Reifiable Types and Bounded and UnBounded in Java

**Reifiable Types**

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

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

**Reifiable Types are**

* **A primitive type**
* **A nonparameterized class or interface type (such as Number, String, or Runnable)**
* **A parameterized type in which all type arguments are unbounded wildcards (such as List<?>, ArrayList<?>, or Map<?, ?>), because, it uses Object**
* **An array whose component type is reifiable(such as int[], Number[], List<?>[], List[]**

**Non-Reifiable Types are**

* **A type variable(such as T)**
* **A parameterized type with actual parameters (such as List<Number>, or Map<String, Integer**
* **A parameterized type with a bound (such as List<? extends Number> or Comparable<? super String>**

## What are bounded and unbounded wildcards in Generics

* Type can be upper bounded by using <? extends T> where **all Types must be sub-class of T**
* Type can be lower bounded using <? super T> where **all Types required to be the super class of T**
* **Single <?> is called an unbounded wildcard** in generic and it can represent any type, similar to Object in Java.

What is Type Erasure

## Type Erasure

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.
* Insert type casts if necessary to preserve type safety.
* Generate bridge methods to preserve polymorphism in extended generic types.

**Bridge Methods in Java**

**Java Bridge Methods are synthetic methods to resolve the incompatibilities between the covariant return types of both sub class and super class methods. It handles the situation in which type erasure in subclass overriding method does not produce the same type erasure in the super class method. Sometimes to support type-erasure use case of generics, Java compiler creates a synthetic method, which is called a bridge method.** This is a synthetic method created by the Java compiler while compiling a class or interface that extends a parameterized class or implements a parameterized interface where method signatures may be slightly different or ambiguous.

**public class MyComparator implements Comparator<Integer> {**

**public int compare(Integer a, Integer b) {**

**//**

**}**

**}**

This can't be used in its raw form, passing two Objects to compare.

public class MyComparator implements Comparator<Integer> {

public int compare(Integer a, Integer b) {

//

}

**//THIS is a "bridge method"**

**public int compare(Object a, Object b) { 🡸 Bridge Method**

**return compare( (Integer)a, (Integer)b );**

**}**

}

Now the class can be used in its raw form as well:

**Object a = 5;**

**Object b = 6;**

**Comparator rawComp = new MyComparator();**

**int comp = rawComp.compare(a, b);**

**Another example**

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;

}

}

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;

It's a method that allows a class extending a generic class or implementing a generic interface (with a concrete type parameter) to still be used as a raw type.

After type erasure, the Node and MyNode classes become:

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;

}

}

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);

}

}

#### **Heap Pollution**

Mixing generically typed code with raw typed code is one common source of heap pollution. When generic and nongeneric types are used together incorrectly, it creates heap pollution.

**Noncompliant Code**

**class** Utility {

**private** **static** **void** addToList(List list, Object obj) {

    list.add(obj); // Unchecked warning

  }

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

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

    addToList(list, 42);

    System.out.println(list.get(0));  // Throws ClassCastException

  }

}

**Compliant Solution (Parameterized Collection)**

**class** Utility {

**private** **static** **void** addToList(List<String> list, String str) {

    list.add(str);     // No warning generated

  }

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

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

    addToList(list, "42");

    System.out.println(list.get(0));

  }

}

[**heap**](https://en.wikipedia.org/wiki/Memory_management#Manual_memory_management)**pollution** is a situation that arises when a variable of a [parameterized type](https://en.wikipedia.org/wiki/Java_syntax#Generics) refers to an object that is not of that parameterized type.