1. Write a generic method to count the number of elements in a collection that have a specific property (for example, odd integers, prime numbers, palindromes).  
     
   **Answer**:
2. public final class Algorithm {
3. public static <T> int countIf(Collection<T> c, UnaryPredicate<T> p) {
4. int count = 0;
5. for (T elem : c)
6. if (p.test(elem))
7. ++count;
8. return count;
9. }
10. }

where the generic UnaryPredicate interface is defined as follows:

public interface UnaryPredicate<T> {

public boolean test(T obj);

}

For example, the following program counts the number of odd integers in an integer list:

import java.util.\*;

class OddPredicate implements UnaryPredicate<Integer> {

public boolean test(Integer i) { return i % 2 != 0; }

}

public class Test {

public static void main(String[] args) {

Collection<Integer> ci = Arrays.asList(1, 2, 3, 4);

int count = Algorithm.countIf(ci, new OddPredicate());

System.out.println("Number of odd integers = " + count);

}

}

The program prints:

Number of odd integers = 2

1. Will the following class compile? If not, why?
2. public final class Algorithm {
3. public static <T> T max(T x, T y) {
4. return x > y ? x : y;
5. }
6. }

**Answer**: No. The greater than (>) operator applies only to primitive numeric types.

1. Write a generic method to exchange the positions of two different elements in an array.  
     
   **Answer**:
2. public final class Algorithm {
3. public static <T> void swap(T[] a, int i, int j) {
4. T temp = a[i];
5. a[i] = a[j];
6. a[j] = temp;
7. }
8. }
9. If the compiler erases all type parameters at compile time, why should you use generics?  
     
   **Answer**: You should use generics because:
   * The Java compiler enforces tighter type checks on generic code at compile time.
   * Generics support programming types as parameters.
   * Generics enable you to implement generic algorithms.
10. What is the following class converted to after type erasure?
11. public class Pair<K, V> {
12. public Pair(K key, V value) {
13. this.key = key;
14. this.value = value;
15. }
16. public K getKey(); { return key; }
17. public V getValue(); { return value; }
18. public void setKey(K key) { this.key = key; }
19. public void setValue(V value) { this.value = value; }
20. private K key;
21. private V value;
22. }

**Answer**:

public class Pair {

public Pair(Object key, Object value) {

this.key = key;

this.value = value;

}

public Object getKey() { return key; }

public Object getValue() { return value; }

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

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

private Object key;

private Object value;

}

1. What is the following method converted to after type erasure?
2. public static <T extends Comparable<T>>
3. int findFirstGreaterThan(T[] at, T elem) {
4. // ...
5. }

**Answer**:

public static int findFirstGreaterThan(Comparable[] at, Comparable elem) {

// ...

}

1. Will the following method compile? If not, why?
2. public static void print(List<? extends Number> list) {
3. for (Number n : list)
4. System.out.print(n + " ");
5. System.out.println();
6. }

**Answer**: Yes.

1. Write a generic method to find the maximal element in the range [begin, end) of a list.  
     
   **Answer**:
2. import java.util.\*;
3. public final class Algorithm {
4. public static <T extends Object & Comparable<? super T>>
5. T max(List<? extends T> list, int begin, int end) {
6. T maxElem = list.get(begin);
7. for (++begin; begin < end; ++begin)
8. if (maxElem.compareTo(list.get(begin)) < 0)
9. maxElem = list.get(begin);
10. return maxElem;
11. }
12. }
13. Will the following class compile? If not, why?
14. public class Singleton<T> {
15. public static T getInstance() {
16. if (instance == null)
17. instance = new Singleton<T>();
18. return instance;
19. }
20. private static T instance = null;
21. }

**Answer**: No. You cannot create a static field of the type parameter T.

1. Given the following classes:
2. class Shape { /\* ... \*/ }
3. class Circle extends Shape { /\* ... \*/ }
4. class Rectangle extends Shape { /\* ... \*/ }
5. class Node<T> { /\* ... \*/ }

Will the following code compile? If not, why?

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

Node<Shape> ns = nc;

**Answer**: No. Because Node<Circle> is not a subtype of Node<Shape>.

1. Consider this class:
2. class Node<T> implements Comparable<T> {
3. public int compareTo(T obj) { /\* ... \*/ }
4. // ...
5. }

Will the following code compile? If not, why?

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

Comparable<String> comp = node;

**Answer**: Yes.

1. How do you invoke the following method to find the first integer in a list that is relatively prime to a list of specified integers?
2. public static <T>
3. int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p)

Note that two integers *a* and *b* are relatively prime if gcd(*a, b*) = 1, where gcd is short for greatest common divisor.  
  
**Answer**:

import java.util.\*;

public final class Algorithm {

public static <T>

int findFirst(List<T> list, int begin, int end, UnaryPredicate<T> p) {

for (; begin < end; ++begin)

if (p.test(list.get(begin)))

return begin;

return -1;

}

// x > 0 and y > 0

public static int gcd(int x, int y) {

for (int r; (r = x % y) != 0; x = y, y = r) { }

return y;

}

}

The generic UnaryPredicate interface is defined as follows:

public interface UnaryPredicate<T> {

public boolean test(T obj);

}

The following program tests the findFirst method:

import java.util.\*;

class RelativelyPrimePredicate implements UnaryPredicate<Integer> {

public RelativelyPrimePredicate(Collection<Integer> c) {

this.c = c;

}

public boolean test(Integer x) {

for (Integer i : c)

if (Algorithm.gcd(x, i) != 1)

return false;

return c.size() > 0;

}

private Collection<Integer> c;

}

public class Test {

public static void main(String[] args) throws Exception {

List<Integer> li = Arrays.asList(3, 4, 6, 8, 11, 15, 28, 32);

Collection<Integer> c = Arrays.asList(7, 18, 19, 25);

UnaryPredicate<Integer> p = new RelativelyPrimePredicate(c);

int i = ALgorithm.findFirst(li, 0, li.size(), p);

if (i != -1) {

System.out.print(li.get(i) + " is relatively prime to ");

for (Integer k : c)

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

System.out.println();

}

}

}

The program prints:

11 is relatively prime to 7 18 19 25