Generics and Collections in Java

UNSW SYDNEY

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
- For example,

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

OR

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

```
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; }
}
```

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:

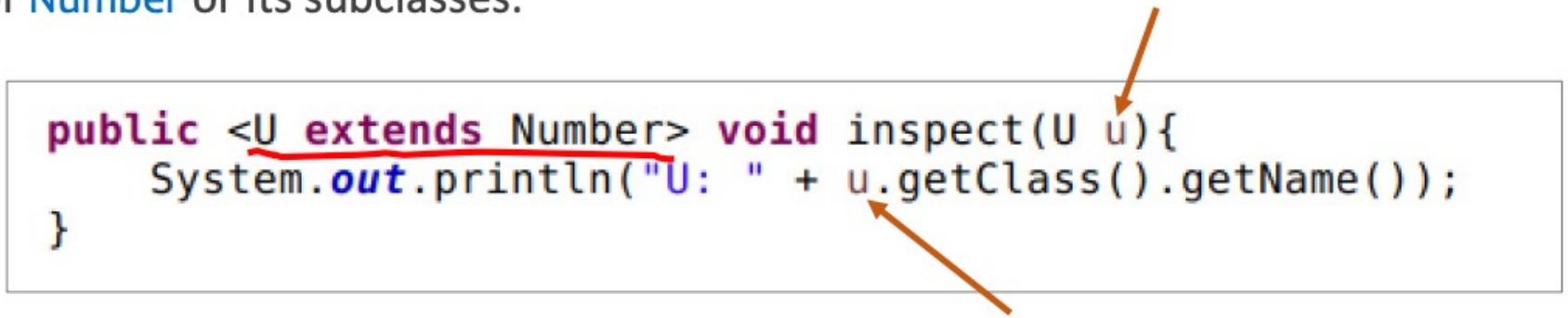
```
Pair<Integer, String> p1 = new Pair<>(1, "apple");
Pair<Integer, String> p2 = new Pair<>(2, "pear");
boolean  same = Util.<Integer, String>compare(p1, p2);
```

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);
```

Bounded Type Parameters

- 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.



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

```
public class NaturalNumber<T extends Integer> {
```

Multiple Bounds

- A type parameter can have multiple bounds:

  `< T extends B1 & B2 & B3 >`

- A type variable with multiple bounds is a subtype of all the types listed in the bound.
- Note that B1, B2, B3, etc. in the above refer to interfaces or a class. There can be at most one class (single inheritance), and the rest (or all) will be interfaces.
- If one of the bounds is a class, it must be specified first.

Generic Methods and Bounded Type Parameters

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem)  // compiler erro
            ++count;
    return count;
}
```

X - invalid

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

```
public static <T extends Comparable<T>> int countGreaterThan(T[] anArray, T elem)
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

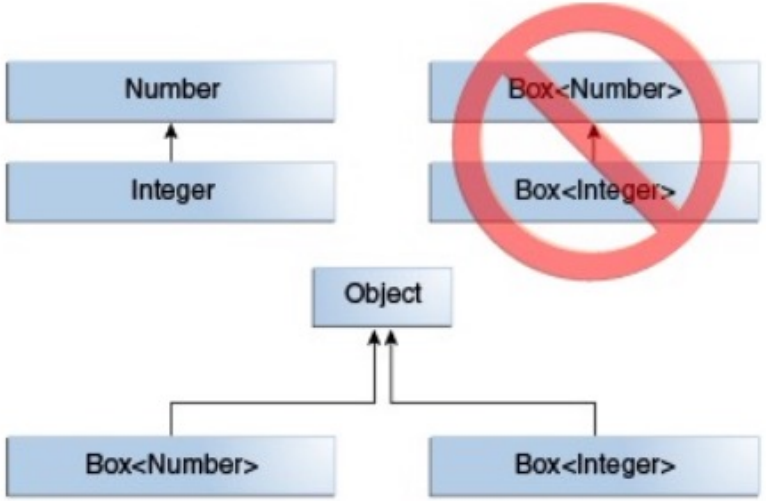
Valid

Generics, Inheritance, and Subtypes

- Consider the following method:

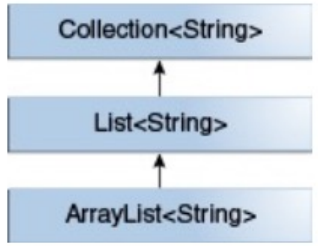
```
public void boxTest( Box<Number> n ) { /* ... */ }
```

- What type of argument does it accept?
- Are you allowed to pass in Box<Integer> or Box<Double> ?
- 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.



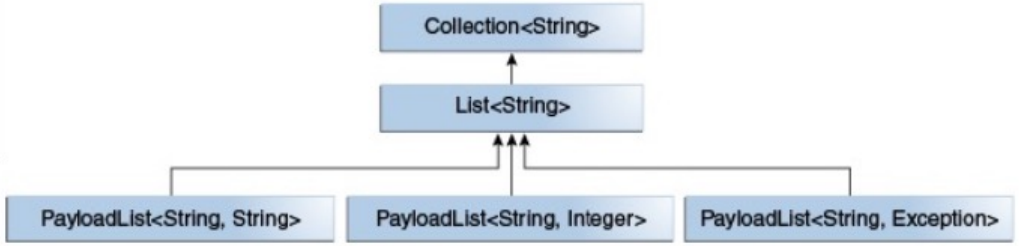
Generic Classes and Subtyping

- You can subtype a generic class or interface by extending or implementing it.
- The relationship between the type parameters of one class or interface and the type parameters of another are determined by the **extends** and **implements** clauses.
- ArrayList<E> implements List<E>, and List<E> extends Collection<E>.
- So ArrayList<String> is a subtype of List<String>, which is a subtype of Collection<String>.
- So long as you do not vary the type argument, the subtyping relationship is preserved between the types.




```
interface PayloadList<E,P> extends List<E> {
  void setPayload(int index, P val);
  ...
}
```

```
PayloadList<String,String>
PayloadList<String,Integer>
PayloadList<String,Exception>
```



Wildcards: Upper bounded

- In generic code, the question mark (**?**), called the **wildcard**, represents an unknown type.
- 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.
- The **upper bounded wildcard**, **< ? extends** Foo >, where Foo is any type, matches Foo and any subtype of Foo .
- You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

```
public static void process(List<? extends Foo> list) {
    for (Foo elem : list) {
        // ...
    }
}
```

```
public static double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Wildcards: Unbounded

- The **unbounded wildcard** type is specified using the wildcard character (?), for example, List< ? >. This is called a list of unknown type.

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

It prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on

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

To write a generic printList method, use **List<?>**

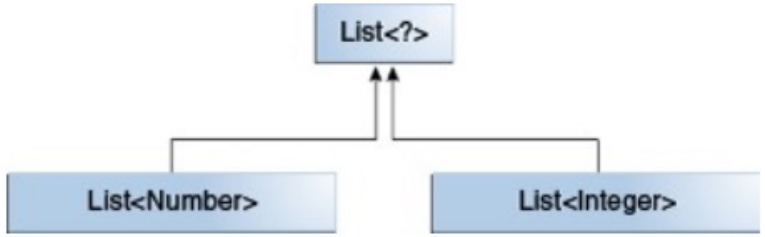
Wildcards: Lower Bounded

- An upper bounded wildcard restricts the unknown type to be a specific type or a subtype of that type and is represented using the **extends** keyword.
- A **lower bounded wildcard** is expressed using the wildcard character ('?'), following by the **super** keyword, followed by its lower bound: < ? **super** A >.
- To write the method that works on lists of Integer and the super types of Integer, such as Integer, Number, and Object, you would specify List<? **Super** Integer>.
- The term List<Integer> is more restrictive than List<? **super** Integer>.

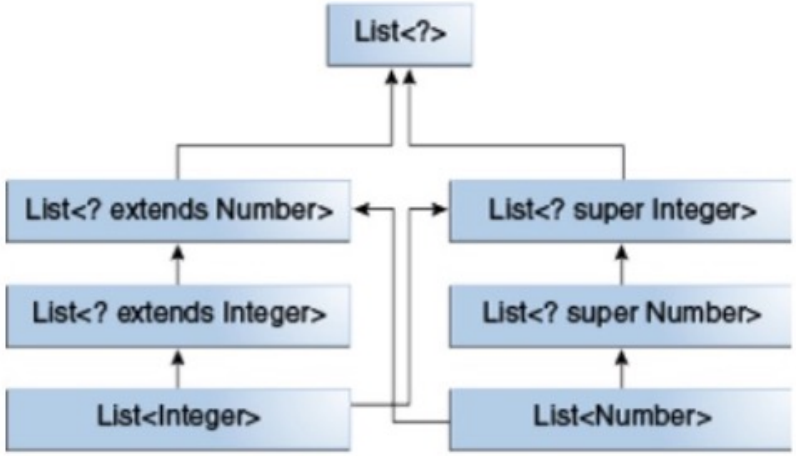
```
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Wildcards and Subtyping

- Although Integer is a subtype of Number, List<Integer> is **not** a subtype of List<Number> and, these two types are not related.



- The **common parent** of List<Number> and List<Integer> is **List<?>.**



A hierarchy of several generic `List` class declarations.

Collections in Java

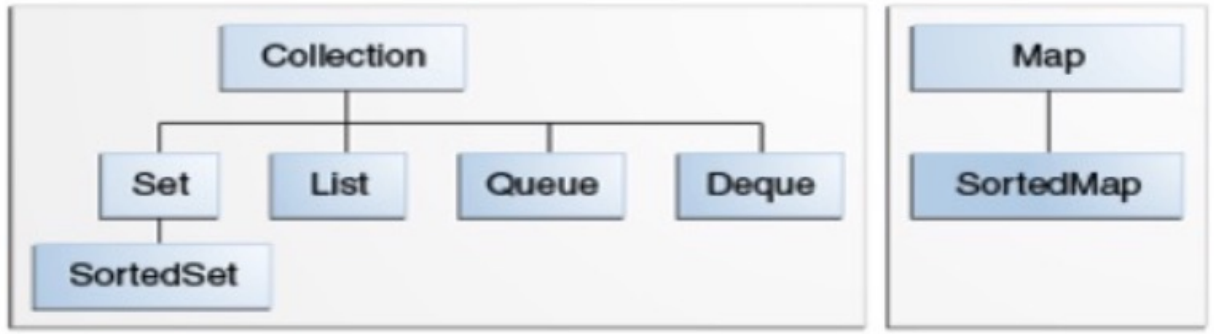
A collections framework is a unified architecture for representing and manipulating collections. A collection is simply an object that groups multiple elements into a single unit.

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