Reifiable Types and Bounded and UnBounded in Java

<http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.6>

**Reifiable Types**

Because some type information is erased during compilation, not all types are available at run time. **Types that are completely available at run time are known as *reifiable types***.

A type is *reifiable* if and only if one of the following holds:

* It refers to a non-generic class or interface type declaration.
* It is a parameterized type in which all type arguments are unbounded wildcards ([§4.5.1](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.5.1)).
* It is a raw type ([§4.8](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.8)).
* It is a primitive type ([§4.2](http://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html#jls-4.2)).
* It is an array type ([§10.1](http://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html#jls-10.1)) whose element type is reifiable.
* It is a nested type where, for each type T separated by a ".", T itself is reifiable.

*For example, if a generic class X<T> has a generic member class Y<U>, then the type X<?>.Y<?> is reifiable because X<?> is reifiable and Y<?> is reifiable. The type X<?>.Y<Object> is not reifiable because Y<Object> is not reifiable.*

<http://stackoverflow.com/questions/18848885/why-following-types-are-reifiable-non-reifiable-in-java>

Understand the meaning of this two terms.

**Reifiable means whose type is fully available at run time means java compiler do not need any process of type erasure**.

**Non-Reifiable** means java compiler needs type erasure process because type is not fully available.

**A type is reifiable if it is one of the following:**

**1. A primitive type (such as int) :**

Here think that when you write or use any int as a reference, do you think that compiler needs any process for identification for the type of int? no because int is int.... same for all primitive type

**2. A nonparameterized class or interface type (such as Number, String, or Runnable)**

same answer as i told in previous answer that compiler do not need any type erasure for Number, String, or Runnable.

**3. A parameterized type in which all type arguments are unbounded wildcards (such as List<?>, ArrayList<?>, or Map<?, ?>)**

All unbounded wildcard are accepted as reifiable type because it is already mention in definition of reifiable type, now it is up to the API developer why they consider it as a reifiable type.

**4. A raw type (such as List, ArrayList, or Map) ::**

same answer as first question

**5. An array whose component type is reifiable(such as int[], Number[], List<?>[], List[], or int[][]) ::**

same answer as first question

**A type is not reifiable if it is one of the following:**

**6. A type variable(such as T) :**

Because java can not identify the type of T, Compiler needs type erasure to identify the type.

**7. A parameterized type with actual parameters (such as List<Number>, ArrayList<String>, or Map<String, Integer>)**:

Here all type is a generic type, at runtime compiler see List as List ... so as per definition of Non-refiable all these collection are consider as a non reifiable.

**8. A parameterized type with a bound (such as List<? extends Number> or Comparable<? super String>).**

<http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ106>

**What is a reifiable type?**

|  |
| --- |
| ***A type whose type information is fully available at runtime, that is, a type that does not lose information in the course of type erasure.*** |

The following types are reifiable:

* primitive types
* non-generic (or non-parameterized) reference types
* unbounded wildcard instantiations
* raw types
* arrays of any of the above

The non-reifiable types, which lose type information as a side effect of type erasure, are:

* instantiations of a generic type with at least one concrete type argument
* instantiations of a generic type with at least one bounded wildcard as type argument

## What are bounded and unbounded wildcards in Generics

<http://javarevisited.blogspot.com/2012/01/how-to-sort-arraylist-in-java-example.html>*bounded and unbounded wildcards in generics* are used to bound any Type. Type can be upper bounded by using <? extends T> where **all Types must be sub-class of T** or lower bounded using <? super T> where **all Types required to be the super class of T**, here T represent the lower bound. Single <?> is called an unbounded wildcard in generic and it can represent any type, similar to Object in Java. For example  **List<?>** can represent any List e.g. List<String> or List<Integer> its provides highest level of flexibility on passing method argument.

What is Type Erasure

<http://www.informit.com/articles/article.aspx?p=2025637&seqNum=8>

## Type Erasure

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 achieve the following:

* 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 element 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 shows how a compiler sometimes creates a synthetic method, called a bridge method, as part of the type erasure process.

Consider 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);

}

}

Now, 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 changes as follows:

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 inherited setData(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 are as follows:

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 match. The Node method becomes setData(Object) and the MyNode method becomes setData(Integer). Therefore, the MyNodesetData method does not override the NodesetData method. To solve this problem and preserve the polymorphism 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.

#### **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 (e.g., 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.