COMP2511

Generics and Collections in Java

Prepared by

Dr. Ashesh Mahidadia

Generics in Java

(Part 1)

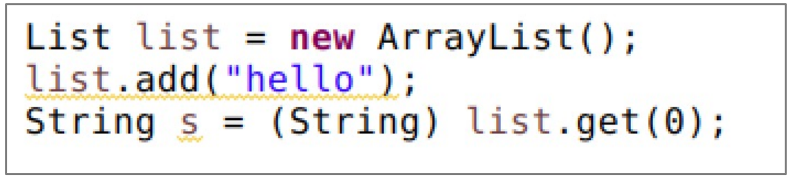
Generics in Java

Generics enable types (classes and interfaces) to be parameters when defining:

- classes,
- interfaces and
- methods.

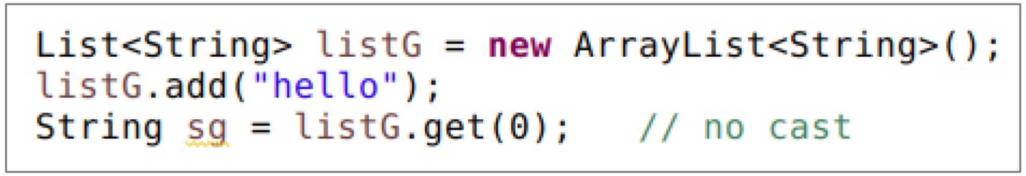
Benefits

- Removes *casting* and offers stronger type checks at compile time.
- Allows implementations of generic algorithms, that work on collections of different types, can be customized, and are type safe.
- Adds stability to your code by making more of your bugs detectable at compile time.



```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Without Generics



```
List<String> listG = new ArrayList<String>();
listG.add("hello");
String sg = listG.get(0);   // no cast
```

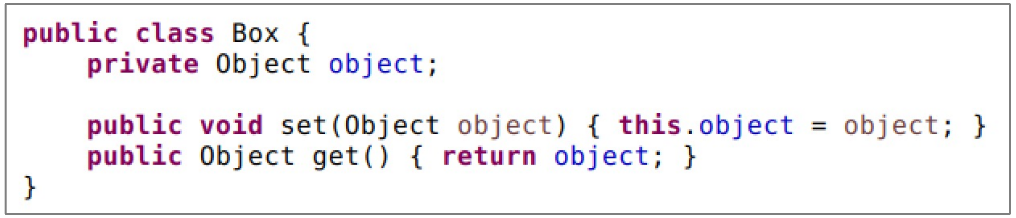
With Generics

Generic Types

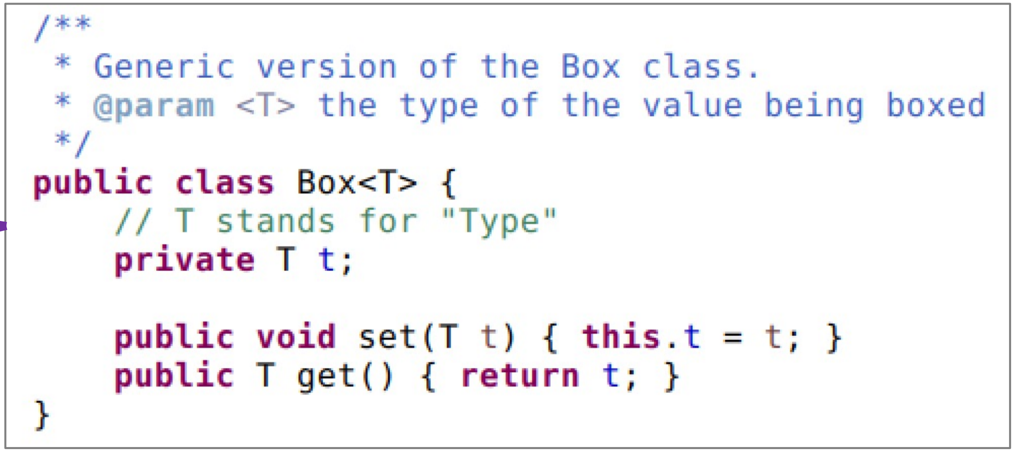
- A generic type is a generic class or interface that is parameterized over types.
- A generic class is defined with the following format:

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

- The most commonly used type parameter names are:
  - 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 Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```



```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

- For example,

  Box<Integer> integerBox = new Box<Integer>();

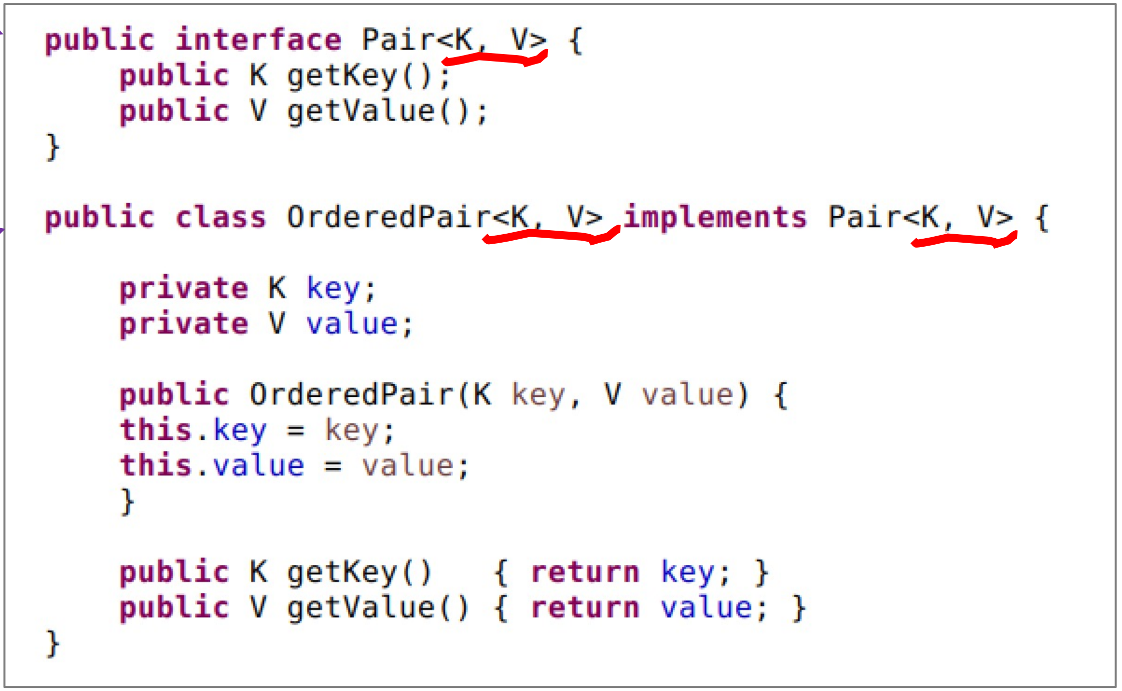
  OR

  Box<Integer> integerBox = new Box<>();

Multiple Type Parameters

- A generic class can have multiple type parameters.

- For example, the generic OrderedPair class, which implements the generic Pair interface



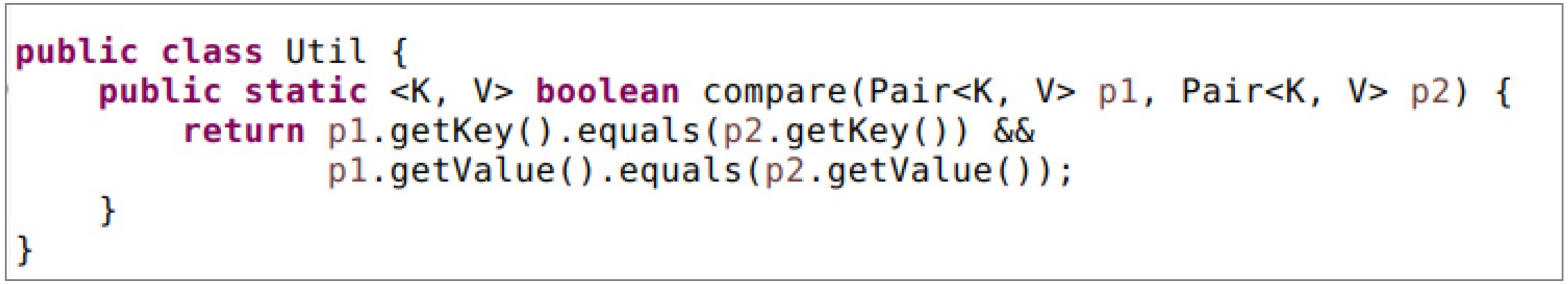
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
    this.key = key;
    this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

- Usage examples,

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
... ...
OrderedPair<String, Integer> p1 = new OrderedPair<>("Even", 8);
OrderedPair<String, String>  p2 = new OrderedPair<>("hello", "world");
... ...
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

Generic Methods

Generic methods are methods that introduce their own type parameters.



```
public class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

The complete syntax for invoking this method would be:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean  same = Util.<Integer, String>compare(p1, p2);
```

The type has been explicitly provided, as shown above.
Generally, this can be left out and the compiler will **infer** the **type** that is needed:

```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean   same = Util.compare(p1, p2);
```

Collections in Java

Collections in Java

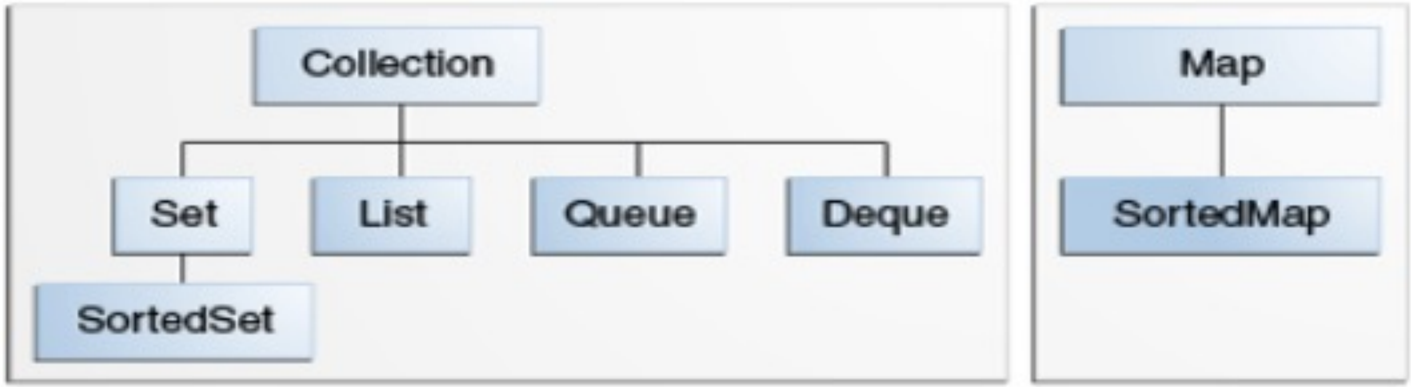
A collections framework is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.

All collections frameworks contain the following:

- *Interfaces*: allows collections to be manipulated independently of the details of their representation.
- *Implementations*: concrete implementations of the collection interfaces.
- *Algorithms*: the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
  - The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

Core Collection Interfaces:

- The core collection interfaces encapsulate different types of collections
- The interfaces allow collections to be manipulated independently of the details of their representation.



The core collection interfaces.

The Collection Interface

- A Collection represents a group of objects known as its elements.
- The Collection interface is used to pass around collections of objects where maximum generality is desired.
- For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument.
- The Collection interface contains methods that perform basic operations, such as
  - int **size**(),
  - boolean **isEmpty**(),
  - boolean **contains**(Object element),
  - boolean **add**(E element),
  - boolean **remove**(Object element),
  - Iterator<E> **iterator**(),
  - … … **many more** …

More at : https://docs.oracle.com/javase/tutorial/collections/**interfaces/collection.html**

Collection Implementations

- The general purpose implementations are summarized in the following table:

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Implemented Classes in the Java Collection, Read their APIs.

- Overview of the *Collections Framework* at the following page:

https://docs.oracle.com/javase/8/docs/technotes/guides/**collections/overview.html**

Wrappers for the Collection classes

- https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html

Demo: Collections Framework

Demo …

End