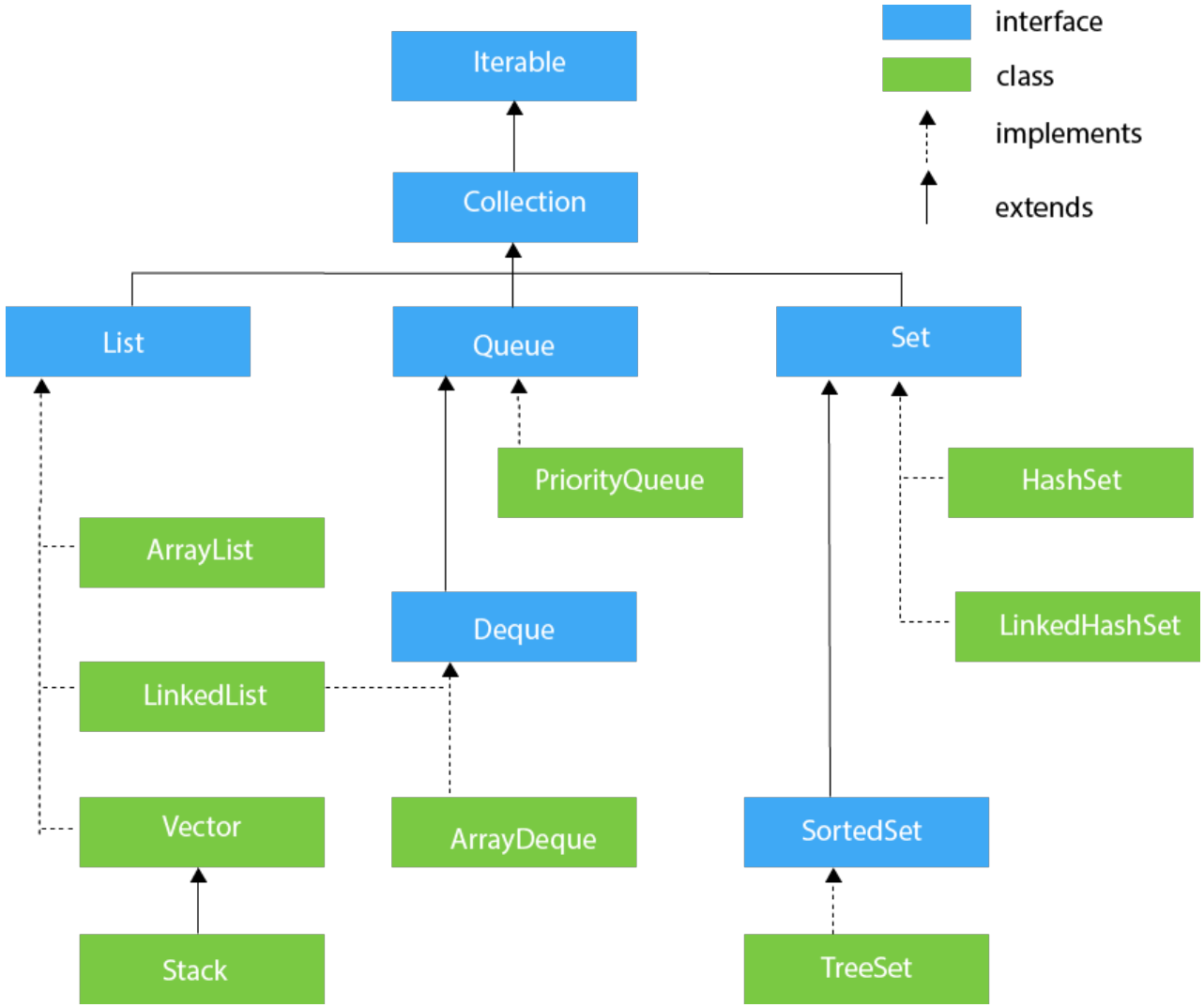


Figure: Figure source: JavaTpoint

# Implementations

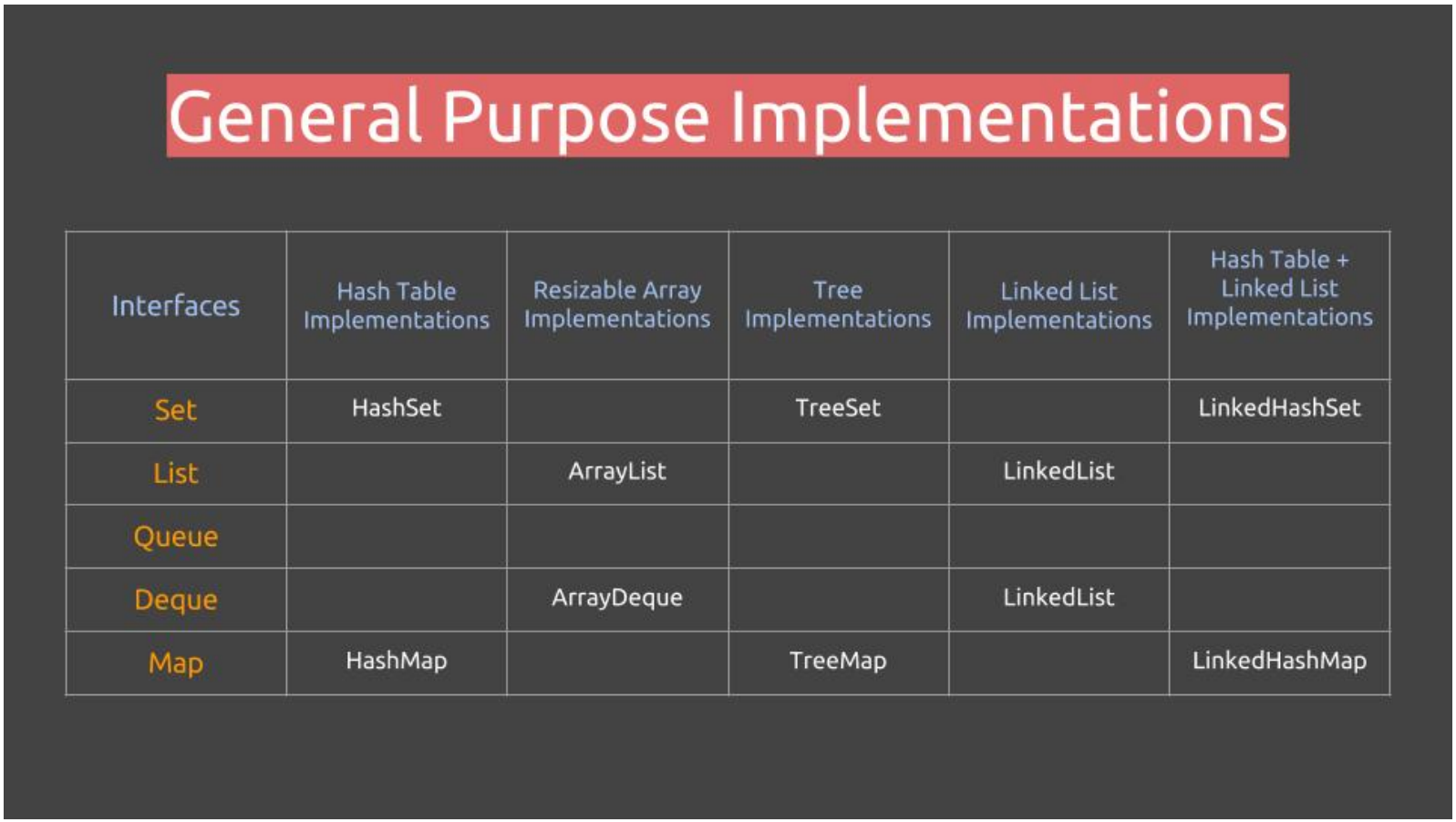


Figure: Figure source: Quickgrid

# Examples

```
import java.util.*;
// ...
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
List<Integer> w = new Vector<>();
Map<Integer, String> map = new HashMap<>();
```

# Collection interface

- It is the root interface in the collection hierarchy.
- It contains methods that perform basic operations:
  - boolean add(Object)
  - boolean addAll(Collection)
  - void clear()
  - boolean contains(Object)
  - boolean containsAll(Collection)
  - boolean equals(Object)
  - boolean isEmpty()
  - Iterator iterator()
  - boolean remove(Object)
  - boolean removeAll(Collection)
  - boolean retainAll(Collection)
  - int size()
  - Object[] toArray()
  - Object[] toArray(Object[])

# Iterators

- Provide a generic way to traverse the collection, access and remove data elements of the collection, regardless of the implementation.
- java.util.Iterator<E>
- Any class implementing the Collection interface can return an Iterator through the method *iterator()*.
- The interface java.lang.Iterable<T> is the root interface in the hierarchy.
- Any class implementing the Java *Iterable* interface can be iterated with the *for-each* loop.

```
public interface Iterable<T> {
  public Iterator<T> iterator();
}
```

## Example

collections.Iterators.

# Comparing objects I

- There are 2 ways of comparing objects in Java:
  - by implementing the *Comparable* interface:

```
public interface Comparable<T>  {
    int compareTo(T o);
}
```

  - by using a *Comparator*.

```
public interface Comparator<T>  {
    int compare(T o1, T o2);
}
```

# Comparing objects II

- Comparable objects can be compared by only one criterion.
- For sorting, if the objects implement *Comparable* they are said to be sorted by their natural order (object itself must know how it is to be ordered).
- Comparable objects can only be sorted by one criterion, the one implemented in the function *compareTo*.
- For permitting several types of sorting, we must use *Comparator*.
- Comparator is external to the object. Multiple classes implementing Comparator can be created for sorting by different criteria.

## Example

collections.Comparators.

# java.util.Collections

- This class consists exclusively of static methods that manipulate or return Collections.
- Examples:
  - sort a list (*sort*)
  - search object in list (*binarySearch*)
  - Finding the minimum or maximum (*min*, *max*)
  - Reversing, shuffling elements (*reverse*, *shuffle*)
  - Copying a list into another (*copy*).

# Lists

- Classes that implement the *List* interface: *ArrayList*, *LinkedList*, *Vector*.
- *Vector* and *ArrayList* use an array internally.
- Both are dynamically resizable.
- The difference between them is that the *Vector*'s methods are synchronised, while the *ArrayList*'s are not.
- *LinkedList* implements a doubly linked list.

# Sets

- Classes that implement the *Set* interface: *HashSet*, *LinkedHashSet*, *TreeSet*.
- *HashSet* uses a hash table which is actually a *HashMap* instance.
- There is no constant order of elements over time.
- Duplicate values are not allowed.
- *LinkedHashSet* is a version of the HashSet that maintains a doubly linked list. It preserves order based on *insertion order*.
- *TreeSet* extends the *SortedSet* interface. Internally it uses *TreeMap* to store the elements.

# Maps

- Classes that implement the *Map* interface: *HashMap*, *WeakHashMap*, *TreeMap*.
- *HashMap* stores (key, value) pairs. It uses a hash table. Elements can be accessed via the key and keys can be iterated.
- *WeakHashMap* is similar to *HashMap*, however when a key is not in use in the application in any place, that entry will be deleted from memory.
- *TreeMap* uses a Red-Black tree as data structure.

# Choosing the right Java collection

- *HashMap* and *HashSet* have slightly better performance than LinkedHashMap, but the iteration order is undefined. Accessing elements is very fast.
- *TreeSet* and *TreeMap* are ordered and sorted, but slower.
- *LinkedList* has fast adding to the start of the list, and fast deletion from the interior via iteration. However, randomly accessing elements is not fast.
- Accessing elements is fast in *Vector* and *ArrayList*. However, inserts and deletes are not fast in these.

## Example

collections.Collections.

# Summary

- Packages: code organisation.
- Nested classes: logical grouping of related classes.
- Generics: methods, classes or interfaces parametrized over types.
- Java collections framework.
- *Next week*:
  - Exceptions.
  - I/O.
  - JUnit.