EE3206
Java Programming and Applications

Lecture 5
Generics and Collections

Van Ting, Department of Electrical Engineering, City University of Hong Kong

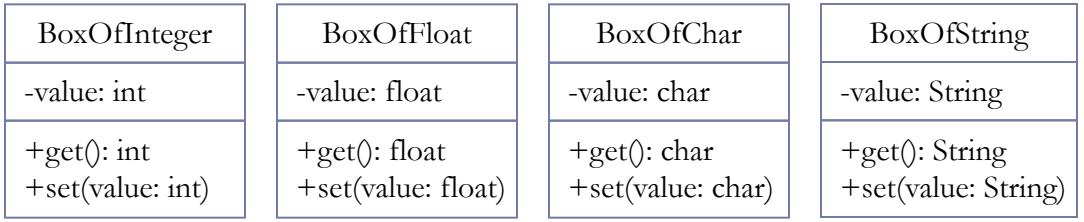
Intended Learning Outcomes

- To understand the advantage of being generic
- To learn the syntax for generic class and generic method
- To create generic class with multiple type parameters
- To understand type inference
- To understand the architecture of Java Collections Framework
- To apply the data structures and algorithms in JCF to solve problems

Van Ting, Department of Electrical Engineering, City University of Hong Kong

A Box of XXX

- Let say we need a box to store some data…



. . . etc

- Because the Box classes above are bind to a particular value type, one may need to create many other implementations for the same set of Box operations just for catering the need of other types of value.
- Namespace is overwhelmed by different value types of the same Box
- Lots of redundant codes
- More effort is needed to maintain multiple implementations of the Box
- Cannot be used to store custom type of objects in future (e.g. *MyCircle*)

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Non-Generic Type - A Simple Box Class

- The previous problem can be partially addressed by using *polymorphism*.
- ***Object*** class is the default parent of any Java classes
- Begin by examining a non-generic Box class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
    private Object object;

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

- Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types.
- However, due to lack of type information, there is no way to verify at compile time, how the class is used. One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Generic Type - A Generic Version of the Box Class

- A *generic type* is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.
- A *generic class* is defined with the following format:

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

- The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.
- To update the Box class to use generics, you create a *generic type declaration* by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

```
public class Box<T> {
    // T stands for "Type"
    private T t;

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

- As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable. This same technique can be applied to create generic interfaces.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Generics

- In any nontrivial software project, bugs are simply a fact of life. Careful planning, programming, and testing can help reduce their pervasiveness, but somehow, somewhere, they'll always find a way to creep into your code. This becomes especially apparent as new features are introduced and your code base grows in size and complexity.

- Fortunately, some bugs are easier to detect than others. Compile-time bugs, for example, can be detected early on; you can use the compiler's error messages to figure out what the problem is and fix it, right then and there. Runtime bugs, however, can be much more problematic; they don't always surface immediately, and when they do, it may be at a point in the program that is far removed from the actual cause of the problem.

- Generics add stability to your code by making more of your bugs detectable at compile time.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Invoking and Instantiating a Generic Type

- To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

```
Box<Integer> integerBox;
```

- You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a *type argument* — Integer in this case — to the Box class itself.
- Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.
- An invocation of a generic type is generally known as a *parameterized type*.
- To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

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


Van Ting, Department of Electrical Engineering, City University of Hong Kong

Why Use Generics?

- In a nutshell, generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

- Code that uses generics has many benefits over non-generic code:

- **Stronger type checks at compile time.**
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Why Use Generics?

- **Elimination of casts.**
  The following code snippet without generics requires casting:

```
List list = new ArrayList(); // accept any Object instances
list.add("hello");
String s = (String) list.get(0);   // “hello” is returned as Object
```

- When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>(); // accept String instance only
list.add("hello");
String s = list.get(0); // no cast, “hello” is returned as String
```

- **Enabling programmers to implement generic algorithms.**
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- 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
- You'll see these names used throughout the Java SE API.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

The Diamond

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set of type arguments (<>) as long as the compiler can determine, or infer, the type arguments from the context.
- This pair of angle brackets, <>, is called *the diamond*. For example, you can create an instance of Box<Integer> with the following statement:

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


Van Ting, Department of Electrical Engineering, City University of Hong Kong

Multiple Type Parameters

- As mentioned previously, 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; }
}
```


Van Ting, Department of Electrical Engineering, City University of Hong Kong

Multiple Type Parameters

- The following statements create two instantiations of the OrderedPair class:

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

- The code, `new OrderedPair<String, Integer>`, instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively.

- You can also substitute a type parameter (i.e., K or V) with a parameterized type (i.e., List<String>). For example, using the OrderedPair<K,V> example:

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<>("primes", new Box<Integer>(...));
```

OrderedPair

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Generic Methods

- *Generic methods* are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

- The syntax for a generic method includes a type parameter, inside angle brackets, and appears before the method's return type.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Generic Methods

- The Util class includes a generic method, ***compare***, which compares two Pair objects:

the class itself not necessary be generic

```
public class Util {
    // Generic static method
    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());
    }
}

public class Pair<K, V> {

    private K key;
    private V value;

    // Generic constructor
    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    // Generic methods
    public void setKey(K key) { this.key = key; }
    public void setValue(V value) { this.value = value; }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```


Van Ting, Department of Electrical Engineering, City University of Hong Kong

Type Inference for Method Invocation

- 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 in bold. Generally, this can be left out and the compiler will infer the type (from the input arguments) 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);
```

- This feature, known as *type inference*, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.

Util

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Bounded Type

- There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of *Number* or its subclasses. This is what *bounded type parameters* are for.

- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is *Number*. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).
- For multiple bounds:
  - <T extends B1 & B2 & B3>
  - where B1 is a class or interface and following bounds must be interface.

```
public class Box<T> {

    private T t;

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

    public T get() {
        return t;
    }

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

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // error: this is still String!
    }
}
```

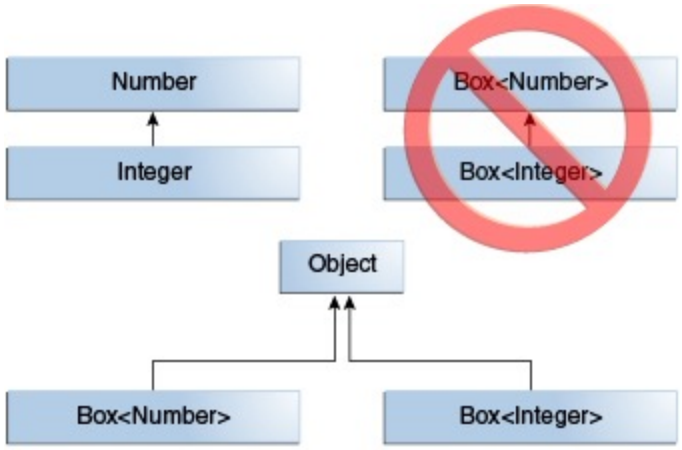

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Generics, Inheritance, and Subtypes

- You can perform a generic type invocation, passing *Number* as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with *Number*:

```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

- Now consider the following method:
  - `public void boxTest(Box<Number> n) { /* ... */ }`
- What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is `Box<Number>`. But what does that mean? Are you allowed to pass in `Box<Integer>` or `Box<Double>`, as you might expect?
- The answer is **NO**, because `Box<Integer>` and `Box<Double>` are not subtypes of `Box<Number>`.
- This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn.



Van Ting, Department of Electrical Engineering, City University of Hong Kong

Wildcards (?)

- In generic code, the question mark (?), called the *wildcard*, representing an unknown type, can be used to relax the restrictions on a variable.
- The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific).
- The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.
- Consider the following method, `printList` where its goal is to print a list of any type but it fails to achieve .

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

List<Integer> li = Arrays.asList(1, 2, 3);
List<String>  ls = Arrays.asList("one", "two", "three");
printList(li);     // compilation error
printList(ls);     // compilation error
```

UnboundedWildcards

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Wildcards (?)

- It prints only a list of Object instances. It cannot print `List<Integer>`, `List<String>`, `List<Double>`, and so on, because they are not subtypes of `List<Object>`. To write a generic `printList` method, use `List<?>` instead.
- When using wildcard, the type information is basically unknown, so there are two scenarios where an unbounded wildcard is a useful approach:
  - If you are writing a method that can be implemented by only using functionality provided in the ***Object*** class.
  - When the code is using methods in the generic class that don't depend on the type parameter. For example, ***List.size()*** or ***List.clear()***.

Name	Syntax	Meaning
Wildcard with upper bound	`? extends B`	Any subtype of `B`
Wildcard with lower bound	`? super B`	Any supertype of `B`
Unbounded wildcard	`?`	Any type

Van Ting, Department of Electrical Engineering, City University of Hong Kong

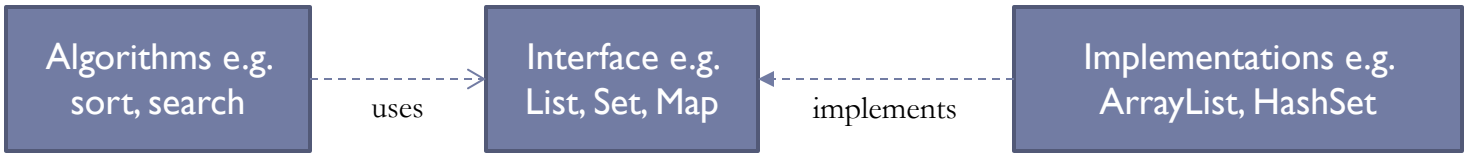
Java Collections Framework (JCF)

- Java counterpart of C++ Standard Template Library (STL)
- The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.
- A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.
- The classes and interfaces of the collections framework are in package *java.util*.
- Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.
- The collections framework was designed to meet several goals.
  - The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) are highly efficient.
  - The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
  - Extending and/or adapting a collection had to be easy.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

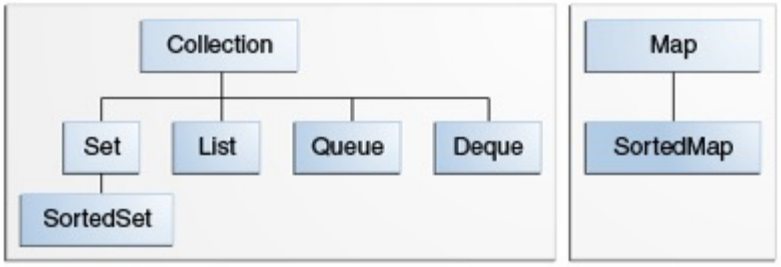
Java Collections Framework (JCF)

- Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **ArrayList, HashMap,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.
- A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:
  - **Interfaces:**
    These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
  - **Implementations, i.e., Classes:**
    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. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.



Van Ting, Department of Electrical Engineering, City University of Hong Kong

JCF Interfaces



- The Java Collections Framework hierarchy consists of two distinct interface trees:
  - The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set, List, and Queue — provide for more specialized collections.
  - The Set interface does **not allow duplicate** elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
  - The List interface provides for an ordered collection (**allows duplicate**), for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
  - The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
  - The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.
  - The second tree starts with the Map interface, which **maps keys and values** similar to a hashtable. Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.
- These interfaces allow collections to be manipulated independently of the details of their representation.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Interface Collection<E>

- `public interface Collection<E> extends Iterable<E>`

- **Basic operations:**
- public int size(); // Return number of elements
- public boolean isEmpty(); // Return true iff collection is empty
- public boolean add(E x); // Make sure collection includes x; return true if it has changed
- public boolean contains(Object x); // Return true iff collection contains x (uses method equals)
- public boolean remove(Object x); // Remove one instance of x from the collection; return true if collection has changed
- public Iterator<E> iterator(); // Return an Iterator that enumerates elements of collection
- **Array operations:**
- public Object[] toArray(); // Return a new array containing all elements of collection
- public <T> T[] toArray(T[] dest); // Return an array (using dest) containing all elements of this collection
- **Bulk Operations:**
- public boolean containsAll(Collection<?> c);
- public boolean addAll(Collection<? extends E> c);
- public boolean removeAll(Collection<?> c);
- public boolean retainAll(Collection<?> c);
- public void clear();

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Interface Iterator<E>

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator() method. The following is the Iterator interface.

- `public boolean hasNext();`
  - Return true if the enumeration has more elements
- `public E next();`
  - Return the next element of the enumeration
  - Throws NoSuchElementException if no next element
- `public void remove();`
  - Remove most recently returned element by next() from the underlying collection
  - Thros IllegalStateException if next() not yet called or if remove() already called since last next()
  - Throw UnsupportedOperationException if remove() not supported

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Iterators: How "for-each" works

- The notation of the enhanced for-loop aka for-each-loop:

```
for(E var: collection) { your code … }
```

  is syntactic sugar. It compiles into this "old code":

```
Iterator<E> _i= collection.iterator();
while (_i.hasNext()) {
   E var = _i.next();
    your code …
}
```

- The two ways of doing this are identical but the *for-each-loop* has nicer looking.

Van Ting, Department of Electrical Engineering, City University of Hong Kong

JCF Implementations

- **General-purpose implementations** are the implementations designed for everyday use. They are summarized in the table. The one in red color is the most commonly used implementation.

Interfaces	Hash table Implementations	Resizable array Implementations	Tree Implementations	Linked list Implementations	Hash table + Linked list Implementations
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- As a rule, you should be thinking about the interfaces, *not* the implementations. For the most part, the choice of implementation affects only performance. The preferred coding style is to choose an implementation when a Collection is created and to immediately assign the new collection to a variable of the corresponding interface type.
- In this way, the program does not become dependent on any added methods in a given implementation, leaving the programmer free to change implementations anytime that it is warranted by performance concerns or behavioral details.
  - `List<Integer> l = new ArrayList<Integer>();`
  - `Set<String> s = new HashSet<String>(64);`
  - `Map<String, Integer> m = new HashMap<String, Integer>();`

ArrayListDemo

HashSetDemo

HashMapDemo

Van Ting, Department of Electrical Engineering, City University of Hong Kong

JCF Algorithms

- The collection framework also provides polymorphic algorithms running on collections. All of them come from the *Collections* class (not *Collection* interface) and all take the form of static methods.
  - Sorting - *sort*
  - Shuffling - *shuffle*
  - Routine Data Manipulation
    - *reverse* — reverses the order of the elements in a List.
    - *fill* — overwrites every element in a List with the specified value. This operation is useful for reinitializing a List.
    - *copy* — takes two arguments, a destination List and a source List, and copies the elements of the source into the destination, overwriting its contents. The destination List must be at least as long as the source. If it is longer, the remaining elements in the destination List are unaffected.
    - *swap* — swaps the elements at the specified positions in a List.
    - *addAll* — adds all the specified elements to a Collection. The elements to be added may be specified individually or as an array.
  - Searching - *binarySearch*
  - Composition
    - *frequency* — counts the number of times the specified element occurs in the specified collection
    - *disjoint* — determines whether two Collections are disjoint; that is, whether they contain no elements in common
  - Finding extreme values – *min* and *max*

AlgorithmsDemo

Van Ting, Department of Electrical Engineering, City University of Hong Kong

Interface Comparator<E>

- What if you want to sort some objects that don't implement Comparable?
- What if you want to sort some objects in an order other than their natural ordering?
- To do either of these things, you'll need to provide a *Comparator* — an object that encapsulates an ordering. The *Comparator* interface lets us sort a given collection any number of different ways and can be used to sort any instances of any class (even classes we cannot modify).
- Like the *Comparable* interface, the *Comparator* interface consists of a single method.

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

  - The *compare* method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second.

Dog

Van Ting, Department of Electrical Engineering, City University of Hong Kong

References

- Generics
  - https://docs.oracle.com/javase/tutorial/java/generics/index.html
- Java Collection Framework
  - http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html
  - http://docs.oracle.com/javase/8/docs/api/java/util/List.html
  - http://docs.oracle.com/javase/8/docs/api/java/util/Set.html
  - http://docs.oracle.com/javase/8/docs/api/java/util/Map.html
  - https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Van Ting, Department of Electrical Engineering, City University of Hong Kong