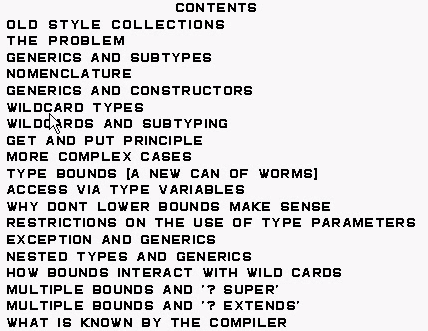
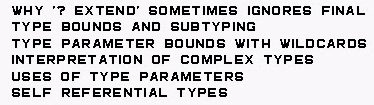
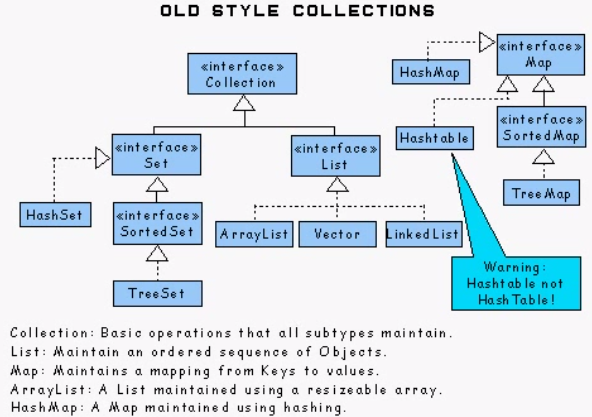
Intro to Generic Types (From Clive Scott Videos)

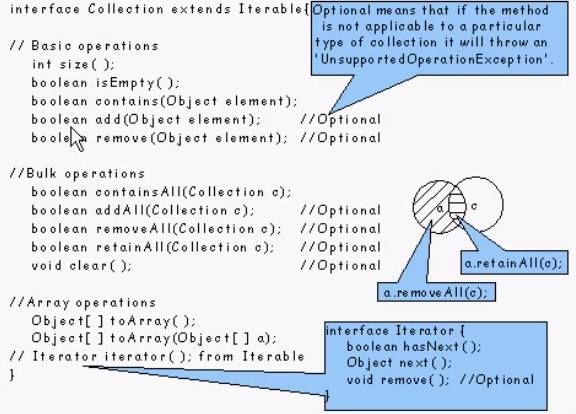


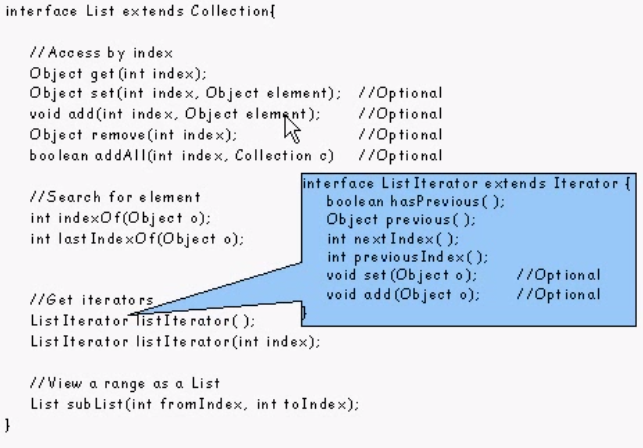


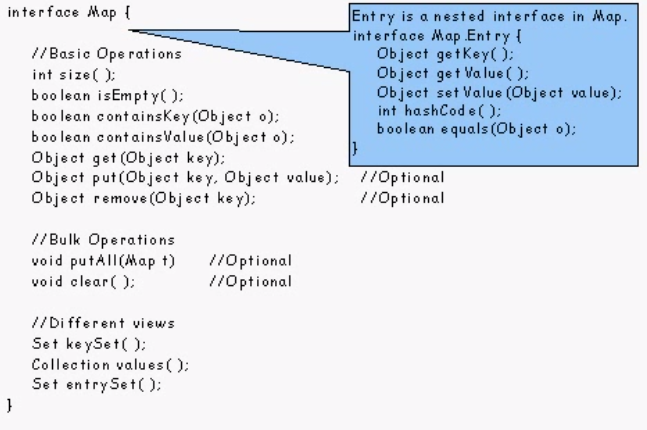
Vector is thread-safe unlike ArrayList

Hashtable is thread-safe unlike HashMap









+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Generic Methods (From Java Doc)

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection. Here's a first attempt:

**static void** fromArrayToCollection(Object[] a, Collection<?> c) {

**for** (Object o : a) {

c.add(o); // *compile-time error*

}

}

By now, you will have learned to avoid the beginner's mistake of trying to use Collection<Object> as the type of the collection parameter. You may or may not have recognized that using Collection<?> isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use *generic methods*. Just like type declarations, method declarations can be generic—that is, parameterized by one or more type parameters.

**static** <T> **void** fromArrayToCollection(T[] a, Collection<T> c) {

**for** (T o : a) {

c.add(o); // *Correct*

}

}

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

Object[] oa = new Object[100];

Collection<Object> co = new ArrayList<Object>();

// *T inferred to be Object*

fromArrayToCollection(oa, co);

String[] sa = new String[100];

Collection<String> cs = new ArrayList<String>();

// *T inferred to be String*

fromArrayToCollection(sa, cs);

// *T inferred to be Object*

fromArrayToCollection(sa, co);

Integer[] ia = new Integer[100];

Float[] fa = new Float[100];

Number[] na = new Number[100];

Collection<Number> cn = new ArrayList<Number>();

// *T inferred to be Number*

fromArrayToCollection(ia, cn);

// *T inferred to be Number*

fromArrayToCollection(fa, cn);

// *T inferred to be Number*

fromArrayToCollection(na, cn);

// *T inferred to be Object*

fromArrayToCollection(na, co);

// *compile-time error*

fromArrayToCollection(na, cs);

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from the Collection libraries.

**interface** Collection<E> {

**public boolean** containsAll(Collection<?> c);

**public boolean** addAll(Collection<? **extends E**> c);

}

We could have used generic methods here instead:

**interface** Collection<E> {

**public** <T> **boolean** containsAll(Collection<T> c);

**public** <T **extends** E> **boolean** addAll(Collection<T> c);

// *Hey, type variables can have bounds too!*

}

However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy():

**class** Collections {

**public static** <T> **void** copy(List<T> dest, List<? **extends** T> src) {

...

}

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T—we don't care which. The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

**class** Collections {

**public static** <T, S **extends** T> **void** copy(List<T> dest, List<S> src) {

...

}

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src—nothing else depends on it. This is a sign that we can replace S with a wildcard. Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside classShape, and have drawAll() store its incoming argument into the history field.

**static** List<List<? extends Shape>>

history = new ArrayList<List<? extends Shape>>();

**public void** drawAll(List<? **extends** Shape> shapes) {

history.addLast(shapes);

**for** (Shape s: shapes) {

s.draw(**this**);

}

}

Finally, again let's take note of the naming convention used for the type parameters. We use T for type, whenever there isn't anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor T in the alphabet, such as S. If a generic method appears inside a generic class, it's a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Type Inference (From Java Doc)

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:

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

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

## Type Inference and Generic Methods

[Generic Methods](file:///C:\project\books\javatutorials\tutorial\java\generics\methods.html) 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](file:///C:\project\books\javatutorials\tutorial\java\generics\examples\BoxDemo.java), which requires the [Box](file:///C:\project\books\javatutorials\tutorial\java\generics\examples\Box.java) class:

public class BoxDemo {

public static <U> void addBox(U u,

java.util.List<Box<U>> boxes) {

Box<U> box = new Box<>();

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:

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 as follows:

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

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

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

## 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](file:///C:\project\books\javatutorials\tutorial\java\generics\types.html#diamond).

For example, consider the following variable declaration:

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

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

Map<String, List<String>> my Map = new HashMap<>();

Note that to take advantage of type inference during generic class instantiation, you must diamond notation. 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:

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

Java supports limited type inference for generic instance creation; you can only use type inference if the parameterized type of the constructor is obvious from the context. For example, the following code does not compile:

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

list.add("A");

// The following statement should fail since addAll expects

// Collection<? extends String>

list.addAll(new ArrayList<>());

Note that the diamond often works in method calls; however, for greater clarity, it is suggested that you use the diamond primarily to initialize a variable where it is declared.

In comparison, the following example compiles:

List<? extends String> list2 = new ArrayList<>();

list.addAll(list2);

## 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:

class MyClass<X> {

<T> MyClass(T t) {

// ...

}

}

Consider the following instantiation of the class MyClass:

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 typeString 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:

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 and, possibly, an obvious expected return type to infer types. The inference algorithm does not use results from later in the program.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Erasure, Reifiable Types & BridgesEnum (From Java Doc)

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.

# Erasure of Generic Types

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:

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:

public class Node {

private Object data;

private Node next;

public Node(Object data, Node next) {

this.data = data;

this.next = next;

}

public Object getData() { return data; }

// ...

}

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

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:

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; }

// ...

}

**Erasure of Generic Methods**

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

// 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:

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:

class Shape { /\* ... \*/ }

class Circle extends Shape { /\* ... \*/ }

class Rectangle extends Shape { /\* ... \*/ }

You can write a generic method to draw different shapes:

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

The Java compiler replaces T with Shape:

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

**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 example (described in [Bridge Methods](file:///C:\project\books\javatutorials\tutorial\java\generics\bridgeMethods.html#bridgeMethods)) shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Given the following two classes:

public class Node<T> {

private T data;

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

public void setData(T data) {

System.out.println("Node.setData");

this.data = data;

}

}

public class MyNode extends Node<Integer> {

public MyNode(Integer data) { super(data); }

public void setData(Integer data) {

System.out.println("MyNode.setData");

super.setData(data);

}

}

Consider the following code:

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:

MyNode mn = new MyNode(5);

Node n = (MyNode)mn; // A raw type - compiler throws an unchecked warning

n.setData("Hello");

Integer x = (String)mn.data; // Causes a ClassCastException to be thrown.

Here is what happens as the code is executed:

* n.setData("Hello"); causes the method setData(Object) to be executed on the object of class MyNode. (The MyNode class inheritedsetData(Object) from Node.)
* In the body of setData(Object), the data field of the object referenced by n is assigned to a String.
* The data field of that same object, referenced via mn, can be accessed and is expected to be an integer (since mn is a MyNode which is a Node<Integer>.
* Trying to assign a String to an Integer causes a ClassCastException from a cast inserted at the assignment by a Java compiler.

## 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, called a bridge method, as part of the type erasure process. You normally don't 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:

public class Node {

private 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(Integer data);

super.setData(data);

}

}

After type erasure, the method signatures do not not match. The Node method becomes setData(Object) and the MyNode method becomessetData(Integer). Therefore, the MyNode setData method does not override the Node setData method.

To solve this problem and preserve the [polymorphism](file:///C:\project\books\javatutorials\tutorial\java\IandI\polymorphism.html) of generic types after type erasure, a 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);

}

// ...

}

As you can see, the bridge method, which has the same method signature as the Node class's setData method after type erasure, delegates to the original setData method.

**Non-Reifiable Types**

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> andList<Number>; the JVM cannot tell the difference between these types at runtime. As shown in [Restrictions on Generics](file:///C:\project\books\javatutorials\tutorial\java\generics\restrictions.html), there are certain situations where non-reifiable types cannot be used: in an instanceof expression, for example, or as an element in an array.

## 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.

## 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(stringListA, "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:

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 toObject[] 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>.

## 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 aClassCastException 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:

@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:

@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[Annotations](file:///C:\project\books\javatutorials\tutorial\java\annotations\index.html).

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++