Advanced Programming

Generics

Collections

The Context

- "Create a data structure that stores elements:
  - *a stack, a linked list, a vector*
  - *a graph, a tree, etc."*
- What data type to use for representing the elements of the structure?

Homogenous structure	Heterogeneous structure

`class Stack {`
`  private int[] items;`
`  public void push (int item) { ...}`
`  public int peek() { … }`
`}`
`…`
`Stack stack = new Stack();`
`stack.push(100);`
`stack.push(200);`
~~`stack.push("Hello World!");`~~
	`class Stack {`
	`  private Object[] items;`
	`  public void push (Object item) {...}`
	`  public Object peek() { … }`
	`}`
	`…`
	`Stack stack = new Stack();`
	`stack.push(100);`
	`stack.push(new Rectangle());`
	`stack.push("Hello World!");`
	`String s = (String) stack.peek;`

Generics

- Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.

```
public Stack<String> { … }
```

- Stronger type checks at compile time.

```
stack.push(new Rectangle());
```

- Elimination of casts.

```
String s = (String) stack.peek();
```

- Enabling generic algorithms.

Defining a Generic Type

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

or

```
interface IName<T1, T2, ..., Tn> { … }
```

```
/**
 * A generic version of the Stack class
 * @param <E> the type of the elements
 */
public class Stack<E> {

    // E is a generic data type
    private E[] items;

    public void push(E item) { ... }
    public E peek() { .. }
}
```

E is the type parameter

Type Parameter Naming Conventions

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

```
public class Node<T> { … }
public interface Pair<K, V> { … }
public class PairImpl<K, V> implements Pair<K, V> {...}
```

Instantiating a Generic Type

- Generic Invocation

String is the type argument

```
Stack<String> stack = new Stack<String>();

Pair<Integer,String> pair =

          new PairImpl<Integer,String>(0, "ab");
Stack<Node<Integer>> nodes = new Stack<Node<Integer>>();
```

- *The Diamond* **<>**

```
Stack<String> stack = new Stack<>();

Pair<Integer,String> pair = new PairImpl<>(0, "ab");

Stack<Node<Integer>> nodes = new Stack<>();
```

The compiler can determine, or infer, the type arguments from the context.

Generic Methods

Generic methods are methods that introduce their own type parameters.

```
public class Util {
  public static <T> int countNullValues(T[] anArray) {
    int count = 0;
    for (T e : anArray)
        if (e == null) {
            ++count;
        }
    return count;
  }
}
Util.countNullValues(new String[]{"a", null, "b"});
Util.countNullValues(new Integer[]{1, 2, null, 3, null});
```

Bounded Type Parameters

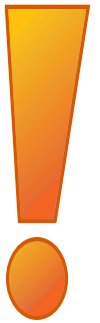
```
class D <T extends A & B & C> { /* ... */ }
```

```
public class Node<T extends Number> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }

    // Generic Method
    public <U extends Integer> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Node<Double> node = new Node<>();
        node.set(12.34);                //OK
        node.inspect(1234);             //OK
        node.inspect(12.34);            //compile error!
        node.inspect("some text");      //compile error!
    }
}
```

Generics, Inheritance, Subtypes

"is a"

- ✔ `Integer extends Object`
- ✔ `Integer extends Number`
- • ~~**`Stack<Integer> extends Stack<Object>`**~~
- • ~~**`Stack<Integer> extends Stack<Number>`**~~



Given two concrete types A and B (for example, *Number* and *Integer*), *MyClass<A>* has no relationship to *MyClass<B>*, regardless of whether or not A and B are related. The common parent of *MyClass<A>* and *MyClass<B>* is *Object*.

"This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn "

Wildcards

- Upper bounded

```
public double sumOfList(List<? extends Number> list) {
    //it works on List<Integer>, List<Double>, List<Number>, etc.
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

- Unbounded

```
public void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
}
```

- Lower bounded

```
public void addNumbers(List<? super Integer> list) {
    //it works on List<Integer>, List<Number>, and List<Object> —
    //anything that can hold Integer values.
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Java Collections Framework

Java Collections Framework

- A *collection* is an object that groups multiple elements into a single unit.
- *Vectors, Lists, Stacks, Sets, Dictionaries, Trees, Tables, etc.*
- Promotes software reuse
- Reduces programming effort
- Increases program speed and quality
- Benefits from *polymorphic algorithms*
- Uses *generics*

Collections Framework

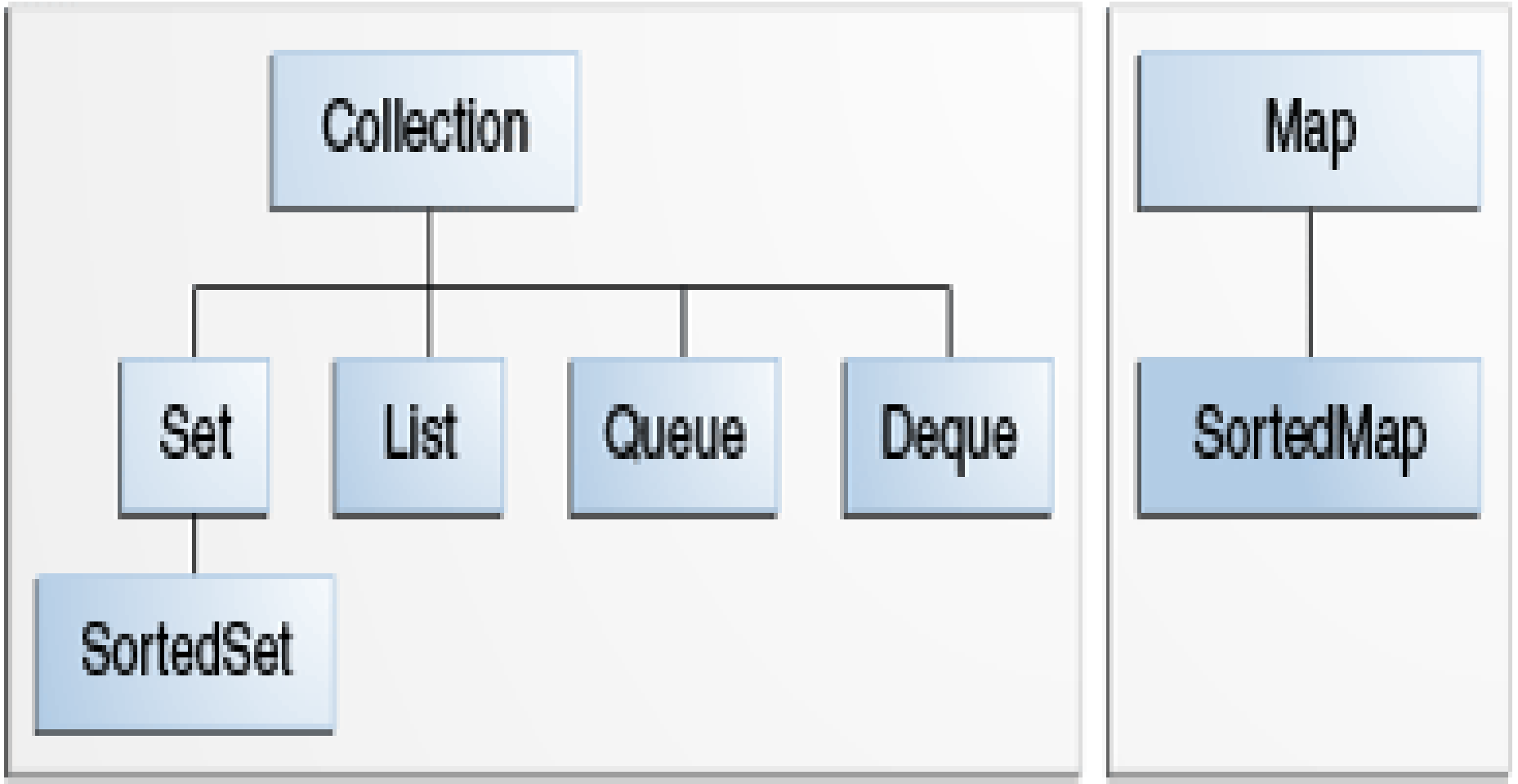
- Interface
- Abstract class
- Concrete implementation

List

AbstractList

ArrayList
LinkedList
Vector

The Core Collection Interfaces

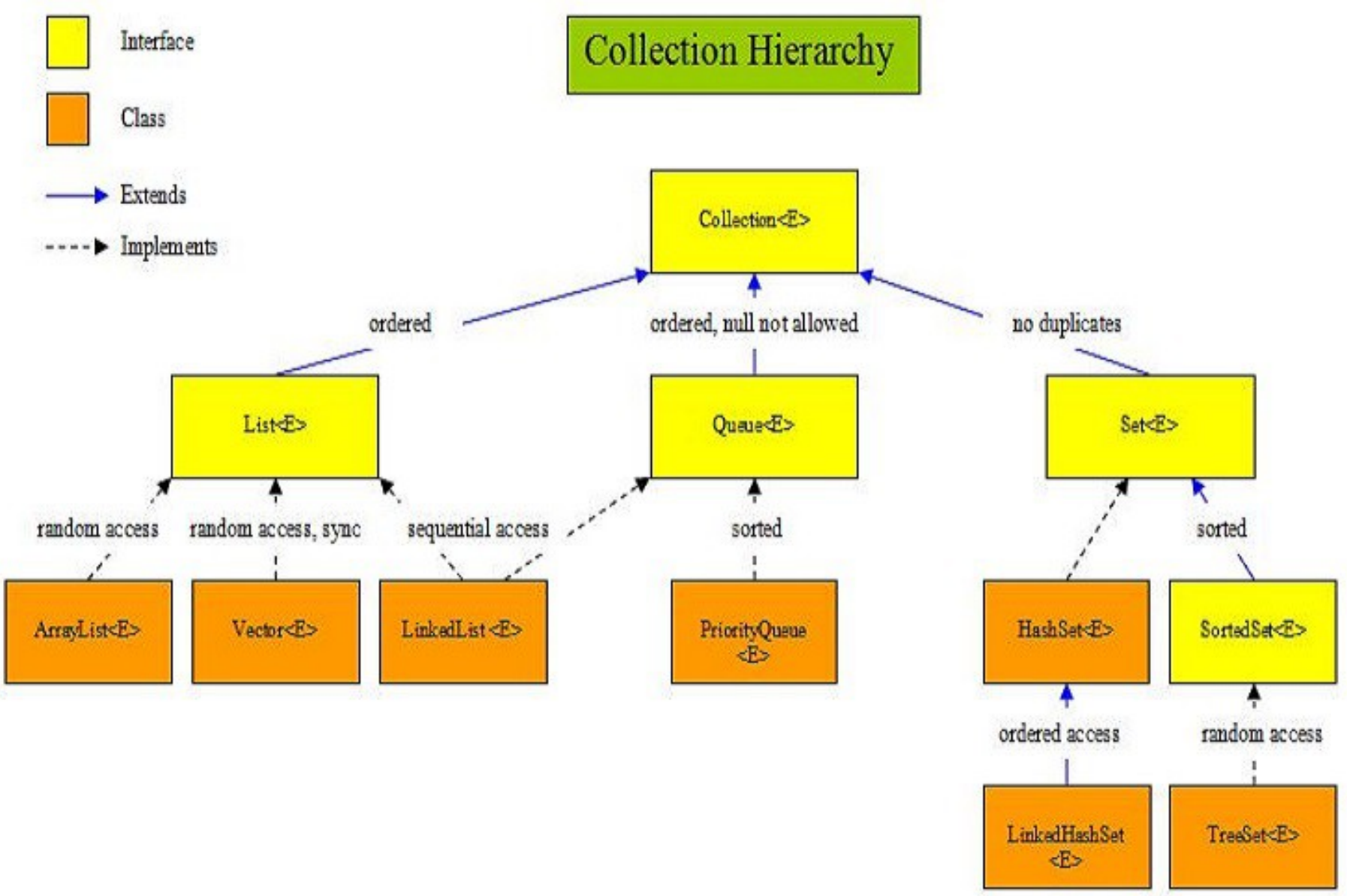


Implementations

Interfața	Hash	Array	Tree	Linked	Hash+Linked
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList Vector		LinkedList	
Queue					
Deque		ArrayDeque		LinkedDeque	
Map	HashMap Hashtable		TreeMap		LinkedHashMap

```
Set set = new HashSet(); --raw generic type (Object)
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
List<Integer> list = new Vector<>();
Map<Integer, String> map = new HashMap<>();
```

Collections Hierarchy



Iterating Over a Collection

- Indexed Collections

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

- *Iterator* and *Enumeration*

```
for (Iterator it = set.iterator(); it.hasNext(); ) {
  System.out.println(it.next());
  it.remove();
}
```

- *for-each*

```
List<Student> students = new ArrayList<Student>();
...
for (Student student : students) {
    student.setGrade(10);
}
```

ArrayList or *LinkedList*?

```
public class TestList {
    private final static int N = 100_000;
    public void testAdd(List<Integer> list) {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            list.add(i);
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Add: " + (t2 - t1));
    }
    public void testGet(List<Integer> list) {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            list.get(i);
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Get: " + (t2 - t1));
    }
    public void testRemove(List<Integer> list) {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            list.remove(0);
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Remove : " + (t2 - t1));
    }
    public void runTests(List<Integer> list) {
        testAdd(list); testGet(list); testRemove(list);
    }
    public static void main(String args[]) {
        TestList app = new TestList();
        app.runTests(new ArrayList<>());
        app.runTests(new LinkedList<>());
    }
}
```

	ArayList	LinkedList
add	6 ms	8 ms
get	3 ms	4320 ms
remove	868 ms	6 ms

Conclusion: Choosing a certain implementation depends on the nature of the problem being solved.

ArrayList or *HashSet*?

```
public class TestSet {
    final static int N = 100_000;
    public void testAdd(Collection<Integer> collection) {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            collection.add(i);
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Add: " + (t2 - t1));
    }
    public void testIterate(Collection<?> collection) {
        long t1 = System.currentTimeMillis();
        for (Object obj : collection) {
            obj.toString(); //do something
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Iterate: " + (t2 - t1));
    }
    public void testContains(Collection<?> collection) {
        long t1 = System.currentTimeMillis();
        for (int i = 0; i < N; i++) {
            collection.contains(i);
        }
        long t2 = System.currentTimeMillis();
        System.out.println("Contains: " + (t2 - t1));
    }
    public void runTests(Collection<Integer> collection) {
        testAdd(collection); testIterate(collection); testContains(collection);

    }
    public static void main(String args[]) {
        TestSet app = new TestSet();
        app.runTests(new ArrayList<>());
        app.runTests(new HashSet<>());
    }
}
```

	ArayList	HashSet
add	6 ms	26 ms
iterate	51 ms	12 ms
contains	3997 ms	7 ms
Memory	low	high

Conclusion: Choosing a certain implementation depends on the nature of the problem being solved.

Aggregate Operations

- *Stream* - sequence of elements supporting sequential and parallel aggregate operations.
- *Pipeline* - a sequence of aggregate operations.

```
persons.stream()
    .filter(p -> p.getAge() >= 18)
    .filter(p -> p.getName().endsWith("escu"))
    .forEach(s -> System.out.println(s.getName()));
```

- *Reduction* and Terminal ops.

```
double averageAge = persons.stream()
    .filter(p -> p.getAge() >= 18)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

- **Source**
  array, collection, ...
- **Intermediate** operations
  filter, distinct, sorted,...
- **Reduction** operations
  map, mapToInt, ...
- **Terminal** operations
  average, min, max,...

Polymorphic Algorithms

`java.util.Collections`

- `sort`
- `shuffle`
- `binarySearch`
- `reverse`
- `fill`
- `copy`
- `min`
- `max`
- `swap`
- `enumeration`
- `unmodifiable`*`CollectionType`*

```
List<String> immutablelist = Collections.unmodifiableList(list);
immutablelist.add("Oops...?!");
```

- `synchronized`*`CollectionType`*

What DesignPattern?