Java Collections, Generic Methods and Classes

CS102A Lecture 14

James YU
yujq3@sustech.edu.cn

Department of Computer Science and Engineering
Southern University of Science and Technology

Dec. 14, 2020

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Objectives

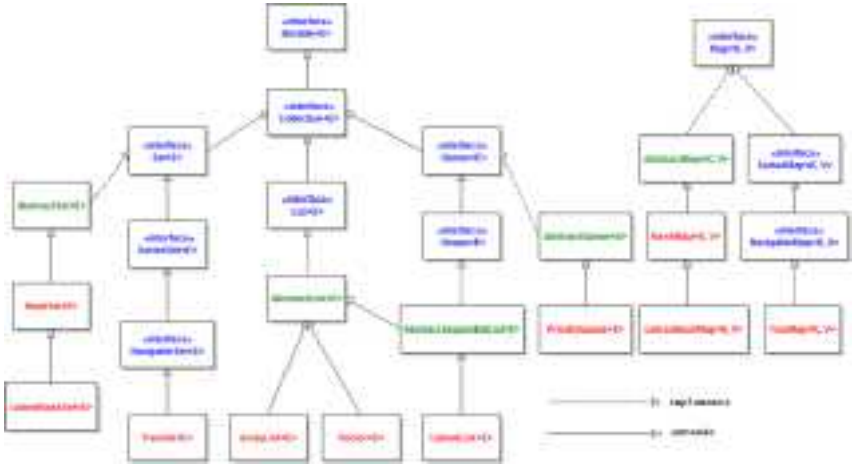
- Java collections framework
- Three common types of collections
- `ArrayList`
- `HashMap`
- Motivation of generic methods
- Declare and use generic methods
- Declare and use generic classes

Java Collections Framework (JCF)

- JCF is a set of classes and interfaces that implement reusable data structures (or containers) to help **group and manage related objects**.
- Similar to arrays, collections hold references to objects that can be managed as a group (one object represents a group of objects).
- Unlike arrays, collections do not need to be assigned a certain capacity when instantiated. **Their size can grow and shrink automatically** when objects are added or removed.
- Unlike arrays, **collections cannot hold primitive type elements** (e.g., int), they can only hold object references (arrays can do both).

JCF class hierarchy

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



The `Collection` interface

- `java.util.Collection` is the root interface in the collection hierarchy.
- Methods declared (**not implemented**) in `Collection`:
  - `add`, `addAll` (adding elements);
  - `remove`, `removeAll`, `removeIf`, `clear` (removing elements);
  - `contains`, `containsAll` (checking the existence of elements);
  - `size` (returning the number of elements);
  - `toArray` (returning an array containing all elements in the collection);
  - ...

`List` and `Set`

- `Collection` has two important offspring: `List` and `Set`.
- A `List` is an **ordered** collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list. `List`s typically **allow duplicate elements**.
- A `Set` is a collection that **contains no duplicate elements**. This interface models the mathematical set abstraction.

The `Map` interface

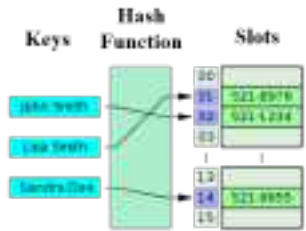
- A `Map` is an object that maps **keys** to **values**, or is a collection of attribute-value pairs.
  - A map of error codes and their descriptions (404 $\rightarrow$ Not found).
  - A map of zip codes and cities (518000 $\rightarrow$ Shenzhen).

The `Map` interface

- `java.util.Map` does not extend `java.util.Collection`. It is not considered to be a true collection and has its own branch in the JCF.
- Methods declared (**not implemented**) in `Map`:
  - `put(K key, V value)`, `putAll` (associating keys and values);
  - `remove(Object key)`, `clear` (removing mappings);
  - `containsKey`, `containsValue` (checking the existence of keys and values);
  - `keySet`, `values` (returning a collection view of the keys and values);
  - `size` (returning the number of key-value mappings).
  - ...

HashMap

- `HashMap` (hash table) is a data structure that can map keys to values. It is a concrete implementation of the `Map` interface.
- It uses a *hash function* to compute an index into an array of slots, from which the desired value can be found (very efficient).
- A hash function is any function that can map data of arbitrary size to data of fixed size. Well-defined hash functions have low chances of collisions (mapping two different keys to the same hash values).



Creating a HashMap

- Use interface name to declare variable:

```
1 Map<Integer, String> mapHttpErrors = new HashMap<>();
2 mapHttpErrors.put(400, "Bad Request"); // key: Integer; Value: String
3 mapHttpErrors.put(301, "Moved Permanently");
4 mapHttpErrors.put(404, "Not Found");
5 mapHttpErrors.put(500, "Internal Server Error");
6 System.out.println(mapHttpErrors);
```

```
{400=Bad Request, 404=Not Found, 500=Internal Server Error, 301=Moved
 Permanently}
```

Getting a value associated with a key

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```
1 String status301 = mapHttpErrors.get(301);
2 System.out.println("301: " + status301);
```

```
301: Moved Permanently
```

Checking existence of keys and values

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
if (mapHttpErrors.containsKey(301)) {
  System.out.println("Found key");
}

if (mapHttpErrors.containsValue("Bad Request")) {
  System.out.println("Found value");
}
```

```
Found key
Found value
```

Removing a mapping

```java
String removedValue = mapHttpErrors.remove(500);

if (removedValue != null) {
  System.out.println("Removed value: " + removedValue);
}
```

```
Removed value: Internal Server Error
```

Update the value of a pair

```java
Map<Integer, String> mapHttpErrors = new HashMap<>();
mapHttpErrors.put(500, "Not found");
System.out.println(mapHttpErrors);
mapHttpErrors.put(500, "Internal Server Error");
System.out.println(mapHttpErrors);
```

```
{500=Not found}
{500=Internal Server Error}
```

- Simply call the `put` method: If the map previously contained a mapping for the key, the old value is replaced by the specified value.

Recall method overloading

- A language feature that allows a class to have multiple methods with the same name, but different parameter lists.

```java
public static void printArray(Integer[] array) {
    for (Integer element : array) System.out.printf("%s ", element);
    System.out.println();
}
public static void printArray(Double[] array) {
    for (Double element : array) System.out.printf("%s ", element);
    System.out.println();
}
public static void printArray(Character[] array) {
    for (Character element : array) System.out.printf("%s ", element)
        ;
    System.out.println();
}
```

Using overloaded methods

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
public static void main(String[] args) {
  Integer[] integerArray = { 1, 2, 3, 4, 5, 6 }; // autoboxing
  Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 }; // autoboxing
  Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' }; // autoboxing
  System.out.print("integerArray contains: ");
  printArray(integerArray);
  System.out.print("doubleArray contains: ");
  printArray(doubleArray);
  System.out.print("characterArray contains: ");
  printArray(characterArray);
}
```

```
integerArray contains: 1 2 3 4 5 6
doubleArray contains: 1.1 2.2 3.3 4.4 5.5
characterArray contains: H E L L O
```

Looks good, but wait ...

```java
public static void printArray(Integer[] array) {
    for (Integer element : array) System.out.printf("%s ", element);
    System.out.println();
}
public static void printArray(Double[] array) {
    for (Double element : array) System.out.printf("%s ", element);
    System.out.println();
}
public static void printArray(Character[] array) {
    for (Character element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

- These methods are identical except the data type part. If the input is `Long[]` or `String[]`, shall we continue the overloading?

A better design with generics

- If the operations performed by several overloaded methods are identical for each argument type, the overloaded methods can be more compactly coded using a generic method.

```
1 public static <T> void printArray(T[] array) {
2   for (T element : array) System.out.printf("%s ", element);
3   System.out.println();
4 }
```

- Type-parameter section: one or more type parameters delimited by <>.
- Each type parameter parameterizes the data types that can be used in the method (in the above example, T can be used anywhere a data type name is expected).
- T is used for the generic name by convention.

Declaring generic methods

- Generic methods can be declared like any other normal methods.
- Type parameters can represent only reference types (not primitive types).

```
public static void printArray(Double[] array) {
    for (Double element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

```
public static <T> void printArray(T[] array) {
    for (T element : array) System.out.printf("%s ", element);
    System.out.println();
}
```

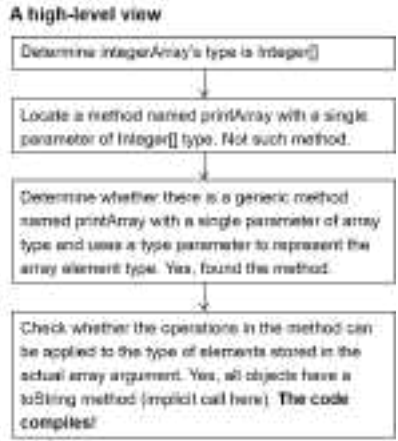
Using generic methods

```java
public static void main(String[] args) {
  Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
  Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };
  Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
  System.out.print("integerArray contains: ");
  printArray(integerArray);
  System.out.print("doubleArray contains: ");
  printArray(doubleArray);
  System.out.print("characterArray contains: ");
  printArray(characterArray);
}
```

```
integerArray contains: 1 2 3 4 5 6
doubleArray contains: 1.1 2.2 3.3 4.4 5.5
characterArray contains: H E L L O
```

How does compiler work here?

```java
public class GenericMethodExample {
  public static void main(String[] args) {
    Integer[] integerArray = ...;
    Double[] doubleArray = ...;
    Character[] characterArray = ...;
    printArray(integerArray);
    printArray(doubleArray);
    printArray(characterArray);
  }
  public static <T> void printArray(T[]
      array) {
    for (T element : array)
      System.out.printf("%s ", element);
    System.out.println();
  }
}
```



Under the hood: Erasure

- When the compiler translates generic method `printArray` into Java bytecodes, it **removes the type-parameter section** and **replaces the type parameters with actual types**. This process is known as *erasure*.
- By default, all generic types are replaced with type `Object`.
- The compiled version of `printArray` is shown below (we show source code instead of bytecodes)

```
1 public static void printArray(Object[] array) {
2     for (Object element : array) System.out.printf("%s ", element);
3     System.out.println();
4 }
```

Benefits of generic methods

- In the earlier example, it seems that using generic methods is the same as using `Object` array as parameter of `printArray` (like the code below).

```
1  public static void main(String[] args) {
2    Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
3    Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5 };
4    Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
5    System.out.print("integerArray contains: ");
6    printArray(integerArray);
7    System.out.print("doubleArray contains: ");
8    printArray(doubleArray);
9    System.out.print("characterArray contains: ");
10   printArray(characterArray);
11 }
12 public static void printArray(Object[] array) {
13   for (Object element : array) System.out.printf("%s ", element);
14   System.out.println();
15 }
```

Benefits of generic methods

```java
public static Object simplyReturn(Object o) {
  return o;
}
public static void main(String[] args) {
  String s = simplyReturn("hello");
}
```

- The compiler sees that the method return type is `Object`, assigning a reference of `Object` to a `String` variable is illegal, so a compilation error will occur.
- Programmers need to perform explicit type cast: `(String) simplyReturn("hello")`, which may generate `ClassCastExceptions` if the cast fails.

Benefits of generic methods

```java
public static <T> T simplyReturn(T o) {
    return o;
}
public static void main(String[] args) {
    String s = simplyReturn("hello");
}
```

- With the generic method, the compiler will perform careful type checking and infer the return type is `String` when the actual argument's type is `String` and inserts type cast automatically (such cast will never throw `ClassCastException`, guaranteed to be safe).
- Therefore, the code can be successfully compiled and is more **type safe**. The benefits become obvious when the return type is also parameterized.

Bounded type parameter

```java
public static <T> T simplyReturn(T o) {
    return o;
}
```

- In generic methods like the above, all reference types up to `Object` can be are passed to the type parameter (we say `Object` is an upper bound).
- There are times when you want to restrict the types that are allowed to be passed to a type parameter, e.g., a method that operates on numbers might only want to accept instances of `Number` or its subclasses.
- *Bounded type parameter*s are useful in such cases.

Bounded Type Parameter

- To declare a bounded type parameter, simply list the type parameter's name followed by the `extends` keyword and an upper bound.
  - Here, `extends` is used in a general sense to mean either "`extends`" as in classes or "`implements`" as in interfaces.

```
1 public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
2   T max = x;
3   if (y.compareTo(max) > 0) max = y;
4   if (z.compareTo(max) > 0) max = z;
5   return max;
6 }
```

Example

```
public static void main(String[] args) {
  System.out.printf("Maximum of %d, %d and %d is %d\n", 3, 4, 5,
    maximum(3, 4, 5));
  System.out.printf("Maximum of %.1f, %.1f and %.1f is %.1f\n", 6.6,
    8.8, 7.7, maximum(6.6, 8.8, 7.7));
  System.out.printf("Maximum of %s, %s and %s is %s\n", "pear",
    "apple", "orange", maximum("pear", "apple", "orange"));
}
```

- `Integer`, `Double` and `String` all implement the `Comparable` interface, so can be passed to the type parameter.

Compiler's view

```java
// Erasure: replacing the type parameter T with the upper bound
    Comparable
public static Comparable maximum(Comparable x, Comparable y,
    Comparable z) {
  Comparable max = x;
  if (y.compareTo(max) > 0) max = y;
  if (z.compareTo(max) > 0) max = z;
  return max;
}
```

- When encountering method calls, infer the return type and insert explicit casts (the compiler guarantees that the cast will never throw `ClassCastException`):
  - `maximum(3, 4, 5)` → `(Integer) maximum(3, 4, 5)`.
  - `maximum(6.6, 8.8, 7.7)` → `(Double) maximum(6.6, 8.8, 7.7)`.
  - `maximum("pear", "apple", "orange")` → `(String) maximum("pear", "apple", "orange")`.

Generic classes

- The concept of many data structures, such as a stack, can be understood independently of the element type it manipulates.
- *Generic class*es provide a means for describing the concept of a stack (or any other classes) in a type independent manner.
- We can then instantiate type-specific objects of the generic classes. This makes software reusable (**program in general, not in specifics**).

We've seen generic classes

- `ArrayList<E>` is a generic class, where `E` is a placeholder (type parameter) for the *type of elements* that you want the `ArrayList` to hold.

Declaring a generic class

- A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a *type-parameter section*.
- The type-parameter section can have one or more type parameters separated by commas.
- Generic classes are also known as *parameterized classes*.
- In a generic class, type parameters can be used anywhere a type is expected (e.g., when declaring parameters, return types, defining variables, ...").

A generic Stack class

南方科技大学 SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

```java
public class Stack<T> {
  private ArrayList<T> elements; // use an ArrayList for the stack
  public Stack() {    this(10);    }
  public Stack(int capacity) {
    int initCapacity = capacity > 0 ? capacity : 10;
    elements = new ArrayList<T>(initCapacity);
  }
  public void push(T value) {
    elements.add(value);
  }
  public T pop() {
    if(elements.isEmpty())
      throw new EmptyStackException("Stack is empty, cannot pop");
    return elements.remove(elements.size() - 1);
  }
}
```

Test the generic `Stack` class

```java
public static void main(String[] args) {
  Stack<Double> doubleStack = new Stack<Double>(5);
  Stack<Integer> integerStack = new Stack<Integer>();

  doubleStack.push(1.2);
  Double value = doubleStack.pop();
  System.out.println(value);

  integerStack.push(1);
  integerStack.push(2);

  while(true) {
    Integer i = integerStack.pop();
    System.out.println(i);
  }
}
```

Compiler's view

- *Erasure* (similar to generic methods): Replacing all type parameters with `Object` or their bounds if the type parameters are bounded.
- The produced bytecodes contain only ordinary classes, interfaces, and methods, i.e., no generics at the bytecode level.

```
public class Stack {
  private ArrayList<Object> elements;
  public Stack() {    this(10);    }
  public Stack(int capacity) {...
    elements = new ArrayList<Object>(initCapacity);
  }
  public void push(Object value) { ... }
  public Object pop() { ... }
}
```

Compiler's view

- The compiler will insert type casts if necessary to preserve type safety.

```
1 Stack<Double> doubleStack = new Stack<Double>(5);
2 doubleStack.push(1.2);
3 Double value = doubleStack.pop();
```

```
1 Stack doubleStack = new Stack(5);
2 doubleStack.push(1.2);
3 Double value = (Double) doubleStack.pop();
```

Let's test out understanding

- **Q1**: Will the compiler successfully compile the following code?

```
1 String s = "hello world";
2 Object obj = s;
```

- It is safe to assign `s` (of type `String`) to `obj` (of type `Object`) because an instance of a subclass (subtype) is also an instance of a superclass (supertype).
- "Safe" means any operations that can be done via the reference `obj` are also allowed to be done via the reference `s`.

A more difficult question

- **Q2**: Will the compiler successfully compile the following code?

```
1 ArrayList<String> ls = new ArrayList<String>();
2 List<String> ls2 = ls;
```

- It is safe to assign `ls` to `ls2` because an `ArrayList` of `String` is also a `List` of `String`.
- Any operations that can be done via the reference `ls2` can also be done via the reference `ls`.

The hardest question about generics

- **Q3**: Will the compiler successfully compile the following code?

```
1 List<String> ls = new ArrayList<String>();
2 List<Object> lo = ls;
```

- This boils down to the question: **is a `List` of `String` a `List` of `Object`**?
- What if we ask the safety question: is it true that any operations that can be done via the reference `lo` can also be done via `ls`?

Let's do some analysis

- As a reference of type `List<Object>`, `lo` can be used for the following operation:

```
1 lo.add(new Double());
```

- However, we cannot perform the same operation via the reference `ls` because it is of type `List<String>`:

```
1 List<String> ls = new ArrayList<String>();
2 List<Object> lo = ls; // type mismatch
```

Further analysis from compiler's view

- If the compiler allows assigning `ls` to `lo`, then the code

```
1 List<String> ls = new ArrayList<String>();
2 List<Object> lo = ls;
3 lo.add(new Double(0.0));
4 String s = ls.get(0);
```

  will be compiled into the following form:

```
1 List ls = new ArrayList();
2 List lo = ls;
3 lo.add(new Double(0.0));
4 String s = (String) ls.get(0);
```

- `ClassCastException`.

General rule

- If `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is not the case that `G<Foo>` is a subtype of `G<Bar>`.
- This is probably the hardest thing one needs to learn about generics, because it goes against our intuitions.
- Suppose we want to write a method to handle all kinds of `collections`:

```
1 public static void printCollection(Collection<Object> c) {
2     for(Object e : c) System.out.println(c);
3 }
```

```
1 Collection<String> strs = new ArrayList<String>();
2 printArray(strs); // is this call ok?
```

- Apply the rule, and we will know that `Collection<String>` is not a subtype of `Collection<Object>`, so the code cannot compile.

Wildcards

- `Collection<?>` (pronounced “collection of unknown”), that is, a collection whose element type matches anything:

```
// We can call this method with any kind of collection
public static void printCollection(Collection<?> c) {
    for(Object e : c) System.out.println(c);
}
```

- Note that we can still read elements from c and give them the type “Object”, since whatever the actual type of the collection element is, it is a subtype of `Object`.
- Please note that the compiler will not have the same erasure operation on `?` like the one we introduced before.

Wildcards

- Let say you want to sum up all the numbers.

```java
public static void main(String[] args) {
  ArrayList<Integer> intlist = new ArrayList<Integer>();
  ArrayList<Double> doublelist = new ArrayList<Double>();
  ...
  System.out.println(sum(intlist));
  System.out.println(sum(doublelist));
}

public static double sum(ArrayList< ? Extends Number> list) {
  double total = 0;
  for ( Number  element : list)
      total += element.doubleValue();
  return total;
}
```

Wildcards

- Java allows the following code:

```
1 Collection<?> c = new ArrayList<String>();
```

- However, it does not allow you to add arbitrary objects to `c`, the “collection of unknown”.

```
1 c.add(new String()); // No good
2 c.add(new Object()); // No good
```

- In the declaration of `Collection<E>`, the add method takes arguments of type `E`, the type of the collection’s element. Here, the actual type parameter is `?` (unknown type), so anything that we pass to add must be the type of the unknown type. That means we cannot pass anything in except `null`.