Object-Oriented Programming and Design with Java

HaQT

HUS
VNU UNIVERSITY OF SCIENCE

Table of Contents

Part I Java Basics

Part II Object-Oriented Programming

Part III Collections Framework

Part IV Correctness, Robustness, Efficiency

Part V Design Patterns

Part VI Java Generics

1 Java Generics By Examples

1.1 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 <>.

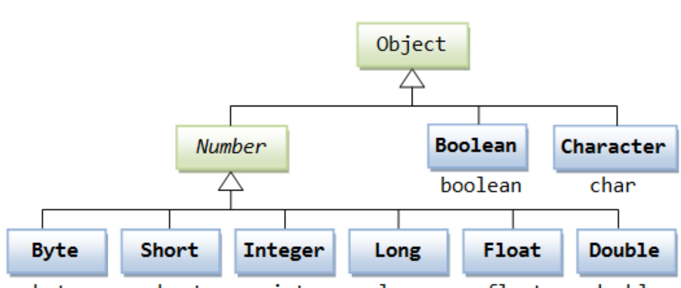
The primary usage of generics is to **abstract over types** for the Collections Framework.

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.

1.1.1 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,

```
// 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,

```
// 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,

```
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
    }
}
```

1.1.2 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]);
    }

```

```java
    // Use for-each loop on Collection
    List<String> coffeeList = new ArrayList<>();
    coffeeList.add("espresso");
    coffeeList.add("latte");
    for (String coffee : coffeeList) {
      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
public class MyMutableInteger {
    private int value;   // private variable, mutable via setter

    // Constructor
    public MyMutableInteger(int value) {
        this.value = value;
    }

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

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

```java
import java.util.Arrays;

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)
   }
}
```

1.1.3 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
    }
}
```

1.2 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.

1.2.1 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:

```
public class ArrayList<E> implements List<E> ... {
   public void add(int index, E element)
   public boolean add(E e)
   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)
   public E set(int index, E element)
   public List<E> subList(int fromIndex, int toIndex)
   public Iterator<E> iterator()
   public ListIterator<E> listIterator()
   public ListIterator<E> listIterator(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,

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

public class GenericArrayListTest {
   public static void main(String[] args) {
      // Set "E" to "String"
      ArrayList<String> fruitList = new ArrayList<String>();
      fruitList.add("apple");
      fruitList.add("orange");
      System.out.println(fruitList); // [apple, orange]
```

```
        // JDK 5 also introduces the for-each loop
        for (String str: fruitList) {   // We need to know type of elements
           System.out.println(str);
        }
        // apple
        // orange

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

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

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

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

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

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

1.2.6 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:

```java
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";
```

```java

      // 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>
      // 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.

1.3 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.

1.3.1 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[ ].

```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));
    }
}
```

```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?

1.3.2 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
   }
}
```

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

1.3.3 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.

```
// A Generic Box with a content
public class GenericBox<E> {
   private E content; // private variable of generic type E

   public GenericBox(E content) {
      this.content = content;
   }

   public E getContent() {
```

```java
        return content;
    }

    public void setContent(E content) {
        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)]
    }
}
```

1.3.4 (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
// Type inferred from the variable
GenericBox<String> box1 = new GenericBox<>("hello");
```

1.3.5 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
  }
}
```

1.3.6 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
```

```java
    private Object[] elements;            // Need to use an Object[], not E[]

    public MyGenericArrayList() {
        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
    }

    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
public class MyGenericArrayList {
  private int size;       // number of elements
  private Object[] elements;

  public MyGenericArrayList() {
    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 {
      Object[] newElements = new Object[size + 10];  // Allocate a larger array
      for (int i = 0; i < size; ++i) {
        newElements[i] = elements[i];
      }
      elements = newElements;
    }
    ++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.

1.3.7 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
  ......
```

1.3.8 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:

```
// 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. For 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> list) {
    for (E e : arr) list.add(e);
  }

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

    String[] strArr = {"alpha", "beta", "charlie"};
    // array2List(strArr, list);
    // 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,  list);
```

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.

1.3.9 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> list  = 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:

```java
1 List<String>  strList  = new ArrayList<>();   // 1
2 List<Object> objList = 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 $objList$, $strList$ will get "corrupted" and no longer contains only Strings, as references $objList$ and $strList$ 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:

```java
// ArrayList is a subtype of List
List<String> list = 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,

```java
import java.util.Arrays;

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

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

Suppose that we want to write a generic method called $printList(List\langle\cdot\rangle)$ to print the elements of a List. If we define the method as $printList(List\langle$Object$\rangle$

list), then it can only accept an argument of List<object>, but not List<String> or List<Integer>. For example,

```
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> list) {
    for (Object o : list) {
      System.out.println(o);
    }
  }

  public static void main(String[] args) {
    List<Object> objList = new ArrayList<>();  // ArrayList<Object> inferred
    objList.add(11);  // int auto-box to Integer, upcast to Object
    objList.add(22);
    objList.add(33);
    printList(objList);
    // 11
    // 22
    // 33

    List<String> strList = new ArrayList<>();  // ArrayList<String> inferred
    strList.add("one");
    // printList(strList);  // 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.

```
public static void printList(List<?> list) {
  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]

    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>
    }
}
```

1.3.11 Example: Upper and Lower Bounded Wildcards

```
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()");
    }

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

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

  public static void main(String[] args) {
```

```java
        // 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.

1.3.12 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;

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

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

  @Override
  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.

2 Java Generics

2.1 Introducing Generics

2.1.1 Why Use Generics?

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

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

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

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

```java
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

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

```java
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast
```

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

2.1.2 Generic Types

A Simple Box Class

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

```java
public class Box {
  private Object object;

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

  public Object get() {
    return object;
  }
}
```

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

A Generic Version of the Box Class

A **generic** class is defined with the following format:

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

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

To update the Box class to use generics, you create a generic type declaration by changing the code "public class Box" to "public class Box<T>". This introduces the type variable, T, that can be used anywhere inside the class.

With this change, the Box class becomes:

```java
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    private T t; // T stands for "Type"

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

    public T get() {
        return t;
    }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

This same technique can be applied to create generic interfaces.

Type Parameter Naming Conventions

By convention, type parameter names are single, uppercase letters. This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason: without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S, U, V etc. - 2nd, 3rd, 4th types

Invoking and Instantiating a Generic Type

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

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

You can think of a generic type invocation as being similar to an ordinary method invocation, but instead of passing an argument to a method, you are passing a type argument — Integer in this case — to the Box class itself.

Type Parameter and **Type Argument Terminology**

Many developers use the terms "type parameter" and "type argument" interchangeably, but these terms are not the same. When coding, one provides type arguments in order to create a parameterized type. Therefore, the T in Foo<T> is a type parameter and the String in Foo<String> is a type argument. This section observes this definition when using these terms.

Like any other variable declaration, this code does not actually create a new Box object. It simply declares that integerBox will hold a reference to a "Box of Integer", which is how Box<Integer> is read.

An invocation of a generic type is generally known as a parameterized type.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

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

The Diamond

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

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

For more information on diamond notation and type inference, see the Type Inference section.

Multiple Type Parameters

As mentioned previously, a generic class can have multiple type parameters. For example, the generic OrderedPair class, which implements the generic Pair interface:

```java
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

```java
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;
    }
}
```

The following statements create two instantiations of the OrderedPair class:

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

The code, new OrderedPair<String, Integer>(), instantiates K as a String and V as an Integer. Therefore, the parameter types of OrderedPair's constructor are String and Integer, respectively. Due to autoboxing, it is valid to pass a String and an int to the class.

As mentioned in The Diamond section, because a Java compiler can infer the K and V types from the declaration OrderedPair<String, Integer>, these statements can be shortened using diamond notation:

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

To create a generic interface, follow the same conventions as for creating a generic class.

Parameterized Types

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

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

2.1.3 Raw Types

A *raw type* is the name of a generic class or interface without any type arguments. For example, given the generic Box class:

```java
public class Box<T> {
  public void set(T t) {
    /* ... */
  }

  // ...
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

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

If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Therefore, Box is the raw type of the generic type Box<T>. However, a non-generic class or interface type is not a raw type.

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a Box gives you Objects. For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;                // Ok
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();            // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;      // warning: unchecked conversion
```

You also get a warning if you use a raw type to invoke generic methods defined in the corresponding generic type:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8); // warning: unchecked invocation to set(T)
```

The warning shows that raw types bypass generic type checks, deferring the catch of unsafe code to runtime. Therefore, you should avoid using raw types.

Unchecked Error Messages

As mentioned previously, when mixing legacy code with generic code, you may encounter warning messages similar to the following:

Command window

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

This can happen when using an older API that operates on raw types, as shown in the following example:

```java
public class WarningDemo {
    public static void main(String[] args) {
        Box<Integer> bi = createBox();
    }

    public static Box createBox() {
        return new Box();
    }
}
```

The term "unchecked" means that the compiler does not have enough type information to perform all type checks necessary to ensure type safety. The "unchecked" warning is disabled, by default, though the compiler gives a hint. To see all "unchecked" warnings, recompile with -Xlint:unchecked.

Recompiling the previous example with -Xlint:unchecked reveals the following additional information:

Command window

```
WarningDemo.java:4: warning: [unchecked] unchecked conversion
found    : Box
required: Box<java.lang.Integer>
    bi = createBox();
                  ^
1 warning
```

To completely disable unchecked warnings, use the -Xlint:-unchecked flag. The @SuppressWarnings("unchecked") annotation suppresses unchecked warnings. If you are unfamiliar with the @SuppressWarnings syntax, see the section Annotations.

2.1.4 Generic Methods

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

The syntax for a generic method includes a list of type parameters, inside angle brackets, which appears before the method's return type. For static generic methods, the type parameter section must appear before the method's return type.

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

```java
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());
  }
}
```

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

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

  public void setKey(K key) {
    this.key = key;
  }

  public void setValue(V value) {
    this.value = value;
  }

  public K getKey() {
```

```
        return key;
    }

    public V getValue() {
        return value;
    }
}
```

The complete syntax for invoking this method would be:

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

The type has been explicitly provided, as shown in bold. Generally, this can be left out and the compiler will infer the type that is needed:

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

This feature, known as type inference, allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets. This topic is further discussed in the following section, Type Inference.

2.1.5 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. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, which in this example is Number. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

```java
public class Box<T> {
  private T t;

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

  public T get() {
    return t;
  }

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

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

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String:

Command window

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot
be applied to (java.lang.String)
  integerBox.inspect("10");
                    ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```java
public class NaturalNumber<T extends Integer> {
  private T n;

  public NaturalNumber(T n) {
    this.n = n;
```

```java
    }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

The isEven() method invokes the intValue() method defined in the Integer class through n.

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have multiple bounds:

```java
<T extends B1 & B2 & B3>
```

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```java
class A { /* ... */ }

interface B { /* ... */ }

interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```java
class D <T extends B & A & C> { /* ... */ }  // Compile-time error
```

2.1.6 Generic Methods and Bounded Type Parameters

Bounded type parameters are key to the implementation of generic algorithms. Consider the following method that counts the number of elements in an array T[] that are greater than a specified element elem.

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

The implementation of the method is straightforward, but it does not compile because the greater than operator (>) applies only to primitive types such as short, int, double, long, float, byte, and char. You cannot use the > operator to compare objects. To fix the problem, use a type parameter bounded by the Comparable<T> interface:

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

The resulting code will be:

```java
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;
}
```

2.1.7 Generics, Inheritance, and Subtypes

As you already know, it is possible to assign an object of one type to an object of another type provided that the types are compatible. For example, you can assign an Integer to an Object, since Object is one of Integer's supertypes:

```
Object someObject = new Object();
Integer someInteger = new Integer(10);
someObject = someInteger;   // Ok
```

In object-oriented terminology, this is called an "is a" relationship. Since an Integer is a kind of Object, the assignment is allowed. But Integer is also a kind of Number, so the following code is valid as well:

```
public void someMethod(Number n) {
    /* ... */
}

someMethod(new Integer(10));    // Ok
someMethod(new Double(10.1));   // Ok
```

The same is also true with generics. You can perform a generic type invocation, passing Number as its type argument, and any subsequent invocation of add will be allowed if the argument is compatible with Number:

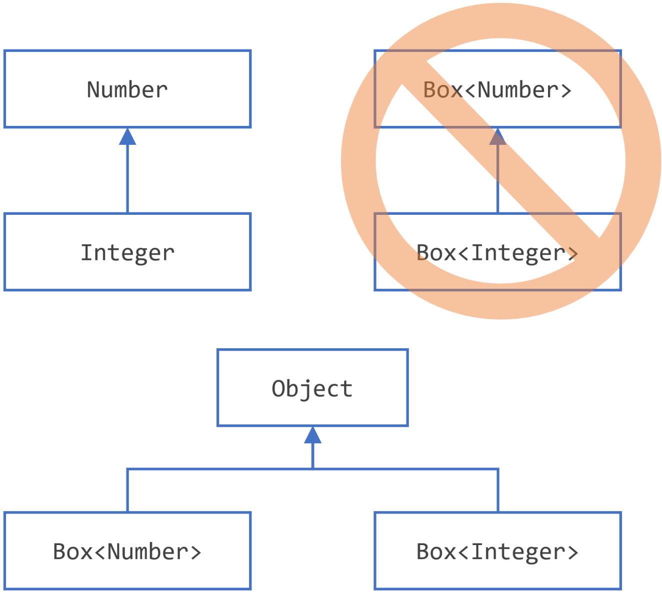
```
Box<Number> box = new Box<Number>();
box.add(new Integer(10));   // OK
box.add(new Double(10.1));  // OK
```

Now consider the following method:

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

What type of argument does it accept? By looking at its signature, you can see that it accepts a single argument whose type is Box<Number>. But what does that mean? Are you allowed to pass in Box<Integer> or Box<Double>, as you might expect? The answer is "no", because Box<Integer> and Box<Double> are not subtypes of Box<Number>.

This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn. Box<Integer> is not a subtype of Box<Number> even though Integer is a subtype of Number.



Subtyping parameterized types.

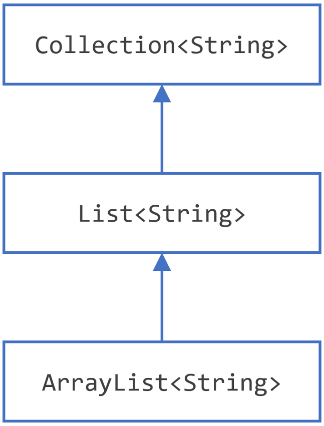
Note: Given two concrete types A and B, for example, Number and Integer, MyClass<A> has no relationship to MyClass<B>, regardless of whether or not A and B are related. The common parent of MyClass<A> and MyClass<B> is Object.

For information on how to create a subtype-like relationship between two generic classes when the type parameters are related, see the section Wildcards and Subtyping.

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.

Using the Collections classes as an example, 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.



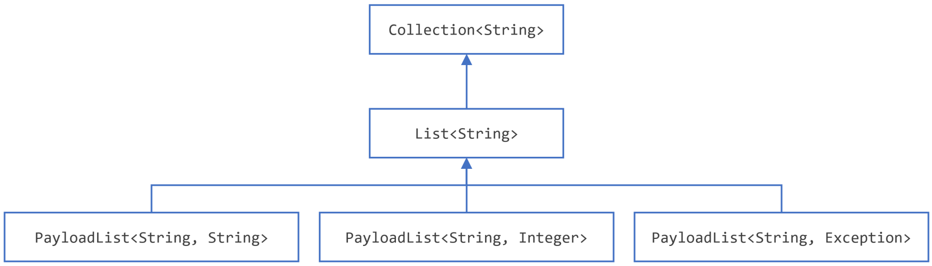
A sample Collection hierarchy.

Now imagine we want to define our own list interface, PayloadList, that associates an optional value of generic type P with each element. Its declaration might look like:

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

The following parameterizations of PayloadList are subtypes of List<String>:

- PayloadList<String, String>
- PayloadList<String, Integer>
- PayloadList<String, Exception>



A sample Payload hierarchy.

2.2 Type Inference

2.2.1 Type Inference and Generic Methods

Type inference is a Java compiler's ability to look at each method invocation and corresponding declaration to determine the type argument (or arguments) that make the invocation applicable. The inference algorithm determines the types of the arguments and, if available, the type that the result is being assigned, or returned. Finally, the inference algorithm tries to find the most specific type that works with all of the arguments.

To illustrate this last point, in the following example, inference determines that the second argument being passed to the pick method is of type Serializable:

```java
public static <T> T pick(T a1, T a2) {
  return a2;
}

Serializable s = pick("d", new ArrayList<String>());
```

Generic Methods introduced you to type inference, which enables you to invoke a generic method as you would an ordinary method, without specifying a type between angle brackets. Consider the following example, BoxDemo, which requires the Box class:

```java
public class BoxDemo {
  public static <U> void addBox(U u, java.util.List<Box<U>> boxes) {
    Box<U> box = new Box<>();
```

```java
        box.set(u);
        boxes.add(box);
    }

    public static <U> void outputBoxes(java.util.List<Box<U>> boxes) {
        int counter = 0;
        for (Box<U> box : boxes) {
            U boxContents = box.get();
            System.out.println("Box #" + counter + " contains [" +
                boxContents.toString() + "]");
            counter++;
        }
    }

    public static void main(String[] args) {
        java.util.ArrayList<Box<Integer>> listOfIntegerBoxes =
            new java.util.ArrayList<>();
        BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes);
        BoxDemo.addBox(Integer.valueOf(30), listOfIntegerBoxes);
        BoxDemo.outputBoxes(listOfIntegerBoxes);
    }
}
```

The following is the output from this example:

Command window

```
Box #0 contains [10]
Box #1 contains [20]
Box #2 contains [30]
```

The generic method addBox() defines one type parameter named U. Generally, a Java compiler can infer the type parameters of a generic method call. Consequently, in most cases, you do not have to specify them. For example, to invoke the generic method addBox(), you can specify the type parameter with a type witness as follows:

```java
BoxDemo.<Integer>addBox(Integer.valueOf(10), listOfIntegerBoxes);
```

Alternatively, if you omit the type witness,a Java compiler automatically infers (from the method's arguments) that the type parameter is Integer:

```java
BoxDemo.addBox(Integer.valueOf(20), listOfIntegerBoxes );
```

2.2.2 Type Inference and Instantiation of Generic Classes

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (<>) as long as the compiler can infer the type arguments from the context. This pair of angle brackets is informally called the diamond.

For example, consider the following variable declaration:

```java
Map<String, List<String>> myMap = new HashMap<String, List<String>>();
```

You can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```java
Map<String, List<String>> myMap = new HashMap<>();
```

Note that to take advantage of type inference during generic class instantiation, you must use the diamond. In the following example, the compiler generates an unchecked conversion warning because the HashMap() constructor refers to the HashMap raw type, not the Map<String, List<String>> type:

```java
Map<String, List<String>> myMap = new HashMap(); // unchecked conversion warning
```

2.2.3 Type Inference and Generic Constructors of Generic and Non-Generic Classes

Note that constructors can be generic (in other words, declare their own formal type parameters) in both generic and non-generic classes. Consider the following example:

```java
public class MyClass<X> {
  <T> MyClass(T t) {
    // ...
  }
}
```

Consider the following instantiation of the class MyClass:

```java
new MyClass<Integer>("")
```

This statement creates an instance of the parameterized type MyClass<Integer>; the statement explicitly specifies the type Integer for the formal type parameter, X, of the generic class MyClass<X>. Note that the constructor for this generic class contains a formal type parameter, T. The compiler infers the type String for the formal type parameter, T, of the constructor of this generic class (because the actual parameter of this constructor is a String object).

Compilers from releases prior to Java SE 7 are able to infer the actual type parameters of generic constructors, similar to generic methods. However, compilers in Java SE 7 and later can infer the actual type parameters of the generic class being instantiated if you use the diamond (<>). Consider the following example:

```java
MyClass<Integer> myObject = new MyClass<>("");
```

In this example, the compiler infers the type Integer for the formal type parameter, X, of the generic class MyClass<X>. It infers the type String for the formal type parameter, T, of the constructor of this generic class.

> Note: It is important to note that the inference algorithm uses only invocation arguments, target types, and possibly an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

2.2.4 Target Types

The Java compiler takes advantage of target typing to infer the type parameters of a generic method invocation. The target type of an expression is the data type that the Java compiler expects depending on where the expression appears. Consider the method Collections.*emptyList*(), which is declared as follows:

```java
static <T> List<T> emptyList();
```

Consider the following assignment statement:

```java
List<String> listOne = Collections.emptyList();
```

This statement is expecting an instance of List<String> this data type is the target type. Because the method emptyList() returns a value of type List<T>, the compiler infers that the type argument T must be the value String. This works in both Java SE 7 and 8. Alternatively, you could use a type witness and specify the value of T as follows:

```java
List<String> listOne = Collections.<String>emptyList();
```

However, this is not necessary in this context. It was necessary in other contexts, though. Consider the following method:

```java
void processStringList(List<String> stringList) {
    // process stringList
}
```

Suppose you want to invoke the method processStringList() with an empty list. In Java SE 7, the following statement does not compile:

```java
processStringList(Collections.emptyList());
```

The Java SE 7 compiler generates an error message similar to the following:

```
List<Object> cannot be converted to List<String>
```

The compiler requires a value for the type argument T so it starts with the value Object. Consequently, the invocation of Collections.emptyList() returns a value of type List<Object>, which is incompatible with the method processStringList(). Thus, in Java SE 7, you must specify the value of the type argument as follows:

```java
processStringList(Collections.<String>emptyList());
```

This is no longer necessary in Java SE 8. The notion of what is a target type has been expanded to include method arguments, such as the argument to the method processStringList(). In this case, processStringList() requires an argument of type List<String>. The method Collections.emptyList() returns a value of List<T>, so using the target type of List<String>, the compiler infers that the type argument T has a value of String. Thus, in Java SE 8, the following statement compiles:

```java
processStringList(Collections.emptyList());
```

2.2.5 Target Typing in Lambda Expressions

Suppose you have the following methods:

```java
static void printPersons(List<Person> roster, CheckPerson tester)
```

and

```java
void printPersonsWithPredicate(List<Person> roster, Predicate<Person> tester)
```

You then write the following code to call these methods:

```java
printPersons(
  people,
  p -> p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25);
```

and

```java
printPersonsWithPredicate(
  people,
  p -> p.getGender() == Person.Sex.MALE
      && p.getAge() >= 18
      && p.getAge() <= 25);)
```

How do you determine the type of the lambda expression in these cases?

When the Java runtime invokes the method printPersons(), it is expecting a data type of CheckPerson, so the lambda expression is of this type. However, when the Java runtime invokes the method printPersonsWithPredicate(), it is expecting a data type of Predicate<Person>, so the lambda expression is of this type. The data type that these methods expect is called the target type. To determine the type of a lambda expression, the Java compiler uses the target type of the context or situation in which the lambda expression was found. It follows that you can only use lambda expressions in situations in which the Java compiler can determine a target type:

- Variable declarations
- Assignments
- Return statements

- Array initializers
- Method or constructor arguments
- Lambda expression bodies
- Conditional expressions, ?:
- Cast expressions

2.2.6 Target Types and Method Arguments

For method arguments, the Java compiler determines the target type with two other language features: overload resolution and type argument inference.

Consider the following two functional interfaces (java.lang.Runnable and java.util.concurrent.Callable<V>:

```java
public interface Runnable {
    void run();
}
```

```java
public interface Callable<V> {
    V call();
}
```

The method Runnable.run() does not return a value, whereas Callable<V>.call() does.

Suppose that you have overloaded the method invoke as follows:

```java
void invoke(Runnable r) {
    r.run();
}

<T> T invoke(Callable<T> c) {
    return c.call();
}
```

Which method will be invoked in the following statement?

```java
String s = invoke(() -> "done");
```

The method invoke(Callable<T>) will be invoked because that method returns a value; the method invoke(Runnable) does not. In this case, the type of the lambda expression () -> "done" is Callable<T>.

2.3 Wildcards

2.3.1 Upper Bounded Wildcards

You can use an upper bounded wildcard to relax the restrictions on a variable. For example, say you want to write a method that works on List<Integer>, List<Double>, and List<Number>; you can achieve this by using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound. Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

To write the method that works on lists of Number and the subtypes of Number, such as Integer, Double, and Float, you would specify List<? extends Number>. The term List<Number> is more restrictive than List<? extends Number> because the former matches a list of type Number only, whereas the latter matches a list of type Number or any of its subclasses.

Consider the following process method:

```java
public static void process(List<? extends Foo> list) {
  /* ... */
}
```

The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo. The process method can access the list elements as type Foo:

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

In the foreach clause, the elem variable iterates over each element in the list. Any method defined in the Foo class can now be used on elem.

The sumOfList() method returns the sum of the numbers in a list:

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

The following code, using a list of Integer objects, prints sum = 6.0:

```java
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println("sum = " + sumOfList(li));
```

A list of Double values can use the same sumOfList() method. The following code prints sum = 7.0:

```java
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println("sum = " + sumOfList(ld));
```

2.3.2 Unbounded Wildcards

The unbounded wildcard type is specified using the wildcard character (?), for example, List<?>. This is called a list of unknown type. There are two scenarios where an unbounded wildcard is a useful approach:

- If you are writing a method that can be implemented using functionality provided in the Object class.
- When the code is using methods in the generic class that do not depend on the type parameter. For example, List.size() or List.clear(). In fact, Class<?> is so often used because most of the methods in Class<T> do not depend on T.

Consider the following method, printList():

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

The goal of printList() is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>. To write a generic printList() method, use List<?>:

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

Because for any concrete type A, List<A> is a subtype of List<?>, you can use printList() to print a list of any type:

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

printList(li);
printList(ls);
```

> **Note**: The Arrays.asList() method is used in examples throughout this section. This static factory method converts the specified array and returns a fixed-size list.

It's important to note that List<Object> and List<?> are not the same. You can insert an Object, or any subtype of Object, into a List<Object>. But you can only insert null into a List<?>.

2.3.3 Lower Bounded Wildcards

The Upper Bounded Wildcards section shows that 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. In a similar way, a lower bounded wildcard restricts the unknown type to be a specific type or a super type of that type.

A lower bounded wildcard is expressed using the wildcard character ('?'), following by the super keyword, followed by its lower bound: <? super A>.

> **Note**: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

Say you want to write a method that puts Integer objects into a list. To maximize flexibility, you would like the method to work on List<Integer>, List<Number>, and List<Object> - anything that can hold Integer values.

To write the method that works on lists of Integer and the supertypes 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> because the former matches a list of type Integer only, whereas the latter matches a list of any type that is a supertype of Integer.

The following code adds the numbers 1 through 10 to the end of a list:

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

2.3.4 Wildcards and Subtyping

As described in previous sections, generic classes or interfaces are not related merely because there is a relationship between their types. However, you can use wildcards to create a relationship between generic classes or interfaces.

Given the following two regular (non-generic) classes:

```java
class A {
    /* ... */
}
```

```java
class B extends A {
    /* ... */
}
```

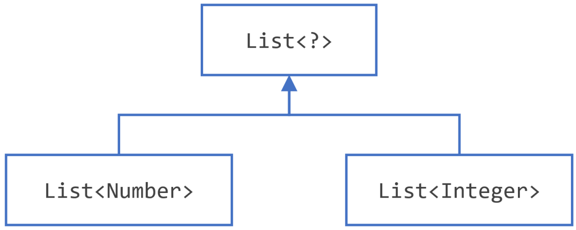
It would be reasonable to write the following code:

```java
B b = new B();
A a = b;
```

This example shows that inheritance of regular classes follows this rule of subtyping: class B is a subtype of class A if B extends A. This rule does not apply to generic types:

```java
List<B> lb = new ArrayList<>();
List<A> la = lb;    // Compile-time error
```

Given that Integer is a subtype of Number, what is the relationship between List<Integer> and List<Number>?

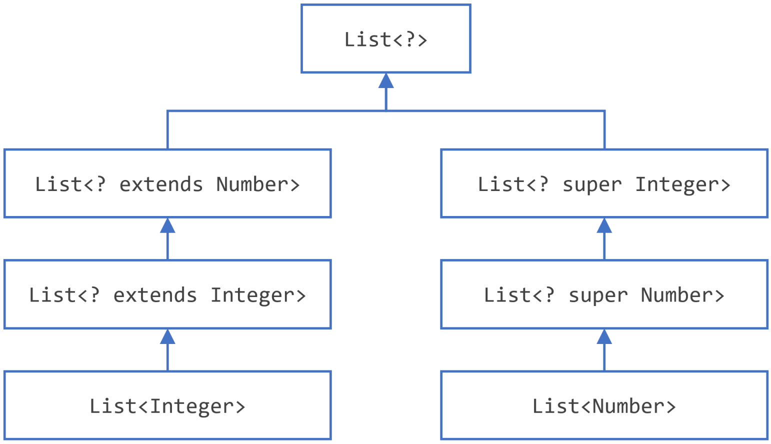


Although Integer is a subtype of Number, List<Integer> is not a subtype of List<Number> and, in fact, these two types are not related. The common parent of List<Number> and List<Integer> is List<?>.

In order to create a relationship between these classes so that the code can access Number's methods through List<Integer>'s elements, use an upper bounded wildcard:

```java
List<? extends Integer> intList = new ArrayList<>();
List<? extends Number> numList = intList;
// Ok. List<? extends Integer> is a subtype of List<? extends Number>
```

Because Integer is a subtype of Number, and numList is a list of Number objects, a relationship now exists between intList (a list of Integer objects) and numList. The following diagram shows the relationships between several List classes declared with both upper and lower bounded wildcards.



A hierarchy of several generic List class declarations.

2.3.5 Wildcard Capture and Helper Methods

In some cases, the compiler infers the type of a wildcard. For example, a list may be defined as List<?> but, when evaluating an expression, the compiler infers a particular type from the code. This scenario is known as wildcard capture.

For the most part, you do not need to worry about wildcard capture, except when you see an error message that contains the phrase "capture of".

The WildcardError example produces a capture error when compiled:

```java
public class WildcardError {
  void foo(List<?> i) {
    i.set(0, i.get(0));
  }
}
```

In this example, the compiler processes the i input parameter as being of type Object. When the foo method invokes List.set(int, E), the compiler is not able to confirm the type of object that is being inserted into the list, and an error is produced. When this type of error occurs it typically means that the compiler believes that you are assigning the wrong type to a variable. Generics were added to the Java language for this reason — to enforce type safety at compile time.

The WildcardError example generates the following error when compiled by Oracle's JDK 7 javac implementation:

Command window

```
WildcardError.java:6: error: method set in interface List<E> cannot be applied to
     given types;
    i.set(0, i.get(0));
    ^
  required: int,CAP#1
  found: int,Object
  reason: actual argument Object cannot be converted to CAP#1 by method
       invocation conversion where E is a type-variable:
    E extends Object declared in interface List
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Object from capture of ?
1 error
```

In this example, the code is attempting to perform a safe operation, so how can you work around the compiler error? You can fix it by writing a private helper method which captures the wildcard. In this case, you can work around the problem by creating the private helper method, fooHelper(), as shown in WildcardFixed:

```java
public class WildcardFixed {
  void foo(List<?> i) {
    fooHelper(i);
  }

  // Helper method created so that the wildcard can be captured
  // through type inference.
  private <T> void fooHelper(List<T> l) {
    l.set(0, l.get(0));
  }
}
```

Thanks to the helper method, the compiler uses inference to determine that T is CAP#1, the capture variable, in the invocation. The example now compiles successfully.

By convention, helper methods are generally named originalMethodNameHelper().

Now consider a more complex example, WildcardErrorBad:

```java
public class WildcardErrorBad {
  void swapFirst(List<? extends Number> l1, List<? extends Number> l2) {
    Number temp = l1.get(0);
    l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
    // got a CAP#2 extends Number; same bound, but different types
    l2.set(0, temp);      // expected a CAP#1 extends Number,
    // got a Number
  }
}
```

In this example, the code is attempting an unsafe operation. For example, consider the following invocation of the swapFirst() method:

```java
List<Integer> li = Arrays.asList(1, 2, 3);
List<Double> ld = Arrays.asList(10.10, 20.20, 30.30);
swapFirst(li, ld);
```

While List<Integer> and List<Double> both fulfill the criteria of List<? extends Number>, it is clearly incorrect to take an item from a list of Integer values and attempt to place it into a list of Double values.

Compiling the code with Oracle's JDK javac compiler produces the following error:

Command window

```
WildcardErrorBad.java:7: error: method set in interface List<E> cannot be applied to given types;
l1.set(0, l2.get(0)); // expected a CAP#1 extends Number,
^
required: int,CAP#1
found: int,Number
reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
where E is a type-variable:
E extends Object declared in interface List
where CAP#1 is a fresh type-variable:
CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:10: error: method set in interface List<E> cannot be applied to given types;
l2.set(0, temp);      // expected a CAP#1 extends Number,
^
required: int,CAP#1
found: int,Number
reason: actual argument Number cannot be converted to CAP#1 by method invocation conversion
where E is a type-variable:
E extends Object declared in interface List
where CAP#1 is a fresh type-variable:
CAP#1 extends Number from capture of ? extends Number
WildcardErrorBad.java:15: error: method set in interface List<E> cannot be applied to given types;
i.set(0, i.get(0));
^
required: int,CAP#1
found: int,Object
reason: actual argument Object cannot be converted to CAP#1 by method invocation conversion
where E is a type-variable:
E extends Object declared in interface List
where CAP#1 is a fresh type-variable:
CAP#1 extends Object from capture of ?
3 errors
```

There is no helper method to work around the problem, because the code is fundamentally wrong: it is clearly incorrect to take an item from a list of Integer values and attempt to place it into a list of Double values.

2.3.6 Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard. This page provides some guidelines to follow when designing your code.

For purposes of this discussion, it is helpful to think of variables as providing one of two functions:

- An "In" Variable. An "in" variable serves up data to the code. Imagine a copy method with two arguments: copy(src, dest). The src argument provides the data to be copied, so it is the "in" parameter. An "Out" Variable. An "out" variable holds data for use elsewhere. In the copy example, copy(src, dest), the dest argument accepts data, so it is the "out" parameter.

Of course, some variables are used both for "in" and "out" purposes — this scenario is also addressed in the guidelines.

You can use the "in" and "out" principle when deciding whether to use a wildcard and what type of wildcard is appropriate. The following list provides the guidelines to follow:

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.

These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.

A list defined by List<? extends ...> can be informally thought of as read-only, but that is not a strict guarantee. Suppose you have the following two classes:

```java
class NaturalNumber {
  private int i;

  public NaturalNumber(int i) {
    this.i = i;
```

```java
    }

    // ...
}


class EvenNumber extends NaturalNumber {
    public EvenNumber(int i) {
        super(i);
    }

    // ...
}
```

Consider the following code:

```java
List<EvenNumber> le = new ArrayList<>();
List<? extends NaturalNumber> ln = le;
ln.add(new NaturalNumber(35));  // compile-time error
```

Because List<EvenNumber> is a subtype of List<? extends NaturalNumber>, you can assign le to ln. But you cannot use ln to add a natural number to a list of even numbers. The following operations on the list are possible:

- You can add null.
- You can invoke clear().
- You can get the iterator and invoke remove().
- You can capture the wildcard and write elements that you have read from the list.

You can see that the list defined by List<? extends NaturalNumber> is not read-only in the strictest sense of the word, but you might think of it that way because you cannot store a new element or change an existing element in the list.

2.4 Type Erasure

2.4.1 Erasure of Generic Types

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:

- Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```java
public class Node<T> {
  private T data;
  private Node<T> next;

  public Node(T data, Node<T> next) {
    this.data = data;
    this.next = next;
  }

  public T getData() {
    return data;
  }
  // ...
}
```

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```java
public class Node {
  private Object data;
  private Node next;

  public Node(Object data, Node next) {
    this.data = data;
    this.next = next;
  }

  public Object getData() {
```

```java
        return data;
    }
    // ...
}
```

In the following example, the generic Node class uses a bounded type parameter:

```java
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }

    public T getData() {
        return data;
    }
    // ...
}
```

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

```java
public class Node {
    private Comparable data;
    private Node next;

    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }

    public Comparable getData() {
        return data;
    }
    // ...
}
```

2.4.2 Erasure of Generic Methods

The Java compiler also erases type parameters in generic method arguments. Consider the following generic method:

```java
// Counts the number of occurrences of elem in anArray.
public static <T> int count(T[] anArray, T elem) {
    int cnt = 0;
    for (T e : anArray) {
        if (e.equals(elem)) {
            ++cnt;
        }
    }

    return cnt;
}
```

Because T is unbounded, the Java compiler replaces it with Object:

```java
public static int count(Object[] anArray, Object elem) {
    int cnt = 0;
    for (Object e : anArray) {
        if (e.equals(elem)) {
            ++cnt;
        }
    }

    return cnt;
}
```

Suppose the following classes are defined:

```java
class Shape { /* ... */ }
class Circle extends Shape { /* ... */ }
class Rectangle extends Shape { /* ... */ }
```

You can write a generic method to draw different shapes:

```java
public static <T extends Shape> void draw(T shape) { /* ... */ }
```

The Java compiler replaces T with Shape:

```java
public static void draw(Shape shape) { /* ... */ }
```

2.4.3 Effects of Type Erasure and Bridge Methods

Sometimes type erasure causes a situation that you may not have anticipated. The following example shows how this can occur. The following example shows how a compiler sometimes creates a synthetic method, which is called a bridge method, as part of the type erasure process.

Given the following two classes:

```java
public class Node<T> {
  public T data;

  public Node(T data) {
    this.data = data;
  }

  public void setData(T data) {
    System.out.println("Node.setData");
    this.data = data;
  }
}
```

```java
public class MyNode extends Node<Integer> {
  public MyNode(Integer data) {
    super(data);
  }

  public void setData(Integer data) {
    System.out.println("MyNode.setData");
```

```java
        super.setData(data);
    }
}
```

Consider the following code:

```java
MyNode mn = new MyNode(5);
Node n = mn;            // A raw type - compiler throws an unchecked warning
n.setData("Hello");        // Causes a ClassCastException to be thrown.
Integer x = mn.data;
```

After type erasure, this code becomes:

```java
MyNode mn = new MyNode(5);
Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning
n.setData("Hello");              // Causes a ClassCastException to be thrown.
Integer x = (String)mn.data;
```

The next section explains why a ClassCastException is thrown at the n.setData("Hello"); statement.

2.4.4 Bridge Methods

When compiling a class or interface that extends a parameterized class or implements a parameterized interface, the compiler may need to create a synthetic method, which is called a bridge method, as part of the type erasure process. You normally do not need to worry about bridge methods, but you might be puzzled if one appears in a stack trace.

After type erasure, the Node and MyNode classes become:

```java
public class Node {
  public Object data;

  public Node(Object data) {
```

```
        this.data = data;
    }

    public void setData(Object data) {
        System.out.println("Node.setData");
        this.data = data;
    }
}
```

```
public class MyNode extends Node {
    public MyNode(Integer data) {
        super(data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
}
```

After type erasure, the method signatures do not match; the Node.setData(T) method becomes Node.setData(Object). As a result, the MyNode.setData(Integer) method does not override the Node.setData(Object) method.

To solve this problem and preserve the polymorphism of generic types after type erasure, the Java compiler generates a bridge method to ensure that subtyping works as expected.

For the MyNode class, the compiler generates the following bridge method for setData():

```
class MyNode extends Node {
    // Bridge method generated by the compiler
    public void setData(Object data) {
        setData((Integer) data);
    }

    public void setData(Integer data) {
        System.out.println("MyNode.setData");
        super.setData(data);
    }
```

```
    // ...
}
```

The bridge method MyNode.setData(object) delegates to the original MyNode.setData(Integer) method. As a result, the n.setData("Hello"); statement calls the method MyNode.setData(Object), and a ClassCastException is thrown because "Hello" cannot be cast to Integer.

2.4.5 Non-Reifiable Types

We discussed the process where the compiler removes information related to type parameters and type arguments. Type erasure has consequences related to variable arguments (also known as varargs) methods whose varargs formal parameter has a non-reifiable type. See the section Arbitrary Number of Arguments in Passing Information to a Method or a Constructor for more information about varargs methods.

This page covers the following topics:

- Non-Reifiable Types
- Heap Pollution
- Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters
- Preventing Warnings from Varargs Methods with Non-Reifiable Formal Parameters

A reifiable type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String> and List<Number>; the JVM cannot tell the difference between these types at runtime. As shown in the Restrictions on Generics section, there are certain situations where non-reifiable types cannot be used: in an instanceof expression, for example, or as an element in an array.

2.4.6 Heap Pollution

Heap pollution occurs when a variable of a parameterized type refers to an object that is not of that parameterized type. This situation occurs if the program performed some operation that gives rise to an unchecked warning at compile-time. An unchecked warning is generated if, either at compile-time (within the limits of the compile-time type checking rules) or at runtime, the correctness of an operation involving a parameterized type (for example, a cast or method call) cannot be verified. For example, heap pollution occurs when mixing raw types and parameterized types, or when performing unchecked casts.

In normal situations, when all code is compiled at the same time, the compiler issues an unchecked warning to draw your attention to potential heap pollution. If you compile sections of your code separately, it is difficult to detect the potential risk of heap pollution. If you ensure that your code compiles without warnings, then no heap pollution can occur.

2.4.7 Potential Vulnerabilities of Varargs Methods with Non-Reifiable Formal Parameters

Generic methods that include vararg input parameters can cause heap pollution.

Consider the following ArrayBuilder class:

```
public class ArrayBuilder {
  public static <T> void addToList (List<T> listArg, T... elements) {
    for (T x : elements) {
      listArg.add(x);
    }
  }

  public static void faultyMethod(List<String>... l) {
    Object[] objectArray = l;     // Valid
    objectArray[0] = Arrays.asList(42);
    String s = l[0].get(0);       // ClassCastException thrown here
  }
}
```

The following example, HeapPollutionExample uses the ArrayBuiler class:

```
public class HeapPollutionExample {
  public static void main(String[] args) {
```

```
List<String> stringListA = new ArrayList<String>();
List<String> stringListB = new ArrayList<String>();

ArrayBuilder.addToList(stringListA, "Seven", "Eight", "Nine");
ArrayBuilder.addToList(stringListB, "Ten", "Eleven", "Twelve");
List<List<String>> listOfStringLists = new ArrayList<List<String>>();
ArrayBuilder.addToList(listOfStringLists,
stringListA, stringListB);

ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
}
}
```

When compiled, the following warning is produced by the definition of the ArrayBuilder .*addToList*() method:

Command window

```
warning: [varargs] Possible heap pollution from parameterized vararg type T
```

When the compiler encounters a varargs method, it translates the varargs formal parameter into an array. However, the Java programming language does not permit the creation of arrays of parameterized types. In the method ArrayBuilder.addToList(), the compiler translates the varargs formal parameter T... elements to the formal parameter T[] elements, an array. However, because of type erasure, the compiler converts the varargs formal parameter to Object[] elements. Consequently, there is a possibility of heap pollution.

The following statement assigns the varargs formal parameter l to the Object array objectArgs:

```
Object[] objectArray = l;
```

This statement can potentially introduce heap pollution. A value that does match the parameterized type of the varargs formal parameter l can be assigned to the variable objectArray, and thus can be assigned to l. However, the compiler does not generate an unchecked warning at this statement. The compiler has already generated a warning when it translated the varargs formal parameter List<String>... l to the formal parameter List[] l. This statement is valid; the variable l has the type List[], which is a subtype of Object[].

Consequently, the compiler does not issue a warning or error if you assign a List object of any type to any array component of the objectArray array as shown by this statement:

```
objectArray[0] = Arrays.asList(42);
```

This statement assigns to the first array component of the objectArray array with a List object that contains one object of type Integer.

Suppose you invoke ArrayBuilder.$faultyMethod()$ with the following statement:

```
ArrayBuilder.faultyMethod(Arrays.asList("Hello!"), Arrays.asList("World!"));
```

At runtime, the JVM throws a ClassCastException at the following statement:

```
// ClassCastException thrown here
String s = l[0].get(0);
```

The object stored in the first array component of the variable l has the type List<Integer>, but this statement is expecting an object of type List<String>.

2.4.8 Prevent Warnings from Varargs Methods with Non-Reifiable Formal Parameters

If you declare a varargs method that has parameters of a parameterized type, and you ensure that the body of the method does not throw a ClassCastException or other similar exception due to improper handling of the varargs formal parameter, you can prevent the warning that the compiler generates for these kinds of varargs methods by adding the following annotation to static and non-constructor method declarations:

Command window

```
@SafeVarargs
```

The @SafeVarargs annotation is a documented part of the method's contract; this annotation asserts that the implementation of the method will not improperly handle the varargs formal parameter.

It is also possible, though less desirable, to suppress such warnings by adding the following to the method declaration:

Command window

```
@SuppressWarnings({"unchecked", "varargs "})
```

However, this approach does not suppress warnings generated from the method's call site. If you are unfamiliar with the @SuppressWarnings syntax, see the section Annotations.

2.5 Restriction on Generics

2.5.1 Cannot Instantiate Generic Types with Primitive Types

Consider the following parameterized type:

```java
class Pair<K, V> {
  private K key;
  private V value;

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

  // ...
}
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

```java
Pair<int, char> p = new Pair<>(8, 'a');   // compile-time error
```

You can substitute only non-primitive types for the type parameters K and V:

```java
Pair<Integer, Character> p = new Pair<>(8, 'a');
```

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

```java
Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));
```

For more information on autoboxing, see Autoboxing and Unboxing in the Numbers and Strings section.

2.5.2 Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```java
public static <E> void append(List<E> list) {
  E elem = new E();   // compile-time error
   list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```java
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
  E elem = cls.newInstance();    // Ok
   list.add(elem);
}
```

You can invoke the append() method as follows:

```java
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

2.5.3 Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```java
public class MobileDevice<T> {
    private static T os;

    // ...
}
```

If static fields of type parameters were allowed, then the following code would be confused:

```java
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

2.5.4 Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```java
public static <E> void rtti(List<E> list) {
    if (list instanceof ArrayList<Integer>) {  // Compile-time error
        // ...
    }
}
```

The set of parameterized types passed to the rtti() method is:

```java
S = {ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ...}
```

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```java
public static void rtti(List<?> list) {
    if (list instanceof ArrayList<?>) {  // Ok; instanceof requires a reifiable type
        // ...
    }
}
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```java
List<Integer> li = new ArrayList<>();
List<Number> ln = (List<Number>) li;  // Compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```java
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // Ok
```

2.5.5 Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```java
List<Integer>[] arrayOfLists = new List<Integer>[2]; // Compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```java
Object[] strings = new String[2];
strings[0] = "hi"; // Ok
strings[1] = 100; // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```java
Object[] stringLists = new List<String>[2]; // Compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>(); // Ok
stringLists[1] = new ArrayList<Integer>(); // An ArrayStoreException should be thrown, but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

2.5.6 Cannot Create, Catch, or Throw Objects of Parameterized Types

A generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```java
// Extends Throwable indirectly
class MathException<T> extends Exception { // Compile-time error
```

```
    /* ... */
}

// Extends Throwable directly
class QueueFullException<T> extends Throwable {  // Compile-time error
    /* ... */
}
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
  try {
    for (J job : jobs)
      // ...
  } catch (T e) {   // compile-time error
      // ...
  }
}
```

You can, however, use a type parameter in a throws clause:

```
public class Parser<T extends Exception> {
  public void parse(File file) throws T {  // Ok
      // ...
  }
}
```

2.5.7 Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

A class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {
  public void print(Set<String> strSet) {
```

```
        /* ... */
    }

    public void print(Set<Integer> intSet) {
        /* ... */
    }
}
```

The overloads would all share the same classfile representation and will generate a compile-time error.

Part VII Java Functional Programming

Part VIII Java Concurrency

Part IX Java GUI

Part X References