Universitatea Tehnica din Cluj-Napoca
Departament Calculatoare

Programming Techniques in Java

Techniques with Generic Type

T. Cioara, C. Pop, V. Chifu
2025

Generic Type

- A reference type which accepts one or more type parameters
  - Generic type declaration in classes, interfaces, methods
  - Concrete type should be specified when used
- Implemented at the compiler level
- At run time JVM has no knowledge of generic type

Advantages

- Improves reliability, readability, compile-time type safety (no cast is necessary)
- Allows writing polymorphic code that works with any type

- **<T> – formal generic type**
- **Convention**
  - **T – Type**
  - **E – Element**
  - **K – Key**
  - **V – Value**

Motivation

- Detection of errors at compile time

Comparable definition before JDK 1.5

```
public interface Comparable {
    public int compareTo(Object o);
}
```

Comparable definition after JDK 1.5

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

Using Comparable before JDK 1.5

```
// compiles ok
// generates a run time error
Comparable c = new Date();
int comp = c.compareTo(“Cluj”);
```

Using Comparable after JDK 1.5

```
 // compile error
 // the code is more reliable
Comparable<Date> c = new Date();
int comp = c.compareTo(“Cluj”);
```

Motivation

- Raw Type is unsafe
  - Raw type is classical definition of classes/interfaces)

```
public class Max {
  public static Comparable max(Comparable o1, Comparable o2) {
   if (o1.compareTo(o2) > 0) return o1;
   else return o2;
  }
}

// compiles ok (warning), runtime error
Max.max("alpha", 3); // 3 is autobox to Integer(3)
```

Generic Type is safe

```
public class MaxSafe {
 public static <E extends Comparable<E>> E max (E o1, E o2) {
  if (o1.compareTo(o2) > 0)  return o1;
  else return o2;
 }
}

MaxSafe.max("alpha", 3); // compilation error
```

Motivation

- Facilitates polymorphic definitions

```
public class ClassicalStack {
    private ArrayList list = new java.util.ArrayList();

    public Object push(Object o) { list.add(o); return o; }

    public Object pop() {
        Object o = list.get(getSize() – 1);
        list.remove(getSize() – 1);
        return o;
    }
}
```

```
ClassicalStack stk1 = new ClassicalStack();
stk1.push(“Cluj");
stk1.push(“Oradea");
stk1.push(“Timisoara");
String s1 = stk1.pop(); // error
String s2 = (String)stk1.pop(); // OK
```

```
ClassicalStack stk2 = new ClassicalStack();
stk2.push(2.5);
stk2.push(0.3);
stk2.push(18.2);
double d1 = stk2.pop(); // error
double d2 = (double)stk2.pop(); // OK
```

Motivation

- Facilitates polymorphic definitions

```
public class GenericStack<E> {
   private java.util.ArrayList<E> list = new java.util.ArrayList<E>();
   public E push(E o) { list.add(o); return(o); }
   public E pop() {
      E o = list.get(getSize() – 1);
      list.remove(getSize() -1);
      return o; }
}
```

```
GenericStack<String> stk1 = new GenericStack<String>();
stk1.push(“Cluj");
stk1.push(“Oradea");
String s = stk1.pop(); // no cast
```

```
GenericStack<Double> stk2 = new GenericStack<> ();
stk2.push(2.5); // auto boxing
stk2.push(18.2);
Double d = stk2.pop(); // no cast
```

- Concrete type substitutes a formal generic type
- Formal generic types cannot be directly instantiated
  - !!! Illegal constructs: new T() , new T[SIZE]

Declaring generic classes and interfaces

- Example of a generic class with two type parameters

```
public class Entry<K, V> {
      private K key;
      private V value;

      public Entry(K key, V value) {
        this.key = key;
        this.value = value;
      }
      public K getKey() { return key; }
      public V getValue() { return value; }
}
```

Instantiation

```
Entry<String, Integer> entry = new Entry<>(“Vasile", 20);
// or
Entry<String, Integer> entry = new Entry<String, Integer>(“Vasile", 20)
```

Declaring generic classes and interfaces

- A generic method of a regular class

```
public class GenericMethodDemo {
   …
   public static <E> void print(E[] list) {
      for (int i = 0; i < list.length; i++)
         System.out.print(list[i] + " ");
      }

   public static void main(String[] args ) {
       Integer[] aInts = {1, 2, 3, 4, 5};
       String[] aStrings = {“Cluj", “Oradea", “Turda”};

       GenericMethodDemo.<Integer>print(aInts);
       GenericMethodDemo.<String>print(aStrings);
   } }
```

- **A generic method can be defined in a regular class or in a generic class**
- **It is possible to be used on different reference type parameters**

Declaring generic classes and interfaces

- A generic method of a generic class

```
public class Wrapper<T> {
  private T ref;
  public Wrapper(T ref) { this.ref = ref;}
  public T get() { return ref; }
  public void set(T a) { this.ref = ref; } }
```

```
public class Test<T> {
  // … other class resources
  public <V> void m1(Wrapper<V> a, Wrapper<V> b, T c) {
    // … do something
  } }
```

```
Test<String> t = new Test<String>();
Wrapper<Integer> iw1 = new Wrapper<Integer>(new Integer(100));
Wrapper<Integer> iw2 = new Wrapper<Integer>(new Integer(200));

// Integer is the actual type for the type parameter V of method m1()
t.<Integer>m1(iw1, iw2, "hello");

// Let the compiler to figure out the actual type parameter for the m1() call
t.m1(iw1, iw2, "hello"); // OK
```

Declaring generic classes and interfaces

- Generic Methods and Constructors

```
// class Test constructor
/* <U extends T> means that type parameter U must be the same
   with T or subtype of T */
public class Test<T> {
    public <U extends T> Test(U k) {
         // … do something
    }
}
```

- The type parameter passed to the constructor is explicitly indicated

```
Test<Number> t1 = new <Double>Test<Number>(new Double(12.89));
```

- The compiler figures out the actual type parameter passed to the constructor (Integer) with the value you pass

```
Test<Number> t2 = new Test<Number>(new Integer(123));
```

Type inference when creating objects

```
List<String> list = new ArrayList<String>();

// Use of diamond operator (Java 7 and later)
// no compile error
List<String> list = new ArrayList<>();

// Using ArrayList as a raw type, not a generic
// type generates unchecked warning
List<String> list = new ArrayList();

List<String> list1 = Arrays.asList("A", "B");
List<Integer> list2 = Arrays.asList(9, 19, 1969);

// Ok - Inferred type is String
List<String> list3 = new ArrayList<>(list1);

// Compile-time error
List<String> list4 = new ArrayList<>(list2);
```

- Compiler uses complex rules for type inferring
- Sometimes compiler fails to infer correctly the parameter type in an object-creation expression
  - Specify the parameter type instead of using the diamond operator (<>)
  - Use diamond operator only when type inference is obvious

```
public static void process(List<String> list) {
   // … code goes here
}
process(new ArrayList<>());
```

- The inferred type is
  - Object in Java 7 (Error),
  - String in Java 8 and later (No error)

Generics and Arrays

- Arrays need to know its type when are created
  - Perform runtime check when an element is stored as to make sure that the element is assignment-compatible with the array type
  - Array's type information is not available at runtime if you use a generic parameter to create the array.

```
public class GenericArrayTest<T> {
   private T[] elements;
   public GenericArrayTest(int nmb) {
      elements = new T[nmb]; // compile time error
   }
   // … Other code goes here
}
```

- **new** is a run time operator
- No runtime information about T !!

```
// compile time error
Wrapper<String> gsArray = new Wrapper<String>[10];
```

```
// It is allowed to create an array of unbounded wild card:
Wrapper<?>[] arr = new Wrapper<?>[10];
```

Generics and Arrays

- How to create an array of generic type?
  - Use newInstance() method of the java.lang.reflect.Array
  - Generates the object of type Class corresponding to Wrapper. Class
  - An unchecked warning at compile time will be issued due to the cast used in the array creation statement (no type information at runtime)

```
Wrapper<String>[] a = (Wrapper<String>[])Array.newInstance(Wrapper.class, 10);

// Populate the array
a[0] = new Wrapper<String>("Hello");  // ok
Object[] objArray = (Object[]) a;     // ok

objArray[0] = new Object();         // Throws a java.lang.ArrayStoreException
```

Bounded generic types

- The type parameters need to fulfill certain requirements
  - A type bound is used

```
public static <T extends Comparable<T>> T max(Collection<T> coll) {
    T candidate = coll.iterator().next();
    for (T elt : coll) { if (candidate.compareTo(elt) < 0) candidate = elt; }
    return candidate;
}
```

- T is bounded by Comparable<T>
- iterator().next() is used rather than get(0) to get the first element, because get is not defined on collections other than lists.
- Method raises a NoSuchElement exception when the collection is empty

- **Bounds for type variables by the keyword extends**
- **An unbounded generic type T is like <T extends Object>**
- **Type variables must always be bounded using extends, never super**

Bounded generic types

- Calling method max
  - Integer (since Integer implements Comparable<Integer>) or
  - String (since String implements Comparable<String>):

```
List<Integer> ints = Arrays.asList(0,1,2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero","one","two");
assert Collections.max(strs).equals("zero");
```

T cannot be Number (Number does not implement Comparable)!

List<Number> nums = Arrays.asList(0,1,2,3.14);

assert Collections.max(nums) == 3,14; // compile-time error

Supertype-Subtype relationship

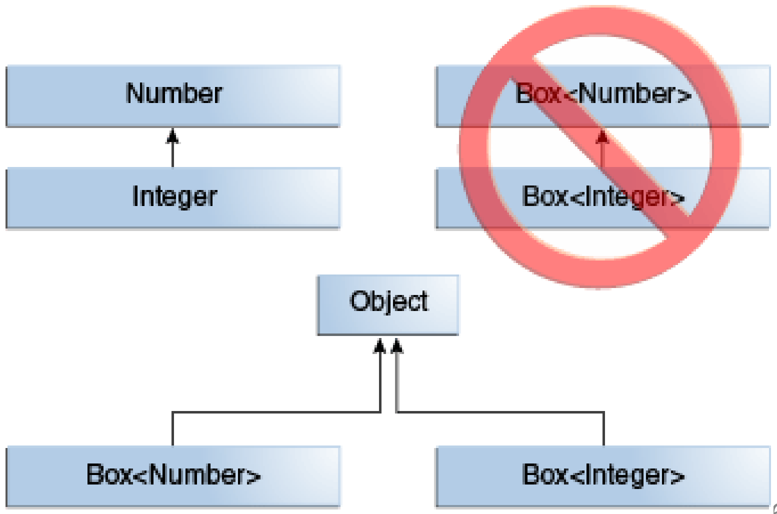
- Does not apply with parameterized types

```
Wrapper<String> stringWrapper = new Wrapper<String>("Hello");
stringWrapper.set("a string");

Wrapper<Object> objectWrapper = new Wrapper<Object>(new Object());
objectWrapper.set(new Object()); // set another object

objectWrapper.set("a string"); // ok
```

```
~~objectWrapper = stringWrapper;~~
```

If possible, the following would happen:

```
objectWrapper = stringWrapper;
objectWrapper.set(new Object());
String s = stringWrapper.get();
// throws a runtime ClassCastException
```



Wildcards

- Wildcard type - denoted by <?>
  - ? - Denotes unknown type
  - Used to deal with supertype-subtype relations in parametrized types
  - Specify that method parameter and return types are allowed to vary
- Wildcard processing - complex set of rules
  - Generics purpose is safety at compile time
  - If the compiler is satisfied that the operation will not produce any surprising results in execution, it allows the statement to pass

If you want to enforce some relationship on the different types of method arguments, you should not used wildcards

Wildcards

- Discussion on examples

```
public class NonGenericClass {
    public void printArray(Number[] numbers){
        for(Number number: numbers){
            … } } }
```

```
public class NonGenericClass {
    public void printList(List<Number> numbers){
        for(Number number: numbers){
            System.out.print(number + " ");
        }
    }
}
```

Polymorphic usage of the printArray method

```
NonGenericClass nonGenericClass = new NonGenericClass();
Number[] numbers = new Number[] {1, 2, 3, 4, 5, 6, 7, 8};
nonGenericClass.printArray(numbers);
System.out.println();
Integer[] integers = new Integer[] {1, 2, 3, 4, 5, 6, 7, 8};
nonGenericClass.printArray(integers);
```

There is no sub-type relationship between List<Number> and List<Integer> !
Solution: create separate methods for printing List<Integer>, List<Double>, etc.

```
List<Number> numberList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
nonGenericClass.printList(numberList);

List<Integer> integerList = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8);
nonGenericClass.printList(integerList);
```

Wildcards

- **Lower and Upper Bounded**

Wrong approach

```
public static double sum(Wrapper<?> n1, Wrapper<?> n2) { … }

double d = sum(new Wrapper<Integer>(new Integer(125)), new Wrapper<String>("Hello"));
// Meaningless computation: Compiles OK => Runtime error
```

Correct approach

```
public static double sum(Wrapper<? extends Number> n1, Wrapper<? extends Number> n2) {
     Number num1 = n1.get(); Number num2 = n2.get();
     double sum = num1.doubleValue() + num2.doubleValue();
    return sum;
}
// compile time error
double d = sum(new Wrapper<Integer>(new Integer(125), new Wrapper<String>("Hello"));

Wrapper<Integer> intWrapper = new Wrapper<Integer>(new Integer(10));
Wrapper<? extends Number> numberWrapper = intWrapper; // Ok
numberWrapper.set(new Integer(1220)); // compile  error
```

Wildcards

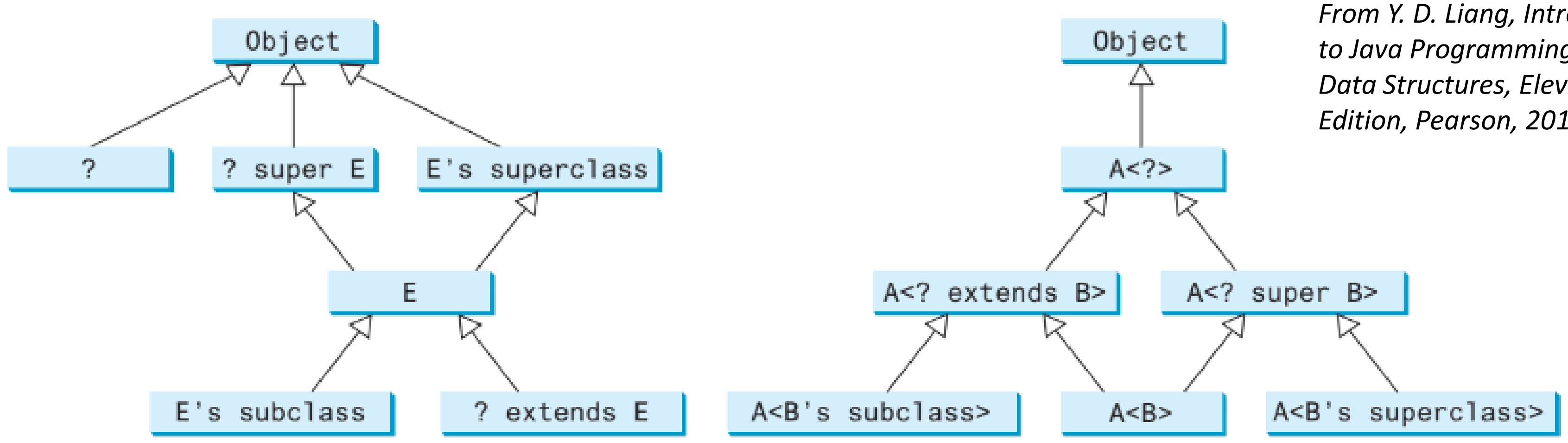
- **Lower and Upper bounded**
  - Copy method of class Collections
  - Copies into a destination list, all the elements from a source list

```
public static <T> void copy(List<? super T> dst, List<? extends T> src) {
      int sz = src.size();
      for (int i = 0; i < sz; i++) {dst.set(i, src.get(i)); }
}

List<Object> objs = Arrays.<Object>asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);

Collections.copy(objs, ints);
assert objs.toString().equals("[2, 3.14, four, 5, 6]");
```

  - dst is defined as lower bounded by T: ? super T
    - the destination has elements of any type that is a supertype of T
  - src is defined as upper bounded by T: ? extends T
    - the source list may have elements of any type that is a subtype of T

Wildcards

- The relationship between generic types and wildcard
  - T is a generic type, A and B are classes or interfaces



From Y. D. Liang, Introduction to Java Programming and Data Structures, Eleventh Edition, Pearson, 2019

? extends E

- **contains every type in an interval bounded by the type of null below and by E above (where the type of null is a subtype of every reference type).**

? super E

- **contains every type in an interval bounded by T below and by Object above**

Producer Extends Consumer Super

- Defines good practice to using wildcards

	Bounded Parameter Produces Ts?		
		Yes	**No**
Bounded Parameter Consumes Ts?	**Yes**	MyClass<T> (*Invariant in T*)	MyClass<? super T> (*Contrainvariant in T*)
	No	MyClass<? extends T> (*Covariant in T*)	MyClass<?> (*Independent of T*)

- Shown in the signature of the copy method copy

```
public static <T> void copy (List<? super T> dest, List<? extends T> src )
```

  - gets values out of the source src => it is declared with an extends wildcard
  - puts values into the destination dst => it is declared with a super wildcard

Producer Extends Consumer Super

Whenever you use an iterator or a loop and you get values out of a data structure, an extends wildcard should be used

```
public static double sum(Collection<? extends Number> nums) {
  double s = 0.0d;
  for (Number num : nums) s += num.doubleValue();
  return s;
}
```

```
// Since extends is used, the following calls are legal:

List<Integer> ints = Arrays.asList(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = Arrays.asList(2.78,3.14);
assert sum(doubles) == 5.92;

List<Number> nums = Arrays.<Number>asList(1,2,2.78,3.14);
assert sum(nums) == 8.92;

// The first two calls would not be legal if extends were not used
```

Producer Extends Consumer Super

Whenever you use the add method, you put values into a data structure, a super wildcard should be used

```
public static void addToCol(Collection<? super Integer> ints, int n) {
  for (int i = 0; i < n; i++) ints.add(i);
}
```

```
// Since this uses super, all of the following calls are legal:

List<Integer> ints = new ArrayList<Integer>();
addToCol(ints, 5);
assert ints.toString().equals("[0, 1, 2, 3, 4]");

List<Number> nums = new ArrayList<Number>();
addToCol(nums, 5);  nums.add(5.0);
assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");

List<Object> objs = new ArrayList<Object>();
addToCol(objs, 5);  objs.add("five");
assert objs.toString().equals("[0, 1, 2, 3, 4, five]");

// The last two calls would not be legal if super were not used
```

Producer Extends Consumer Super

Whenever you both put values into and get values out of the same data structure, you should not use a wildcard

```
public static double sumCount(Collection<Number> nums, int n) {
    addToCol(nums, n);
    return sum(nums);
}
```

```
List<Number> nums = new ArrayList<Number>();
double sum = sumCount(nums,5);
assert sum == 10;

// Since there is no wildcard, the argument must be a collection of Number
```

The collection nums is passed to both sum and addToCol must both:

- **extend Number (as sum requires) and**
- **be super to Integer (as addToCol requires)**

The only two classes that satisfy both constraints are Number and Integer, and we have picked the first of these.

Producer Extends Consumer Super

The principles also works in the opposite way:

- *If extends is present = > all you will be able to do is get but not put values of that type*
- *If super is present => all you will be able to do is put but not get values of that type*

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
double dbl = sum(nums);  // ok

nums.add(3.14);  // compile-time error
```

- The call to sum is OK as it gets values from the list
- The call to add is not OK as it puts a value into the list
  - One could add a double to a list of integers!

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
ints.add(3);  // ok

double dbl = sum(ints); //compile-time error
```

- The call to add is OK as it puts a value into the list
- The call to sum is not OK
- The sum of a list containing a string makes no sense!

Erasure and restrictions

- Type erasure
  - The compiler uses the generic type information to compile the code but erases it afterwards
  - Generic information is not available at runtime
  - Enables the generic code to be backwards compatible with legacy code that uses raw types
  - Once the compiler confirms that the generic type is safely (correctly) used it is converted to raw type

```
ArrayList<String> list = new ArrayList<String>();
list.add("UTCN");
String univ = list.get(0);
```

Translated into raw types

```
ArrayList list = new ArrayList();
list.add("UTCN");
String univ = (String) (list.get(0));
```

Erasure and restrictions

- Type erasure examples

```
public static <E> void print(E[] list) {
   for (int i = 0; i < list.length; i++)
        System.out.print(;ist[i] + " ");
}
```

Translated into raw types

```
public static Object void print(Object[] list) {
   for (int i = 0; i < list.length; i++)
        System.out.print(;ist[i] + " ");
}
```

```
public <E extends GeometricObject> boolean equalArea (E o1, E o2) {
   return o1.getArea() == o2.getArea();
}
```

Translated into raw types

```
public GeometricObject boolean equalArea (GeometricObject o1, GeometricObject o2) {
   return o1.getArea() == o2.getArea();
}
```

Erasure and restrictions

- Restrictions on how generic types are used

Restriction 1 - generic type instances are not allowed

```
// incorrect because new E is executed at runtime
// but E is not available at runtime
E object = new E();
```

Restriction 2 - no primitive type arguments

```
~~ArrayList<int>~~
```

Restriction 3 - Exception classes cannot be generic

- The JVM must check the exception thrown from the try clause to see if it matches the type specified in the catch clause
- This is impossible because the type information is not present in runtime

```
~~public class MyException<T> extends Exception { ... }~~
If this would be allowed, you can do
try { ... } catch (My Exception<T> ex) { ... }
```

Erasure and restrictions

Restriction 4 - generic array creation is not allowed

```
E[] arr = new E[dim];
This can be rewritten as
E[] arr = (E[]) new Object[dim];
Casting to E[] causes an unchecked compile warning
The compiler could not be sure that the casting will succeed at runtime
```

Restriction 5 - No generic anonymous classes

- An anonymous class is a one-time class that does not have a name.
- You need a class name to specify the actual type parameter, therefore, you cannot have a generic anonymous class.
- However, you can have generic methods inside an anonymous class
- An anonymous class can inherit a generic class, and an anonymous class can implement generic interfaces.