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.

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.

public class Box<T> {

private T data;

public Box(T data) {

this.data = data;

}

public T getData() { return data; }

}

In the above example, "T" is unbounded type. So Java Compiler replaces Every Occurrence of T with Object.

public class Box {

private Object data;

public Box(Object data) {

this.data = data;

}

public Object getData() { return data; }

}

public class Container<T> {

public T data;

public Container(T data) {

this.data = data;

}

public void setData(T data) {

this.data = data;

}

}

public class IntegerContainer extends Container<Integer> {

public IntegerContainer(Integer data) {

super(data);

}

public void setData(Integer data) {

super.setData(data);

}

}

 Consider the following code.

IntegerContainer integerContainer = new IntegerContainer(5);

// A raw type - compiler throws an unchecked warning

Container container = integerContainer;

// Below statement throws a ClassCastException

container.setData("Hello");

Integer x = integerContainer.data;

After type erasure, this code becomes like below.

IntegerContainer integerContainer = new IntegerContainer(5);

Container container = (Container)integerContainer;

container.setData("Hello");

Integer x = (String)integerContainer.data;

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. When you decompile the class file ‘IntegerContainer.class’, it contains below code snippet.

$javap -c com/sample/app/IntegerContainer.class

Compiled from "IntegerContainer.java"

public class com.sample.app.IntegerContainer extends com.sample.app.Container<java.lang.Integer> {

public com.sample.app.IntegerContainer(java.lang.Integer);

Code:

0: aload\_0

1: aload\_1

2: invokespecial #1 // Method com/sample/app/Container."<init>":(Ljava/lang/Object;)V

5: return

public void setData(java.lang.Integer);

Code:

0: aload\_0

1: aload\_1

2: invokespecial #2 // Method com/sample/app/Container.setData:(Ljava/lang/Object;)V

5: return

public void setData(java.lang.Object);

Code:

0: aload\_0

1: aload\_1

2: checkcast #3 // class java/lang/Integer

5: invokevirtual #4 // Method setData:(Ljava/lang/Integer;)V

8: return

}

#### **Heap Pollution**

In the [Java programming](https://en.wikipedia.org/wiki/Java_programming) language, [**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. Heap pollution in Java can occur when type arguments and variables are not [reified](https://en.wikipedia.org/wiki/Reification_(computer_science)) at run-time. As a result, different parameterized types are [implemented](https://en.wikipedia.org/wiki/Implementation_(computer_science)) by the same class or [interface](https://en.wikipedia.org/wiki/Interface_(Java)) at run time. For example, **heap pollution occurs when mixing raw types and parameterized types or when performing unchecked casts**. The **heap pollution scenario refers to a situation where we attempt to make an unchecked cast when working with generics and non-generic type classes**.

**Heap Pollution in a non-[varargs](https://en.wikipedia.org/wiki/Variadic_function" \o "Variadic function) context**

**public** **class** **HeapPollutionDemo**

{

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

{

Set s = **new** TreeSet<Integer>();

Set<String> ss = s; *// unchecked warning*

s.add(**new** Integer(42)); *// another unchecked warning*

Iterator<String> iter = ss.iterator();

**while** (iter.hasNext())

{

String str = iter.next(); *// ClassCastException thrown*

System.out.println(str);

}

}

}