1. Write a generic method to count the number of elements in a collection that are prime numbers.

Attached as Question1.java

1. Will the following class compile? If not, why?

public final class Algorithm {

public static <T> T max(T x, T y) {

return x > y ? x : y;

}

}

This method will not compile because you cannot use > on object references.

1. Write a generic method to exchange the positions of two different elements in an array.

Attached as Question3.java

1. If the compiler erases all type parameters at compile time, why should you use generics?

Generics allow Java developers to write generic code while ensuring type safety. The generic types are replaced by their lower bounds, so the type safety enforced by the compiler is maintained during run time.

1. What is the following class converted to after type erasure?

public class Pair<K, V> {

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey(); { return key; }

public V getValue(); { return value; }

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

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

private K key;

private V value;

}

Types K and V have no lower bound, so they are placed by the type Object.  
Attached Question5.java

1. What is the following method converted to after type erasure?

public static <T extends Comparable<T>>

int findFirstGreaterThan(T[] at, T elem) {

// ...

}

Attached as Question6.java

1. Will the following method compile? If not, why?

public static void print(List<? extends Number> list) {

for (Number n : list)

System.out.print(n + " ");

System.out.println();

}

The method compiles and prints expected results. Number implements .toString()

1. Write a generic method to find the maximal element in the range [begin, end) of a list.

Attached as Question8.java

1. Will the following class compile? If not, why?

public class Singleton<T> {

public static T getInstance() {

if (instance == null)

instance = new Singleton<T>();

return instance;

}

private static T instance = null;

}

This class will not compile because the type T belongs to instances of an object and is not a static property of the class. The static ‘instance’ variable cannot have a different type as Singleton<String> as it would as Singleton<Integer>.

1. Given the following classes:

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

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

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

class Node<T> { /\* ... \*/ }

Will the following code compile? If not, why?

Node<Circle> nc = new Node<>();

Node<Shape> ns = nc;

This code will not compile because Node<Circle> does not extend Node<Shape>. Circle extends Shape, but that relationship does not extend to their containers.

1. Consider this class:

class Node<T> implements Comparable<T> {

public int compareTo(T obj) { /\* ... \*/ }

// ...

}

Will the following code compile? If not, why?

Node<String> node = new Node<>();

Comparable<String> comp = node;

This code will compile because, unlike the question 10 code, Node<String> implements Comparable<String>. The types are compatible.