326.041 (2015S) – Practical Software Technology

(Praktische Softwaretechnologie)

Generic Types, Collections

Alexander Baumgartner
Alexander.Baumgartner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Motivating Example

A pair of strings:

```
1  public class Pair {
2      private String first;
3      private String second;
4
5      public Pair(String first, String second) {
6          this.first = first;
7          this.second = second;
8      }
9
10     public String getFirst() { return first; }
11     public String getSecond() { return second; }
12 }
```

Motivating Example

A pair of integers:

```
1  public class Pair {
2      private Integer first;
3      private Integer second;
4
5      public Pair(Integer first, Integer second) {
6          this.first = first;
7          this.second = second;
8      }
9
10     public Integer getFirst() { return first; }
11     public Integer getSecond() { return second; }
12 }
```

Motivating Example

A pair of one string and one integer:

```
1  public class Pair {
2      private String  first;
3      private Integer second;
4
5      public Pair(String first, Integer second) {
6          this.first = first;
7          this.second = second;
8      }
9
10     public String  getFirst() { return first; }
11     public Integer getSecond() { return second; }
12 }
```

Motivating Example

A pair of objects:

```
1  public class Pair {
2      private Object first;
3      private Object second;
4
5      public Pair(Object first, Object second) {
6          this.first = first;
7          this.second = second;
8      }
9
10     public Object getFirst() { return first; }
11     public Object getSecond() { return second; }
12 }
```

Motivating Example

A pair of objects:

```
1  public static void main(String[] args) {
2      Pair p = new Pair("Age", 22);
3      String propertyName = (String) p.getFirst();
4      int propertyValue = (Integer) p.getSecond();
5      ...
6  }
```

- p.getFirst() returns an Object.
- p.getSecond() returns an Object.
- Explicit casts are needed.
  - Complicated.
  - Error prone.

Alexander.Baumgartner@risc.jku.at

Classification

- A **type variable** is an unqualified identifier.
- A (abstract) **class** is generic if it declares one or more type variables.
- An **interface** is generic if it declares one or more type variables.
- A **method** is generic if it declares one or more type variables.
- A **constructor** is generic if it declares one or more type variables.
  A constructor can be declared as generic, independently of whether the class that the constructor is declared in is itself generic.

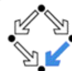
Generic Class

A **generic class** is defined with the following format:

```
1  ... class ClassName<T1, T2, ..., Tn> {
2        ...
3  }
```

- It specifies the **type variables** (parameters) T1, T2, . . . , and Tn.

A Generic Implementation of Pair



A generic pair of arbitrary typed values:

```
1  public class Pair<T1, T2> {
2      private T1 first;
3      private T2 second;
4
5      public Pair(T1 first, T2 second) {
6          this.first = first;
7          this.second = second;
8      }
9      public T1 getFirst() { return first; }
10     public T2 getSecond() { return second; }
11 }
```

- T1 and T2 are **type variables**.
- Use single, uppercase letters for type variable names, possibly followed by a single digit.
  - E – Element N – Number
  - K – Key V – Value
  - T – Type S,U,V,T1,T2 etc. – 2nd, 3rd,... types

Using the Generic Class Pair I

The generic class can be **typed as needed**:

```
1 public static void main(String[] args) {
2     Pair<String, Integer> p = new Pair<>("Age", 22);
3     String propertyName = p.getFirst();
4     int propertyValue = p.getSecond();
5     ...
6 }
```

- p.getFirst() returns a String.
- p.getSecond() returns an Integer.
- The types are declared as needed.
  - No casts.
  - Type safe.

Using the Generic Class Pair II

The generic class can be **typed as needed**:

```
1 public static void main(String[] args) {
2     Pair<String, Integer> p = new Pair<>("Age", 22);
3     String propertyName = p.getFirst();
4     int propertyValue = p.getSecond();
5     ...
6 }
```

- The diamond <> may be used if the compiler can infer the type arguments. Above Pair<> is a shortcut for Pair<String, Integer>.
- A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section.
- All of these parameterized types share the same class at runtime.

Nesting Generic Types

- You can also substitute a type parameter (i.e., T1 or T2) with a parameterized type (i.e., Pair<Integer, Integer>).
- For example, using the Pair<T1, T2> example:

```
1 Pair<String, Pair<Integer, Integer>> p
2     = new Pair<>("LifespanMinMax", new Pair<>(5,8));
```

Generic Methods

- **Generic methods** introduce their own type parameters:

```
public class Util {
    public static <T, U> boolean compare(
                Pair<T, U> p1, Pair<T, U> p2) {
        return p1.getFirst().equals(p2.getFirst()) &&
            p1.getSecond().equals(p2.getSecond());
    }
}
```

- The syntax for invoking this method would be:

```
Pair<String, Integer> p1 = new Pair<>("Age", 22);
Pair<String, Integer> p2 = new Pair<>("Age", 22);
Util.<String, Integer>compare(p1, p2);
```

- In most cases, the compiler can infer the type. It suffices to write:

```
Util.compare(p1, p2);
```

- Generic constructors work in the same way.

The Wildcard ?

- You can use the wildcard character ? to relax the restrictions on a variable.
- For instance:

```
public static int secondInt(Pair<?, ? extends Number> p){
    return p.getSecond().intValue();
}
```

- The first value can be of any type (unbounded wildcard).
- The second value can be of any subtype of Number (upper bound wildcard).

Raw Types

- You can (but **should never**) **omit** the type arguments.
- Writing

```
1 Pair p = new Pair("Age", 22);
```

is similar (but not the same) to writing

```
1 Pair<?, ?> p = new Pair<>("Age", 22);
```

Bounded Type Parameters

- Use keyword **extends** to define an **upper bound** for a type variable. E.g. A box which contains a number (Integer, Double, BigDecimal,..):

```
1 public class Box<T extends Number> { ...
```

- A type variable can have multiple bounds:

```
1 public class D <T extends A & B & C> { ...
```

- If one of the bounds is a class, it must be specified first. (In the above example, A could be a class)
- Use the wildcard ? and the keyword **super** to define a **lower bound** for a type variable. E.g. <? super Integer> allows super classes of Integer, i.e. Integer, Number, and Object (lower bound wildcard).
- You can specify either an upper bound or a lower bound, but not both.

Generic Classes and Subtyping

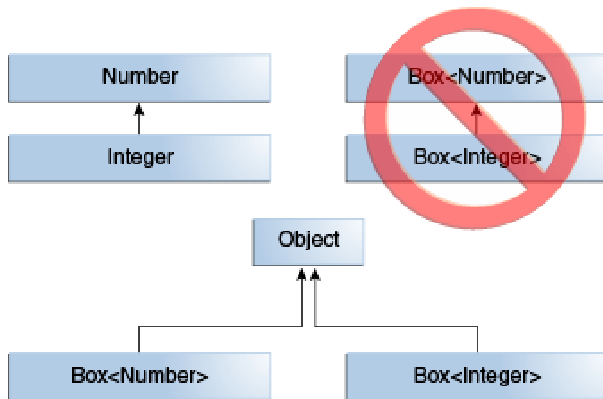


Figure: Box<Integer>is not a subtype of Box<Number>.

Generic Classes and Subtyping

- As an example of subtyping we use the class ArrayList:
- ArrayList<E> implements List<E>.
- List<E> extends Collection<E>.

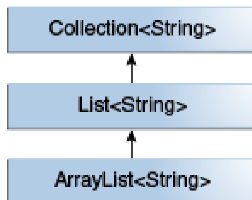


Figure: ArrayList<String>is a subtype of List<String>and Collection<String>.

Arrays vs. Collections

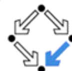
- An array is a collection of data with **fixed size**.

```
1 String[] someStrings = new String[5];
2 someStrings[0] = "Text 1";
3 someStrings[1] = "Text 2";
4 ...
5 someStrings[4] = "Text 5";
```

- A collection is an object that groups multiple elements into a single unit. Collections are of **variable size**.

```
1 List<String> someStrings = new ArrayList<>();
2 someStrings.add("Text 1");
3 someStrings.add("Text 2");
4 ...
```

What Is a Collections Framework?



- **Interfaces**: Define the behavior independently of the details of their implementation (representation).
- **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces (see class Collections).

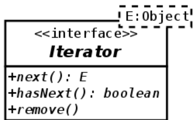
The JDK provides a collections framework.
The Standard Template Library (STL) in C++.

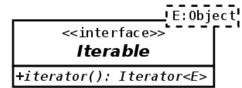
Advantages of the Collections

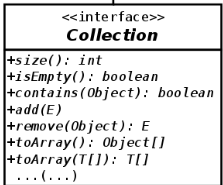
- **Reduces programming effort**: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level data structures.
- **Speed and quality**: The Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.
- **Abstraction / Interoperability**: Different implementations use the same interfaces.
- **Reusablity**: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

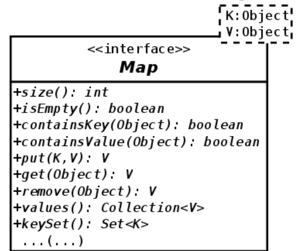
Overview of Collections Interfaces in Java

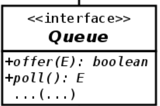
Implementation of Iterable allows "foreach" statement.

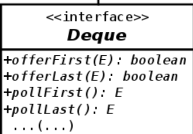


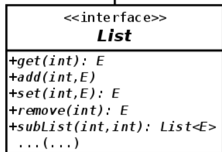


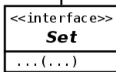


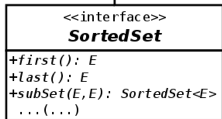


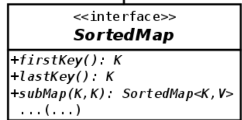












The Core Collection Interfaces

- **Collection**: Root of the collection hierarchy. Represents a group of objects known as its elements. No direct implementations of this interface.
- **Queue**: A collection used to hold multiple elements prior to processing. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner.
- **List**: An ordered collection (sequence). Can contain duplicate elements. (Dynamically resizable array.)
- **Set**: A collection that cannot contain duplicate elements.
- **Map**: An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value (abstraction of functions).

Example: Command Line Arguments

- Remove duplicate command line arguments.
- Sort command line arguments.

```
1 public static void main(String[] args) {
2     SortedSet<String> words = new TreeSet<>();
3     for (String arg : args)
4         words.add(arg);
5
6     for (String word : words)
7         System.out.println(word);
8 }
```

Bulk Operations of Collections

- **containsAll**: Test whether a Collection contains all the elements of another Collection.
- **addAll**: Adds all the elements of another Collection.
- **removeAll**: Removes all the elements which are also contained in the specified Collection.

```
1 c.removeAll(Collections.singleton(null));
```

- **retainAll**: Retains only those elements which are also contained in the specified Collection.

```
1 c.retainAll(Arrays.asList(2, 4, 6, 8));
```

- **clear**: Removes all elements from the Collection.
- **toArray**: Returns an array which contains all the elements.

```
1 Object[] a = c.toArray();
2 String[] a = c.toArray(new String[0]);
```

Range-View Operations of Lists

- **subList** generates a range-view of a given list.
- Operations are performed on the original list.
- Removing a range of elements from a List:

```
1  list.subList(fromIndex, toIndex).clear();
```

- Similar idioms can be constructed to search for an element in a range:

```
1  fromIdx += list.subList(fromIdx, toIdx).indexOf(obj);
```

fromIdx points to the first occurrence of obj within the given range.

Utility Classes

- **java.util.Arrays**:
  - Fill an array with values — Arrays.fill(array, value);
  - Sort an array — Arrays.sort(array);
  - Search in a sorted array — Arrays.binarySearch(array, value);
  - ...
- **java.util.Collections**:
  - Replace all the values — Collections.fill(list, value);
  - Sort a list — Collections.sort(list);
  - Search in a sorted list — Collections.binarySearch(list, value);
  - Searching for the maximal element in a collection
  - ...
- For sorting, the elements must either **implement the Comparable interface**, **or** you must **provide a Comparator** implementation.

Empty and Singleton Collections

- Creating **empty** collections:
  - Set<String> s = Collections.emptySet();
  - List<String> l = Collections.emptyList();
  - Map<String, String> m = Collections.emptyMap();
- Creating **singleton** collections:
  - Set<String> s = Collections.singleton("value");
  - List<String> l = Collections.singletonList("value");
  - Map<String, String> m = Collections.singletonMap("key", "value");

Exercise

- Refactoring Ecosystem: Improve the Ecosystem simulation such that:
  - The river does not contain any specific knowledge of a species.
  - There are no if statements inside of the class River which determine a certain species.
  - Adding another species (with similar behavior) must not concern any of the existing interfaces and classes. (Excluding your test cases.)
- Implement another species of your choice (e.g. Zombies) without changing any other class or interface. (Excluding your test cases.)
- Test the ecosystem including your new species.

See the guidance for this exercise on the Moodle page.