Java Generics

1 Introduction

JDK 5 introduces *generics*, which supports *abstraction over types* (or parameterized types) on classes and methods. The class or method designers can be *generic about types in the definition*, while the users are to provide the *specific types (actual type) during the object instantiation or method invocation.*

You are certainly familiar with passing arguments into methods. You place the arguments inside the round bracket () and pass them into the method. In generics, instead of passing arguments, we pass *type information* inside the angle brackets <>.

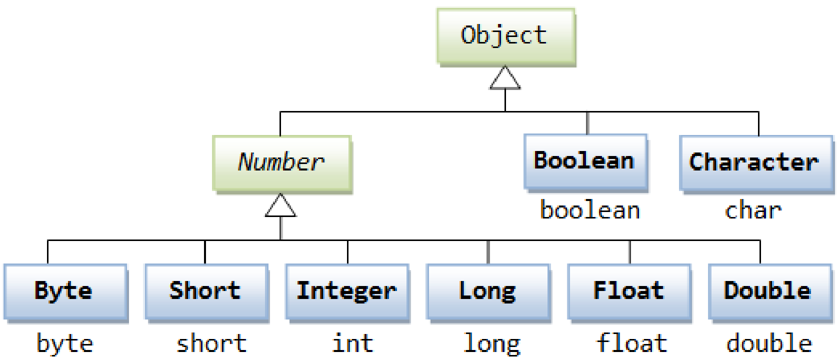
The primary usage of generics is to *abstract over types* for the Collection Framework.

Before discussing generics, we need to introduce these related new Java language features introduced in JDK 5:

1. Auto-Boxing and Auto-Unboxing between primitives and their wrapper objects.
2. Enhanced for-each loop.

1.1 Auto-Boxing/Unboxing between Primitives and their Wrapper Objects (JDK 5)

A Java Collection (such as List and Set) contains only objects. It cannot holds primitives (such as int and double). On the other hand, arrays can hold primitives and objects, but they are not resizable. To put a primitive into a Collection (such as ArrayList), you have to wrap the primitive into an object using the corresponding primitive wrapper class as shown below:



Prior to JDK 5, you need to **explicitly** wrap a primitive value into an object and unwrap the primitive value from the wrapper object, for example,

```java
// Pre-JDK 5
Integer intObj = new Integer(5566);      // wrap an int to Integer by
                                         // constructing an instance of Integer
int i = intObj.intValue();               // unwrap Integer to int

Double doubleObj = new Double(55.66);    // wrap double to Double
double d = doubleObj.doubleValue();      // unwrap Double to double
```

The pre-JDK 5 approach involves quite a bit of codes to do the wrapping and unwrapping. JDK 5 introduces a new feature called auto-boxing and auto-unboxing to resolve this problem, by delegating the compiler to do the job. For example,

```java
// JDK 5
Integer intObj = 5566;      // auto-box from int to Integer by the compiler
int i = intObj;             // auto-unbox from Integer to int by the compiler

Double doubleObj = 55.66;   // auto-box from double to Double
double d = doubleObj;       // auto-unbox from Double to double
```

Primitive Wrapper Objects, Like Strings, are Immutable!

For example,

```java
public class PrimitiveWrapperImmutableTest {
  public static void main(String[] args) {
    Integer iObj = 123;   // auto-box
    // Print reference
    System.out.println(Integer.toHexString(System.identityHashCode(iObj)));
    // 36baf30c

    iObj += 1;   // a new Integer object is created and assigned to iObj
    System.out.println(iObj);   // 124
    System.out.println(Integer.toHexString(System.identityHashCode(iObj)));
    // 7a81197d  (different reference!)

    // This is similar to the immutable String
    String str = "hello";
    System.out.println(Integer.toHexString(System.identityHashCode(str)));
    // 5ca881b5
    str += "world";
    System.out.println(str);   // helloworld
    System.out.println(Integer.toHexString(System.identityHashCode(str)));
```

```
        // 7adf9f5f
      }
    }
```

1.2 Enhanced for-each Loop (JDK 5)

JDK 5 also introduces a new for-each loop, which you can use to traverse through all the elements of an array or a Collection.

The syntax is as follows. You should read as for each element in the collection/array.

```
for ( type item : array_collection ) {
  body ;
}
```

For example,

```
import java.util.List;
import java.util.ArrayList;

public class J5ForEachLoopTest {
  public static void main(String[] args) {
    // Use for-each loop on Array
    int[] numArray = {11, 22, 33};
    for (int num : numArray) {
      System.out.println(num);
    }
    // 11
    // 22
    // 33

    // Same as:
    for (int idx = 0; idx < numArray.length; ++idx) {
      System.out.println(numArray[idx]);
    }

    // Use for-each loop on Collection
    List<String> coffeeLst = new ArrayList<>();
    coffeeLst.add("espresso");
    coffeeLst.add("latte");
    for (String coffee : coffeeLst) {
      System.out.println(coffee.toUpperCase());
    }
```

```
            // ESPRESSO
            // LATTE
        }
    }
```

Can you modify the Array/Collection via Enhanced for-each Loop?
For primitive arrays, the for-each loop's local variable clones a value for each item and, hence, you cannot modify the original array. (A Collection cannot hold primitives.) For example,

```java
import java.util.Arrays;

public class ForEachLoopPrimitiveTest {
  public static void main(String[] args) {
    // Using for-each loop on an array of primitive (e.g., int[])
    int[] iArray = {11, 22, 33};
    for (int item : iArray) {
      System.out.print(item + " ");
      item += 99;  // try changing
    }
    // 11 22 33
    System.out.println(Arrays.toString(iArray));
    // [11, 22, 33]  (no change)

    // You need to use the traditional for-loop to modify the array
    for (int i = 0; i < iArray.length; ++i) {
      iArray[i] += 99;
    }
    System.out.println(Arrays.toString(iArray));
    // [110, 121, 132]  (changed!)
  }
}
```

For object arrays or Collections, an object reference is passed to the loop's local variable, you can modify the object via this reference. For example,

```java
import java.util.Arrays;

class MyMutableInteger {
  private int value;  // private variable, mutable via setter

  public MyMutableInteger(int value) {  // constructor
    this.value = value;
```

```java
    }

    public String toString() {
      return "MyMutableInteger[value=" + value + "]";
    }

    public void setValue(int value) {  // setter
      this.value = value;
    }
  }

  public class ForEachLoopMutableObjectTest {
    public static void main(String[] args) {
      // Using for-each loop on an array of primitive (e.g., int[])
      MyMutableInteger[] iArray = {new MyMutableInteger(11),
        new MyMutableInteger(22)};
      for (MyMutableInteger item : iArray) {
        System.out.println(item);
        item.setValue(99);  // try changing via setter
      }

      // MyMutableInteger[value=11]
      // MyMutableInteger[value=22]
      System.out.println(Arrays.toString(iArray));
      // [MyMutableInteger[value=99], MyMutableInteger[value=99]] (changed!)
    }
  }
```

However, for immutable object arrays and Collections (such as String and Integer), you cannot modify the contents, as new objects were created and assigned to the reference. For example,

```java
import java.util.Arrays;
public class ForEachLoopImmutableObjectTest {
  public static void main(String[] args) {
    // Using for-each loop on an array of immutable objects (such as String[])
    String[] sArray = {"dog", "cat", "turtle"};
    for (String item : sArray) {
      System.out.print(item + " ");
      item += "hello";  // a new String is created as Strings are immutable
    }

    // dog cat turtle
    System.out.println(Arrays.toString(sArray));
    // [dog, cat, turtle]  (no change)
  }
}
```

1.3 A Brief Summary of Inheritance, Polymorphism and Type Casting

The following rules applied to inheritance substitution and polymorphism:

1. A reference c of class C accepts instances of C. It also accepts instances of C's subtypes (says CSub), which is known as substitution. This is because CSub inherits all attributes and behaviors of C, and hence, can act as C.
2. Once substituted, you can only invoke methods defined in C, not CSub, since c is a reference of C.
3. If CSub overrides a method m of the supertype C, then c.m() runs the overridden version in the subtype CSub, not the C's version.

The following rules applied to type casting:

1. Casting from subtype up to supertype (up casting) is type-safe, and does not require an explicit type casting operator.
2. Casting from supertype down to subtype (down casting) is NOT type-safe, and requires an explicit type casting operator.

For example,

```java
class C1 {
  public void sayHello() {
    System.out.println("C1 runs sayHello()");
  }

  public void methodC1() {
    System.out.println("C1 runs methodC1()");
  }
}
```

```java
class C2 extends C1 {  // C2 is a subclass of C1
  @Override
  public void sayHello() {
    System.out.println("C2 runs overridden sayHello()");
  }

  public void methodC2() {
    System.out.println("C2 runs methodC2()");
  }
}
```

```java
public class PolymorphismTest {
  public static void main(String[] args) {
    // Substitution: Reference to C1 can accept instance of C1 and its subclasses
    C1 c1Ref = new C2();  // substituted with C1 subclass' instance
    c1Ref.methodC1();      // C1 runs methodC1()
    // c1Ref.methodC2();     // CANNOT reference subclass method
    // error: cannot find symbol

    // Polymorphism: run the overridden version
    c1Ref.sayHello();     // C2 runs overridden sayHello()

    // Upcasting is type-safe, does not require explicit type cast operator
    C1 c1Ref2 = new C2();
    // Downcasting is NOT type-safe, require explicit type cast operator
    C2 c2Ref = (C2)c1Ref2;
    // C2 c2Ref = c1Ref2;
    // error: incompatible types: C1 cannot be converted to C2
  }
}
```

2 Introduction to Generics by Examples (JDK 5)

This section gives some examples on working with generics, meant for experienced programmers to get a quick review. For novices, start with the next section.

2.1 Example 1: Using Generic Collection: List<E> and ArrayList<E>

The class java.util.ArrayList<E> is designed (by the class designer) to take a generics type <E> as follows:

```java
public class ArrayList<E> implements List<E> ... {
  public boolean add(E e)
  public void add(int index, E element)
  public boolean addAll(Collection<? extends E> c)
  public boolean addAll(int index, Collection<? extends E> c)
  public E get(int index)
  public E remove(int index)
  .......
}
```

To construct an instance of an ArrayList<E>, we need to provide the actual type for E. The actual type provided will then substitute all references to E inside the class. For example,

```java
import java.util.List;
import java.util.ArrayList;

public class GenericArrayListTest {
  public static void main(String[] args) {
    // Set "E" to "String"
    ArrayList<String> fruitLst = new ArrayList<String>();
    fruitLst.add("apple");
    fruitLst.add("orange");
    System.out.println(fruitLst);  // [apple, orange]
    // JDK 5 also introduces the for-each loop
    for (String str: fruitLst) {   // we need to know type of elements
      System.out.println(str);
    }
    // apple
    // orange

    // Adding non-String type triggers compilation error
    // fruitLst.add(99);
    // compilation error: incompatible types: int cannot be converted to String

```

```java
        // JDK 7 introduces diamond operator <> for type inference to shorten the code
        ArrayList<String> coffeeLst = new ArrayList<>();  // can omit type in
                                                          // instantiation
        coffeeLst.add("espresso");
        coffeeLst.add("latte");
        System.out.println(coffeeLst);  // [espresso, latte]

        // We commonly program at the specification in List instead of implementation ArrayList
        List<String> animalLst = new ArrayList<>();  // Upcast ArrayList<String>
                                                     // to List<String>
        animalLst.add("tiger");
        System.out.println(animalLst);  // [tiger]

        // A Collection holds only objects, not primitives
        // Try auto-box/unbox between primitives and wrapper objects
        List<Integer> intLst = new ArrayList<>();
        intLst.add(11);           // primitive "int" auto-box to "Integer" (JDK 5)
        int i1 = intLst.get(0);  // "Integer" auto-unbox to primitive "int"
        System.out.println(intLst);  // [11]
        // intLst.add(2.2);
        // compilation error: incompatible types: double cannot be converted to Integer

        // "Number" is a supertype of "Integer" and "Double"
        List<Number> numLst = new ArrayList<>();
        numLst.add(33);    // primitive "int" auto-box to "Integer", upcast to Number
        numLst.add(4.4);   // primitive "double" auto-box to "Double", upcast to Number
        System.out.println(numLst);  // [33, 4.4]
    }
}
```

The above example showed that the class designers could be generic about type; while the users provide the specific actual type during instantiation. With generics, we can design one common class that is applicable to all types with compile-time type-safe checking. The actual types are passed inside the angle bracket <>, just like method arguments are passed inside the round bracket ().

2.2 Example 2: Pre-Generic Collections (Pre-JDK 5) are not Compile-Time Type-Safe

If you are familiar with the pre-JDK 5's collections such as ArrayList, they are designed to hold java.lang.Object. Since Object is the common root class of all the Java's classes, a collection designed to hold Object can hold any Java objects. There is, however, one big problem. Suppose, for example, you wish to define an ArrayList of String. In the add(Object) operation, the String will be upcasted implicitly into Object by the compiler. During retrieval, however, it is the programmer's responsibility to downcast the Object back to a String explicitly. If you inadvertently added in a non-String object, the compiler cannot detect the error, but the downcasting will fail at runtime. Below is an example:

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

// Pre-JDK 5 Collection
public class PreJ5ArrayListTest {
  public static void main(String[] args) {
    // We create a List meant for String
    List strLst = new ArrayList();  // Pre-JDK 5 List holds Objects
    strLst.add("alpha");     // String upcasts to Object implicitly
    strLst.add("beta");
    Iterator iter = strLst.iterator();
    while (iter.hasNext()) {
      // need to explicitly downcast Object back to String
      String str = (String)iter.next();
      System.out.println(str);
    }

    // We inadvertently add a non-String into the List meant for String
    strLst.add(new Integer(1234));  // Compiler and runtime cannot detect
                                    // this logical error
    String str = (String)strLst.get(2);  // Retrieve and downcast back to String
    // Compile ok, but runtime exception
    // java.lang.ClassCastException: class java.lang.Integer cannot be cast to class
        java.lang.String
  }
}
```

We could use an instanceof operator to check for proper type before downcasting. But again, instanceof detects the problem at runtime. How about *compile-time type-checking*? JDK 5 introduces generics to resolve this problem to provide compile-time type-safe checking, as shown in the above example.

2.3 Generic Wildcard (?) and Bounded Type Parameters

Wildcard (?) can be used to represent an unknown type in Generics:

- <? extends T>: called upper bounded wildcard which accepts type T or T's subtypes. The upper bound type is T.
- <? super T>: called lower bounded wildcard which accepts type T or T's supertypes. The lower bound type is T.
- <?>: called unbounded wildcard which accepts all types.

Bounded Type Parameters have the forms:

- <T extends ClassName>: called upper bounded type parameter which accepts the specified ClassName and its subtypes. The upper bound type is ClassName.

2.4 Example 3: Upper-Bounded Wildcard <? extends T> for Accepting Collections of T and T's Subtypes

As an example, the ArrayList<E> has a method addAll() with the following signature:

```java
public class ArrayList<E> implements List<E> .... {
  public boolean addAll(Collection<? extends E> c)
    .......
}
```

The addAll() accepts a Collection of E and E's subtypes. Via substitution, it also accepts subtypes of Collection.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedList;
import java.util.Set;
import java.util.HashSet;

public class GenericUpperBoundedTest {
  public static void main(String[] args) {
    // Set E to Number.
    // Number is supertype of Integer, Double and Float
    List<Number> numLst = new ArrayList<>();
    numLst.add(1.1f);    // primitive float auto-box to Float, upcast to Number
    System.out.println(numLst);  // [1.1]

    // Integer is a subtype of Number, which satisfies <? extends E=Number>
    Collection<Integer> intColl = new LinkedList<>();
    intColl.add(2);  // primitive int auto-box to Integer
    intColl.add(3);
    System.out.println(intColl);  // [2, 3]
    // Try .addAll(Collection<? extends E>)
    numLst.addAll(intColl);
    System.out.println(numLst);   // [1.1, 2, 3]

    // Double is a subtype of Number, which satisfies <? extends E=Number>
    // Set is a subtype of Collection. Set<Double> is a subtype of Collection<Double>
    Set<Double> numSet = new HashSet<>();
    numSet.add(4.4);   // primitive double auto-box to Double
    numSet.add(5.5);
    System.out.println(numSet);  // [5.5, 4.4]
    // Try .addAll(Collection<? extends E>)
    numLst.addAll(numSet);
    System.out.println(numLst);  // [1.1, 2, 3, 5.5, 4.4]
```

```
    }
}
```

Notes

- The addAll() is not merge, but iterating through the Collection and add elements one-by-one.
- If addAll() is defined as addAll(Collection<E>) without the upper bound wildcard, and E is Number, then it can accept Collection<Number>, but NOT Collection<Integer>.
- In generics, Collection<Integer> is not a subtype of Collection<Number>, although Integer is a subtype of Number. You cannot substitute Collection<Integer> for Collection<Number>. But Collection<Number> can contain Integers. See next section for the explanation.
- In generics, Set<String> is a subtype of List<String>, as Set is a subtype of List and they have the same parametric type.
- The upper bounded wildcard <? extends E> is meant to handle "Collection of E and E's subtypes", for maximum flexibility.

2.5 Example 4: Lower-Bounded Wildcard <? super T> for Applying Operations on T and T's Supertype

As an example, the List has a method forEach(Consumer<? super E> action) (introduced in JDK 8 inherited from its supertype Iterable), which accepts a Consumer capable of operating on type E and E's supertypes, to operate on each of the elements.

```java
public class List<E> implements Iterable<E> .... {
  public void forEach(Consumer<? super E> action)
    .......
}
```

```java
import java.util.List;
import java.util.function.Consumer;   // JDK 8

public class GenericLowerBoundedTypeTest {
  public static void main(String[] args) {
    // Set E to Double to create a List<Double>
    List<Double> dLst = List.of(1.1, 2.2);   // JDK 9 to generate an
                                            // unmodifiable List
```

```
        // Set up a Consumer<Double> that is capable of operating on Double
        // We can only use methods supported by Double, such as Double.doubleToRawLongBits(d)
        Consumer<Double> dConsumer = d -> System.out.printf("%x%n", Double.doubleToRawLongBits(d));
        // Run .forEach() with Consumer<Double> operating on each Double element
        dLst.forEach(dConsumer);
        // 3ff199999999999a
        // 400199999999999a

        // Set up a Consumer<Number>
        // Number is a supertype of Double, which satisfies <? super E=Double>.
        // We can only use methods supported by Number, such as .intValue()
        Consumer<Number> numConsumer = num -> System.out.println(num.intValue());  // JDK 8
        // Run .forEach() with Consumer<Number> operating on each Double element
        // Since Double is a subtype of Number. It inherits and supports all methods in Number.
        dLst.forEach(numConsumer);
        // 1
        // 2
    }
}
```

Notes

- If forEach() is defined as forEach(Consumer<E>) without the lower bound wildcard, and E is Double, then it can only accepts Consumer<Double>, but NOT Consumer<Number>. Since Number is a supertype of Double, Consumer<Number> can also be used to process Double. Hence, it make sense to use Consumer<? extends Double> to include the supertypes Consumer<Number> and Consumer<Object> for maximum flexibility.

- The lower bounded wildcard <? super E> is meant to operate on E, with function objects operating on E and E's supertype, for maximum flexibility.

2.6 Example 5: Generic Method with Upperbound and Lowerbound Wildcards

As an example, the java.lang.String class (a non-generic class) contains a generic method called transform() (JDK 12) with the following signature:

```
public class String {
  public <R> R transform(Function<? super String, ? extends R> f) {
    return f.apply(this);
```

```
    }
    ......
}
```

This method takes a Function object as argument and returns a generic type R. The generic types used in generic methods (which is not declared in the class statement) are to be declared before the return type, in this case, <R>, to prevent compilation error "cannot find symbol".

The generic Function object takes two type arguments: a String or its supertypes <? super String>, and a return-type R or its subtypes <? extends R>.

For example,

```java
import java.util.function.Function;
import java.util.List;
import java.util.ArrayList;

public class StringTransformTest {
  public static void main(String[] args) {
    String str = "hello";

    // Set the return-type R to Number
    // Set up Function<String, Number>, which takes a String and returns a Number
    Function<String, Number> f1 = String::length;   // int auto-box to Integer,
        upcast to Number
    // Run the .transform() on Function<String, Number>
    Number n1 = str.transform(f1);
    System.out.println(n1);                  // 5
    System.out.println(n1.getClass());   // class java.lang.Integer
    // Integer i1 = str.transform(f1);
    // compilation error: incompatible types: inference variable R has incompatible bounds
    Integer i1 = (Integer)str.transform(f1);   // Explicit downcast
    System.out.println(i1);                          // 5

    // Double is a subtype of Number, satisfying <? extends R = Number>
    // Set up Function<String, Double>, which takes a String and returns a Double
    Function<String, Double> f2 = s -> (double)s.length();   // double ->
        Double
    Number n2 = str.transform(f2);        // Double upcast to Number
    System.out.println(n2);                  // 5.0
    System.out.println(n2.getClass());   // class java.lang.Double
    Double d2 = str.transform(f2);        // Okay

    // CharSequence is a supertype of String, which satisfies <? super String>
    // Integer is a subtype of Number, satisfying <? extends R = Number>
```

```java
        // Set up Function<CharSequence, Integer>, which takes a CharSequence and returns a
            Integer
        Function<CharSequence, Integer> f3 = CharSequence::length;  // int
                                                                    // auto-box to Integer
        Number n3 = str.transform(f3);  // Upcast Integer to Number
        System.out.println(n3);         // 5
    }
}
```

Notes

- Suppose that R is Number, Function<? super String, ? extends R> includes Function<String, Number>, Function<String, Integer>, Function<CharSequence, Number>, Function<CharSequence, Integer>, and etc.
- The upper bounded wildcard <? super String> allows function objects operating on String and its supertypes to be used in processing String, for maximum flexibility. See Example 4.
- The return type of R and the lower bounded wildcard <? extends R> permits function object producing R and R's subtype to be used, for maximum flexibility. See Example 3.

3 Generics Explained

We shall illustrate the use of generics by writing our own type-safe resizable array (similar to an ArrayList).

We shall begin with a non-type-safe non-generic version, explain generics, and write the type-safe generic version.

3.1 Example 1: Non-Type-Safe Non-Generic MyArrayList

Let us begin with a version without generics called MyArrayList, which is a linear data structure, similar to array, but resizable. For the MyArrayList to hold all types of objects, we use an Object[] to store the elements. Since Object is the single root class in Java, all Java objects can be upcasted to Object and store in the Object[].

MyArrayList.java

```java
import java.util.Arrays;

// A resizable array without generics, which can hold any Java objects
public class MyArrayList {
  private int size;              // number of elements
  private Object[] elements;     // can store all Java objects

  public MyArrayList() {         // constructor
    elements = new Object[10];   // allocate initial capacity of 10
    size = 0;
  }

  // Add an element, any Java objects can be upcasted to Object implicitly
  public void add(Object o) {
    if (size >= elements.length) {
      // allocate a larger array and copy over
      Object[] newElements = new Object[size + 10];
      for (int i = 0; i < size; ++i) {
        newElements[i] = elements[i];
      }
      elements = newElements;
    }
    elements[size] = o;
    ++size;
  }

  // Retrieves the element at Index. Returns an Object to be downcasted back to its original
     type
  public Object get(int index) {
    if (index >= size) {
      throw new IndexOutOfBoundsException("Index: " + index
        + ", Size: " + size);
```

```java
      }
      return elements[index];
    }

    // Returns the current size (length)
    public int size() {
      return size;
    }

    // toString() to describe itself
    @Override
    public String toString() {
      return Arrays.toString(Arrays.copyOfRange(elements, 0, size));
    }
}
```

MyArrayListTest.java

```java
public class MyArrayListTest {
  public static void main(String[] args) {
    // Create a MyArrayList to hold a list of Strings
    MyArrayList strLst = new MyArrayList();
    // Adding elements of type String
    strLst.add("alpha");   // String upcasts to Object implicitly
    strLst.add("beta");
    System.out.println(strLst);  // toString()
    // [alpha, beta]

    // Retrieving elements: need to explicitly downcast back to String
    for (int i = 0; i < strLst.size(); ++i) {
      String str = (String)strLst.get(i);
      System.out.println(str);
    }
    // alpha
    // beta

    // Inadvertently added a non-String object. Compiler cannot detect this logical error.
    // But trigger a runtime ClassCastException during downcast.
    strLst.add(1234);  // int auto-box to Integer, upcast to Object.
                       // Compiler/runtime cannot detect this logical error
    String str = (String)strLst.get(2);
    // compile ok
    // runtime ClassCastException: class java.lang.Integer cannot be cast to class
       java.lang.String
  }
}
```

This MyArrayList is not *type-safe*. It suffers from the following drawbacks:

1. The upcasting to java.lang.Object is done implicitly by the compiler. But, the programmer has to explicitly downcast the Object retrieved back to their original class (e.g., String).

2. The compiler is not able to check whether the downcasting is valid at *compile-time*. Incorrect downcasting will show up only at *runtime*, as a ClassCastException. This is known as *dynamic binding* or *late binding*. For example, if you accidentally added an Integer object into the above list which is intended to hold String, the error will show up only when you try to downcast the Integer back to String - at runtime.

Why not let the compiler does the upcasting/downcasting and check for casting error, instead of leaving it to the runtime, which could be too late? Can we make the compiler to catch this error to ensure *type safety* at runtime?

3.2 Generics Classes with Parameterized Types

JDK 5 introduces the so-called generics to resolve this problem. Generics allow us to abstract over types. The class designer can design a class with a generic type. The users can create specialized instance of the class by providing the specific type during instantiation. Generics allow us to pass type information, in the form of <type>, to the compiler, so that the compiler can perform all the necessary type-check during compilation to ensure type-safety at runtime.

Let's take a look at the declaration of interface java.util.List<E>:

```java
public interface List<E> extends Collection<E> {
  abstract boolean add(E element)
  abstract void add(int index, E element)
  abstract E get(int index)
  abstract E set(int index, E element)
  abstract E remove(int index)
  boolean addAll(Collection<? extends E> c)
  boolean containsAll(Collection<?> c)
  ......
}
```

The <E> is called the *formal "type" parameter* for passing type information into the generic class. During instantiation, the *formal type parameters* are replaced by the *actual type parameters*.

The mechanism is similar to method invocation. Recall that in a method's definition, we declare the *formal parameters* for passing data into the method. During the method invocation, the *formal parameters* are substituted by the *actual arguments*. For example,

```java
// Defining a method
public static int max(int a, int b) {   // int a, int b are formal parameters
  return (a > b) ? a : b;
}

// Invoke the method: formal parameters substituted by actual parameters
int max1 = max(55, 66);   // 55 and 66 are actual parameters
int x = 77;
int y = 88;
int max2 = max(x, y);     // x and y are actual parameters
```

Formal type parameters used in the class declaration have the same purpose as the formal parameters used in the method declaration. A class can use *formal type parameters* to receive type information when an instance is created for that class. The actual types used during instantiation are called *actual type parameters*. Compare with method which passes parameters through round bracket ( ), type parameters are passed through angle bracket < >.

Let's return to the List<E>. In an actual instantiation, such as a List<String>, all occurrences of the formal type parameter E are replaced by the actual type parameter String. With this additional type information, compiler is able to perform type check during compile-time and ensure that there won't have type-casting error at runtime. For example,

```java
import java.util.List;
import java.util.ArrayList;

public class J5GenericListTest {
  public static void main(String[] args) {
    // Set E to String
    List<String> fruitLst = new ArrayList<>();   // JDK 7 supports type inference
    // List<String> fruitLst = new ArrayList<String>();   // Pre-JDK 7
    fruitLst.add("apple");
    fruitLst.add("orange");
    for (String fruit : fruitLst) {
      System.out.println(fruit);
    }
    // apple
    // orange

    // fruitLst.add(123);   // This generic list accepts String only
    // compilation error: incompatible types: int cannot be converted to String
    // fruitLst.add(new StringBuffer("Hello"));
    // compilation error: incompatible types: StringBuffer cannot be converted to String
```

```java
  }
}
```

Generic Type vs. Parameterized Type
A *generic type* is a type with *formal type parameters* (e.g. List<E>); whereas a *parameterized type* is an instantiation of a generic type with *actual type arguments* (e.g., List<String>).

Formal Type Parameter Naming Convention
Use an uppercase single-character for formal type parameter. For example,

- <E> for an element of a collection;
- <T> for type;
- <K,V> for key and value.
- <N> for number
- S, U, V, etc. for 2nd, 3rd, 4th type parameters

3.3 Example 2: A Generic Class GenericBox

In this example, a class called GenericBox, which takes a generic type parameter E, holds a content of type E. The constructor, getter and setter work on the parameterized type E. The toString() reveals the actual type of the content.

```java
// A Generic Box with a content
public class GenericBox<E> {
  private E content;  // private variable of generic type E
  public GenericBox(E content) {  // constructor
    this.content = content;
  }

  public E getContent() {  // getter
    return content;
  }

  public void setContent(E content) {  // setter
    this.content = content;
  }

  public String toString() {  // describe itself
    return "GenericBox[content=" + content + "(" + content.getClass() + ")]";
  }
}
```

The following test program creates GenericBoxes with various types (String, Integer and Double). Take note that JDK 5 also introduces auto-boxing and unboxing to convert between primitives and wrapper objects.

```java
public class GenericBoxTest {
  public static void main(String[] args) {
    GenericBox<String> box1 = new GenericBox<>("hello");  // JDK 7
      ↪ supports type inference
    String str = box1.getContent();  // no explicit downcasting needed
    System.out.println(box1);
    // GenericBox[content=hello(class java.lang.String)]

    GenericBox<Integer> box2 = new GenericBox<>(123);  // int auto-box to Integer
    int i = box2.getContent();        // Integer auto-unbox to int
    System.out.println(box2);
    // GenericBox[content=123(class java.lang.Integer)]

    GenericBox<Double> box3 = new GenericBox<>(55.66);  // double auto-box to
      Double
    double d = box3.getContent();     // Double auto-unbox to double
    System.out.println(box3);
    // GenericBox[content=55.66(class java.lang.Double)]
  }
}
```

3.4 (JDK 7) Improved Type Inference for Generic Instance Creation with the Diamond Operator <>

Before JDK 7, to create an instance of the above GenericBox, you need to specify to type in the constructor:

```java
GenericBox<String> box1 = new GenericBox<String>("hello");
```

JDK 7 introduces the type *inference* to shorten the code, as follows:

```java
GenericBox<String> box1 = new GenericBox<>("hello");  // type inferred
   ↪ from the variable
```

3.5 Type Erasure

From the previous example, it seems that compiler substituted the parameterized type E with the actual type (such as String, Integer) during instantiation. If this is the case, the compiler would need to create a new class for each actual type (similar to C++'s template). In fact, the compiler replaces all reference to parameterized type E with java.lang.Object. For example, the above GenericBox is compiled as follows, which is compatible with the code without generics:

```java
public class GenericBox {
  private Object content;                 // private variable
  public GenericBox(Object content) {   // Constructor
    this.content = content;
  }

  public Object getContent() {            // getter
    return content;
  }

  public void setContent(Object content) {  // setter
    this.content = content;
  }

  public String toString() {              // describe itself
    return "GenericBox[content=" + content + "(" + content.getClass() + ")]";
  }
}
```

The compiler performs the type checking and inserts the required downcast operator when the methods are invoked:

```java
// Constructor: public GenericBox(E content)
GenericBox<String> box1 = new GenericBox<>("hello");  // Knowing E = String, compiler performs the type check

// Getter: public E getContent()
String str = (String)box1.getContent();  // Compiler inserts the downcast operator to downcast Object to String
```

In this way, the same class definition is used for all the types. Most importantly, the bytecode are compatible with those without generics. This process is called type erasure.

For example, GenericBox<Integer> and GenericBox<String> are compiled into the same runtime class GenericBox.

```java
public class GenericBoxTypeTest {
  public static void main(String[] args) {
    GenericBox<Integer> box1 = new GenericBox<>(123);
    GenericBox<String> box2 = new GenericBox<>("hello");
    System.out.println(box1.getClass() == box2.getClass()); //true (same
                                                            // runtime class)
    System.out.println(box1.getClass()); //class GenericBox
    System.out.println(box2.getClass()); //class GenericBox
  }
}
```

3.6 Example 3: Type-Safe MyGenericArrayList<E>

Let's return to the MyArrayList example. With the use of generics, we can rewrite our program as follows:

```java
// A dynamically allocated array with generics
public class MyGenericArrayList<E> { // E is the generic type of the elements
  private int size;                  // number of elements
  private Object[] elements;         // Need to use an Object[], not E[]

  public MyGenericArrayList() {   // constructor
    elements = new Object[10];    // allocate initial capacity of 10
    size = 0;
  }

  public void add(E e) {
    if (size >= elements.length) {
      // allocate a larger array and copy over
      Object[] newElements = new Object[size + 10];
      for (int i = 0; i < size; ++i) {
        newElements[i] = elements[i];
      }
      elements = newElements;
    }
    elements[size] = e;
    ++size;
  }

  @SuppressWarnings("unchecked")
  public E get(int index) {
    if (index >= size) {
      throw new IndexOutOfBoundsException("Index: " + index
        + ", Size: " + size);
    }

    return (E)elements[index]; // triggers an "unchecked cast" warning
```

```java
    }

    public int size() {
      return size;
    }
}
```

Dissecting the Program

MyGenericArrayList<E> declare a generics class with a *formal type parameter* <E>. During an actual invocation, e.g., MyGenericArrayList<String>, a specific type <String>, or *actual type parameter*, replaced the formal type parameter <E>.

Type Erasure

Behind the scene, generics are implemented by the Java compiler as a front-end conversion called *erasure*, which translates or rewrites code that uses generics into non-generic code to ensure backward compatibility. This conversion erases all generic type information. The formal type parameter, such as <E>, are replaced by Object by default (or by the upper bound of the type). When the resulting code is not type correct, the compiler insert a type casting operator.

Hence, the translated code is as follows:

```java
// The translated code
public class MyGenericArrayList {
  private int size;     // number of elements
  private Object[] elements;

  public MyGenericArrayList() {  // constructor
    elements = new Object[10];      // allocate initial capacity of 10
    size = 0;
  }

  // Compiler replaces E with Object, but check e is of type E,
  //  when invoked to ensure type-safety
  public void add(Object e) {
    if (size < elements.length) {
      elements[size] = e;
    } else {
      // allocate a larger array and copy over
      Object[] newElements = new Object[size + 10];
      for (int i = 0; i < size; ++i) {
        newElements[i] = elements[i];
      }
      elements = newElements;
    }
```

```java
      ++size;
    }

    // Compiler replaces E with Object, and insert downcast operator
    //  (E<E>) for the return type when invoked
    public Object get(int index) {
      if (index >= size) {
        throw new IndexOutOfBoundsException("Index: " + index
          + ", Size: " + size);
      }

      return (Object)elements[index];
    }

    public int size() {
      return size;
    }
  }
```

When the class is instantiated with an actual type parameter, e.g. MyGenericArrayList<String>, the compiler performs type check to ensures add(E e) operates on only String type. It also inserts the proper downcasting operator to match the return type E of get(). For example,

```java
public class MyGenericArrayListTest {
  public static void main(String[] args) {
    // type-safe to hold a list of Strings
    MyGenericArrayList<String> strLst = new MyGenericArrayList<>();  // JDK 7
        ↪ diamond operator
    strLst.add("alpha");          // compiler checks if argument is of type String
    strLst.add("beta");

    for (int i = 0; i < strLst.size(); ++i) {
      String str = strLst.get(i);  // compiler inserts the downcasting
                                   // operator (String)
      System.out.println(str);
    }

    // strLst.add(123);  // compiler detected argument is NOT String, issues
        compilation error
    // compilation error: incompatible types: int cannot be converted to String
  }
}
```

With generics, the compiler is able to perform type checking during compilation to ensure type safety at runtime.

Unlike "template" in C++, which creates a new type for each specific parameterized type, in Java, a generics class is only compiled once, and there is only one single class file which is used to create instances for all the specific types.

3.7 Backward Compatibility

If you compile a Pre-JDK 5 program using JDK 5 and above compiler, you will receive some warning messages to warn you about the unsafe operations, i.e., the compiler is unable to check for the type (because it was not informed of the type via generics) and ensure type-safety at runtime. You could go ahead and execute the program with warnings. For example,

```java
// Pre-JDK 5 Collection without generics
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListPreJ5Test {
  public static void main(String[] args) {
    List lst = new ArrayList();  // A List contains instances of Object
    lst.add("alpha");  // add() takes Object. String upcasts to Object implicitly
    lst.add("beta");
    System.out.println(lst);     // [alpha, beta]

    Iterator iter = lst.iterator();
    while (iter.hasNext()) {
      String str = (String)iter.next();  // explicitly downcast from
                                         // Object back to String
      System.out.println(str);
    }
    // alpha
    // beta
  }
}
```

Command window

```
> javac ArrayListPreJ5Test.java
Note: ArrayListPreJ5Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

> javac -Xlint:unchecked ArrayListPreJ5Test.java
ArrayListPreJ5Test.java:9: warning: [unchecked] unchecked call to
    add(E) as a member of the raw type List
......
```

3.8 Generic Methods

Other than generic class described in the above section, we can also define methods with generic types.

For example, the java.lang.String class, which is non-generic, contain a generic method .transform() defined as follows:

```java
// Class java.lang.String
public <R> R transform(Function<? super String, ? extends R> f) // JDK 12
```

A generic method should declare formal type parameters, which did not appear in the class statement, (e.g. <R>) *preceding the return type*. The formal type parameters can then be used as *placeholders* for return type, method's parameters and local variables within a generic method, for proper type-checking by compiler.

Example

```java
import java.util.List;
import java.util.ArrayList;

public class GenericMethodTest {
  // A static generic method to append an array to a List
  public static <E> void Array2List(E[] arr, List<E> lst) {
    for (E e : arr) lst.add(e);
  }

  public static void main(String[] args) {
    // Set E to Integer
    Integer[] arr = {55, 66};  // int auto-box to Integer
    List<Integer> lst = new ArrayList<>();
    Array2List(arr, lst);
    System.out.println(lst);   // [55, 66]

    String[] strArr = {"alpha", "beta", "charlie"};
    // Array2List(strArr, lst);
    // compilation error: method Array2List in class GenericMethodTest
    // cannot be applied to given types
  }
}
```

In this example, we define a static generic method Array2List() to append an array of generic type E to a List<E>. In the method definition, we need to declare the generic type <E>

before the return-type void.

Similar to generic class, when the compiler translates a generic method, it replaces the formal type parameters using *erasure*. All the generic types are replaced with type Object by default (or the upper bound of type). The translated version is as follows:

```java
public static void Array2List(Object[] arr, List lst) {
  for (Object e : arr) lst.add(e);
}
```

When the method is invoked, the compiler performs type check and inserts downcasting operator during retrieval.

Generics have an optional syntax for specifying the type for a generic method. You can place the actual type in angle brackets <>, between the dot operator and method name. For example,

```java
GenericMethodTest.<Integer>Array2List(arr, lst);
```

The syntax makes the code more readable and also gives you control over the generic type in situations where the type might not be obvious.

3.9 Generic Subtypes

Knowing that String is a subtype of Object. Consider the following lines of codes:

```java
// String is a subtype of Object
Object obj = "hello";     // A supertype reference holding a subtype instance
System.out.println(obj);  // hello

// But ArrayList<String> is not a subtype of ArrayList<Object>
ArrayList<Object> lst = new ArrayList<String>();
// compilation error: incompatible types: ArrayList<String> cannot be converted to
    ArrayList<Object>
```

When we try to upcast ArrayList<String> to ArrayList<Object>, it trigger a compilation error "incompatible types". This is because ArrayList<String> is NOT a subtype of

ArrayList<Object>, even through String is a subtype of Object.

This error is against our intuition on inheritance. Why? Consider these two statements:

```
1 List<String>  strLst = new ArrayList<>();   // 1
  List<Object> objLst = strLst;              // 2
3 // compilation error: incompatible types: List<String> cannot be converted to List<Object>
```

Line 2 generates a compilation error. But if line 2 succeeds and some arbitrary objects are added into objLst, strLst will get "corrupted" and no longer contains only Strings, as references objLst and strLst share the same value.

Hence, List<String> is NOT a subtype of List<Object>, although String is a subtype of Object.

On the other hands, the following is valid:

```
// ArrayList is a subtype of List
List<String> lst = new ArrayList<>();  // valid
```

That is, ArrayList<String> is a subtype of List<String>, since ArrayList is a subtype of List and both have the same parametric type String.

In summary:

1. Different instantiation of the same generic type for different concrete type arguments (such as List<String>, List<Integer>, List<Object>) have NO type relationship.
2. Instantiations of super-sub generic types for the same actual type argument exhibit the same super-sub type relationship, e.g., ArrayList<String> is a subtype of List<String>.

Array Subtype?

String[] is a subtype of Object[]. But if you upcast a String[] to Object[], you cannot re-assign value of non-String type. For example,

```
import java.util.Arrays;
public class ArraySubtypeTest {
  public static void main(String[] args) {
    String[] strArr = {"apple", "orange"};
    Object[] objArr = strArr;  // upcast String[] to Object[]
```

```java
        System.out.println(Arrays.toString(objArr));
        objArr[0] = 123;   // compile ok, runtime error
        // Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
    }
}
```

Arrays carry runtime type information about their component type. Hence, you CANNOT use E[] in your generic class, but need to use Object[], as in the MyGenericArrayList<E>.

3.10 Wildcards <? extends T>, <? super T> and <?>

Suppose that we want to write a generic method called printList(List<.>) to print the elements of a List. If we define the method as printList(List<Object> lst), then it can only accept an argument of List<object>, but not List<String> or List<Integer>. For example,

```java
import java.util.List;
import java.util.ArrayList;
public class GenericWildcardTest {
  // Accepts List<Object>, NOT list<String>, List<Integer>, etc.
  public static void printList(List<Object> lst) {
    for (Object o : lst) {
      System.out.println(o);
    }
  }

  public static void main(String[] args) {
    List<Object> objLst = new ArrayList<>();   // ArrayList<Object> inferred
    objLst.add(11);   // int auto-box to Integer, upcast to Object
    objLst.add(22);
    printList(objLst);
    // 11
    // 22

    List<String> strLst = new ArrayList<>();   // ArrayList<String> inferred
    strLst.add("one");
    // printList(strLst);   // only accept List<Object>
    // error: incompatible types: List<String> cannot be converted to List<Object>
  }
}
```

Unbounded Wildcard <?>

To resolve this problem, a wildcard (?) is provided in generics, which stands for *any unknown type*. For example, we can rewrite our printList() as follows to accept a List of any unknown type.

```java
public static void printList(List<?> lst) {
  for (Object o : lst) System.out.println(o);
}
```

The unbounded wildcard <?> is, at times, too relax in type.

Upper Bounded Wildcard <? extends T>

To write a generic method that works on List<Number> and the subtypes of Number, such as List<Integer>, List<Double>, we could use an upper bounded wildcard <? extends Number>.

In general, the wildcard <? extends T> stands for type T and T's subtypes.

For example,

```java
import java.util.List;
public class GenericUpperBoundedWildcardTest {
  // Generic method which accepts List<Number>
  //   and Number's subtypes such as Integer, Double
  public static double sumList(List<? extends Number> lst) {
    double sum = 0.0;
    for (Number num : lst) {
      sum += num.doubleValue();
    }
    return sum;
  }

  public static void main(String[] args) {
    List<Integer> intLst = List.of(1, 2, 3);  // JDK 9 unmodifiable List
    System.out.println(sumList(intLst));      // 6.0

    List<Double> doubleLst = List.of(1.1, 2.2, 3.3);
    System.out.println(sumList(doubleLst));  // 6.6

    List<String> strLst = List.of("apple", "orange");
    // sumList(strLst);
    // error: incompatible types: List<String> cannot be converted to List<? extends
       Number>
  }
}
```

List<? extends Number> accepts List of Number and any subtypes of Number, e.g., List<Integer> and List<Double>.

Another example,

```java
// List<Number> lst = new ArrayList<Integer>();
// compilation error: incompatible types: ArrayList<Integer> cannot be converted to List<Number>

List<? extends Number> lst = new ArrayList<Integer>();  // valid
```

Revisit Unbounded Wildcard <?>
Clearly, <?> can be interpreted as <? extends Object>, which accepts ALL Java classes. You should use <?> only if:

1. The implementation depends only on methods that provided in the Object class.
2. The implementation does not depend on the type parameter.

Lower Bounded Wildcard <? super T>
The wildcard <? super T> matches type T, as well as T's supertypes. In other words, it specifies the lower bound type.

Suppose that we want to write a generic method that puts an Integer into a List. To maximize flexibility, we also like the method to work on List<Integer>, as well as List<Number>, List<Object> that can hold Integer. In this case, we could use the less restrictive lower bounded wildcard <? super Integer>, instead of simply List<Integer>. For example,

```java
import java.util.List;
import java.util.ArrayList;

public class GenericLowerBoundedWildcardTest {
  // Generic method which accepts List<Integer>
  //  and Integer's supertypes such as Number and Object
  public static void addIntToList(List<? super Integer> lst, int num) {
    lst.add(num);
  }

  public static void main(String[] args) {
    List<Integer> intLst = new ArrayList<>(); // modifiable List
    intLst.add(1);
    intLst.add(2);
    System.out.println(intLst);  // [1, 2]
    addIntToList(intLst, 3);
    System.out.println(intLst);  // [1, 2, 3]

```

```java
    List<Number> numLst = new ArrayList<>();
    numLst.add(1.1);
    numLst.add(2.2);
    System.out.println(numLst);  // [1.1, 2.2]
    addIntToList(numLst, 3);
    System.out.println(numLst);  // [1.1, 2.2, 3]

    List<String> strLst = new ArrayList<>();
    // addIntToList(strLst, "hello");
    // error: incompatible types: List<String> cannot be converted to List<? super Integer>
  }
}
```

3.11 Example: Upper and Lower Bounded Wildcards

```java
import java.util.*;

@FunctionalInterface
interface MyConsumer<T> {
  void accept(T t);  // public abstract
}

// Need 3 levels of class hierarchy for testing
class C1 {
  protected String value;

  public C1(String value) {
    this.value = value;
  }

  public void methodC1() {
    System.out.println(this + " runs methodC1()");
  }

  @Override
  public String toString() {
    return "C1[" + value + "]";
  }
}

class C2 extends C1 {
  public C2(String value) {
    super(value);
  }

  public void methodC2() {
    System.out.println(this + " runs methodC2()");
```

```java
  }

  @Override
  public String toString() {
    return "C2[" + value + "]";
  }
}

class C3 extends C2 {
  public C3(String value) {
    super(value);
  }

  public void methodC3() {
    System.out.println(this + " runs methodC3()");
  }

  @Override
  public String toString() {
    return "C3[" + value + "]";
  }
}

public class GenericUpperLowerWildcardTest {
  // For a specific T only
  public static <T> T processAll1(Collection<T> coll,
                                  MyConsumer<T> consumer) {
    T last = null;
    for (T t : coll) {
      last = t;
      consumer.accept(t);
    }
    return last;
  }

  // Lower bounded wildcard
  public static <T> T processAll2(Collection<T> coll,
                                  MyConsumer<? super T> consumer) {
    T last = null;
    for (T t : coll) {
      last = t;
      consumer.accept(t); // t supports all its supertype's operations
    }
    return last;
  }

  // Lower bounded and upper bounded wildcards
  public static <T> T processAll3(Collection<? extends T> coll,
                                  MyConsumer<? super T> consumer) {
    T last = null;
    for (T t : coll) {      // T's subtype elements can be upcast to T
      last = t;
```

```java
            consumer.accept(t);  // t supports all its supertype's operations
        }
        return last;
    }

    public static void main(String[] args) {
        // Set T to C2
        // Try processAll1(Collection<C2>, MyConsumer<C2>)
        Collection<C2> fruits = List.of(new C2("apple"), new C2("orange"));
        MyConsumer<C2> consumer1 = C2::methodC2;  // Can use C2's methods
        C2 result1 = processAll1(fruits, consumer1);
        // C2[apple] runs methodC2()
        // C2[orange] runs methodC2()
        System.out.println(result1);
        // C2[orange]

        // Try processAll2(Collection<C2>, MyConsumer<C1 super C2>)
        MyConsumer<C1> consumer2 = C1::methodC1;
        // Can use only C1's methods. But subtype C2 supports all C1's methods
        // processAll1(fruits, consumer2); // wrong type for consumer2 in processAll1()
        // error: method processAll1 in class GenericWildardTest cannot be applied to given types
        C2 result2 = processAll2(fruits, consumer2);
        // C2[apple] runs methodC1()
        // C2[orange] runs methodC1()
        System.out.println(result2);
        // C2[orange]

        // Try processAll3(Collection<C3 extends C2>, MyConsumer<C1 super C2>)
        Collection<C3> coffees = List.of(new C3("espresso"), new C3("latte"));
        C2 result3 = processAll3(coffees, consumer2);
        // C3[espresso] runs methodC1()
        // C3[latte] runs methodC1()
        System.out.println(result3);
        // C3[latte]
        processAll3(coffees, consumer2).methodC3();
        // C3[espresso] runs methodC1()
        // C3[latte] runs methodC1()
        // C3[latte] runs methodC3()

        // Try subclass List of Collection
        List<C3> animals = List.of(new C3("tiger"), new C3("lion"));
        C2 result4 = processAll3(animals, consumer2);
        // C3[tiger] runs methodC1()
        // C3[lion] runs methodC1()
        System.out.println(result4);
        // C3[lion]
    }
}
```

In summary:

1. List<String> is NOT a subtype of List<Object>, but ArrayList<String> is a subtype of List<String> and can be upcasted.
2. Upper Bounded Wildcard <? extends T> for collection: To be able to process Collection of T and T's subtypes, use Collection<? extends T>. For example, printList<? extends Number> works on printList<Number>, printList<Integer>, printList<Double>, etc.
3. Lower Bounded Wildcard <? super T> for operation: The type T inherits and supports all its supertypes' operations. A operation that is operating on T's supertype also works on T, because T support all its supertype's operation. For maximum flexibility in operation on T, we could use <? super T> to operation on T's supertypes.

3.12 Bounded Type Parameters

Upper Bounded Type Parameters <T extends TypeName>
A bounded parameter type is a generic type that specifies a bound for the generic, in the form of <T extends TypeName>, e.g., <T extends Number> accepts Number and its subclasses (such as Integer and Double).

For example, the static method add() takes a type parameter <T extends Number>, which accepts Number and its subclasses (such as Integer and Double).

```java
public class UpperBoundedTypeParamAddTest {
  public static <T extends Number> double add(T first, T second) {
    // Need to use only methods defined in Number, such as doubleValue
    // Subtypes Integer and Double inherit and support these methods too.
    return first.doubleValue() + second.doubleValue();
  }

  public static void main(String[] args) {
    System.out.println(add(55, 66));      // int -> Integer. T is Integer.
    System.out.println(add(5.5f, 6.6f)); // float -> Float. T is Float.
    System.out.println(add(5.5, 6.6));    // double -> Double. T is Double.
    System.out.println(add(55, 6.6)); // int -> double -> Double. T is Double.

    // System.out.println(add("apple", "orange"));
    // compilation error: method add in class UpperBoundedTypeParameterTest
    // cannot be applied to given types;
  }
}
```

How the compiler treats the bounded generics?
As mentioned, by default, all the generic types are replaced with type Object during the code translation. However, in the case of <T extends Number>, the generic type is replaced

by the type Number, which serves as the *upper bound* of the generic types.

For example,

```java
public class UpperBoundedTypeParamMaximumTest {
  public static <T extends Comparable<T>> T maximum(T x, T y) {
    // Need to restrict T to Comparable and its subtype for .compareTo()
    return (x.compareTo(y) > 0) ? x : y;
  }

  public static void main(String[] args) {
    System.out.println(maximum(55, 66));      // 66
    System.out.println(maximum(6.6, 5.5));    // 6.6
    System.out.println(maximum("Monday", "Tuesday"));   // Tuesday
  }
}
```

By default, Object is the *upper-bound* of the parameterized type.
<T extends Comparable<T>> changes the upper bound to the Comparable interface, which declares an abstract method compareTo() for comparing two objects.

The compiler translates the above generic method to the following codes:

```java
public static Comparable maximum(Comparable x, Comparable y) {   // replace T by upper bound type Comparable
  // Compiler checks x, y are of the type Comparable
  // Compiler inserts a type-cast for the return value
  return (x.compareTo(y) > 0) ? x : y;
}
```

When this method is invoked, e.g. via maximum(55, 66), the primitive ints are auto-boxed to Integer objects, which are then implicitly upcasted to Comparable. The compiler checks the type to ensure type-safety. The compiler also inserts an explicit downcast operator for the return type. That is,

Command window

```
(Comparable)maximum(55, 66);
(Comparable)maximum(6.6, 5.5);
(Comparable)maximum("Monday", "Tuesday");
```

We do not have to pass an actual type argument to a generic method. The compiler infers the type argument automatically, based of the type of the actual argument passed into the method.

Bounded Type Parameter for Generic Class The bounded type parameter <T extends ClassName> can also be applied to generic class, e.g.,

```java
public class MagicNumber<T extends Number> {
  private T value;

  public MagicNumber(T value) {
    this.value = value;
  }

  public boolean isMagic() {
    return value.intValue() == 9;
  }

  public String toString() {
    return "MagicNumber[value=" + value +"]";
  }

  public static void main(String[] args) {
    MagicNumber<Double> n1 = new MagicNumber<>(9.9);
    System.out.println(n1);            // MagicNumber[value=9.9]
    System.out.println(n1.isMagic()); // true

    MagicNumber<Float> n2 = new MagicNumber<>(1.23f);
    System.out.println(n2);            // MagicNumber[value=1.23]
    System.out.println(n2.isMagic()); // false

    MagicNumber<Number> n3 = new MagicNumber<>(1);
    System.out.println(n3);            // MagicNumber[value=1]
    System.out.println(n3.isMagic()); // false

    // MagicNumber<String> n4 = new MagicNumber<>("hello");
    // error: type argument String is not within bounds of type-variable T
  }
}
```

Lower Bounded Type Parameters <T super Class>
Not useful and hence, not supported.