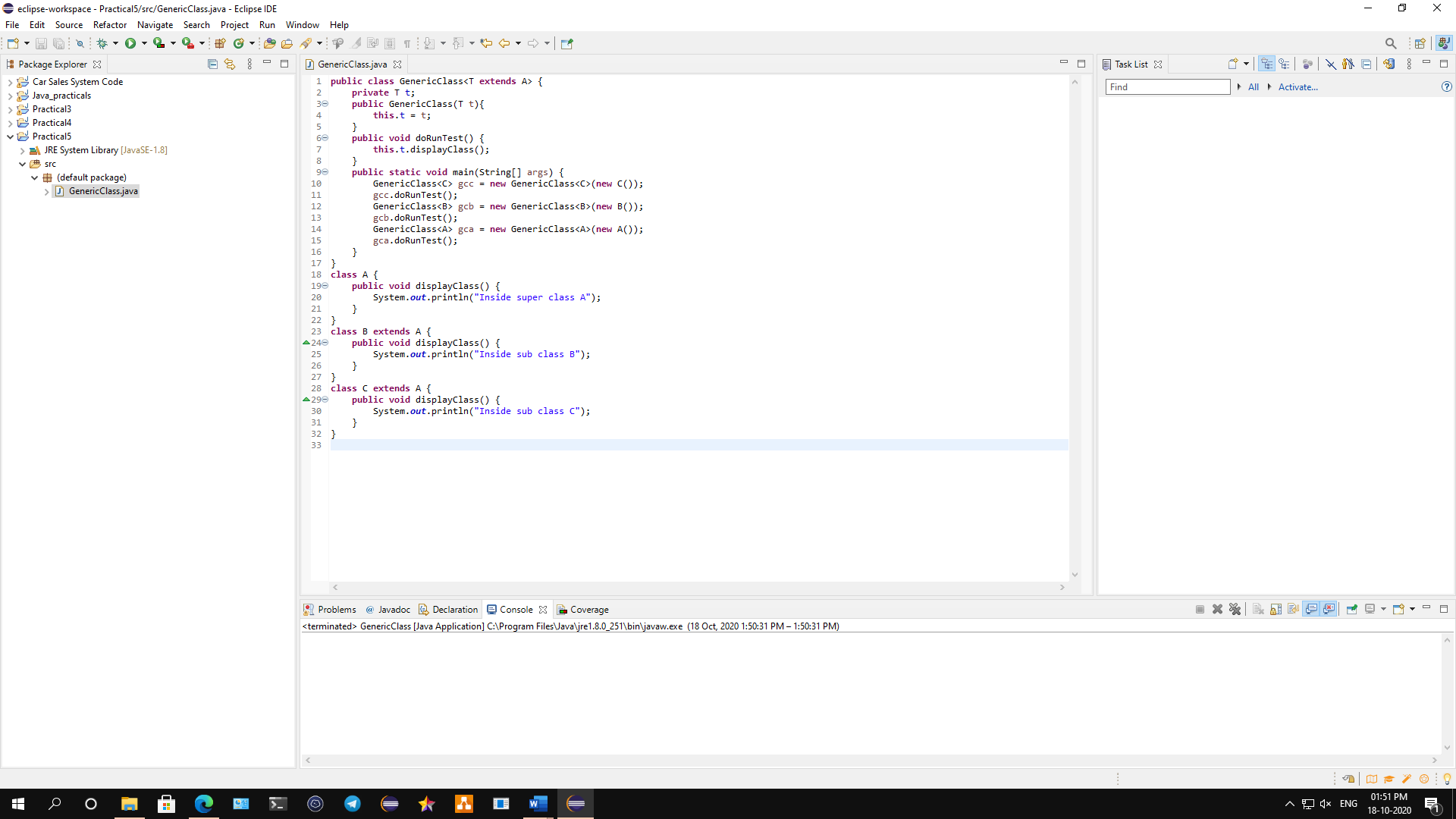
**Practical No 5**

**A) Implement bounded types (extendsuperclass) with generics.**

**Aim: Write a program to implement bounded types (extendsuperclass) with generics.**

**Description:**

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively. Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time. Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements. We have defined generic class T which extend A and then we defined the private keyword T with public generic class. The t.display class will display the output with other generic classes. Then we have defined class A which will show that class A is inside the sub class A. Then class B has been extended with class A which shows class A is inside of sub class B and for class C the output is repeated same as other classes.



**Conclusion: We have written a program to implement bounded types (extendsuperclass) with generics.**

**Code:**

**public** **class** GenericClass<T **extends** A> {

**private** T t;

**public** GenericClass(T t){

**this**.t = t;

}

**public** **void** doRunTest() {

**this**.t.displayClass();

}

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

GenericClass<C> gcc = **new** GenericClass<C>(**new** C());

gcc.doRunTest();

GenericClass<B> gcb = **new** GenericClass<B>(**new** B());

gcb.doRunTest();

GenericClass<A> gca = **new** GenericClass<A>(**new** A());

gca.doRunTest();

}

}

**class** A {

**public** **void** displayClass() {

System.***out***.println("Inside super class A");

}

}

**class** B **extends** A {

**public** **void** displayClass() {

System.***out***.println("Inside sub class B");

}

}

**class** C **extends** A {

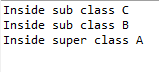
**public** **void** displayClass() {

System.***out***.println("Inside sub class C");

}

}

**Output:**

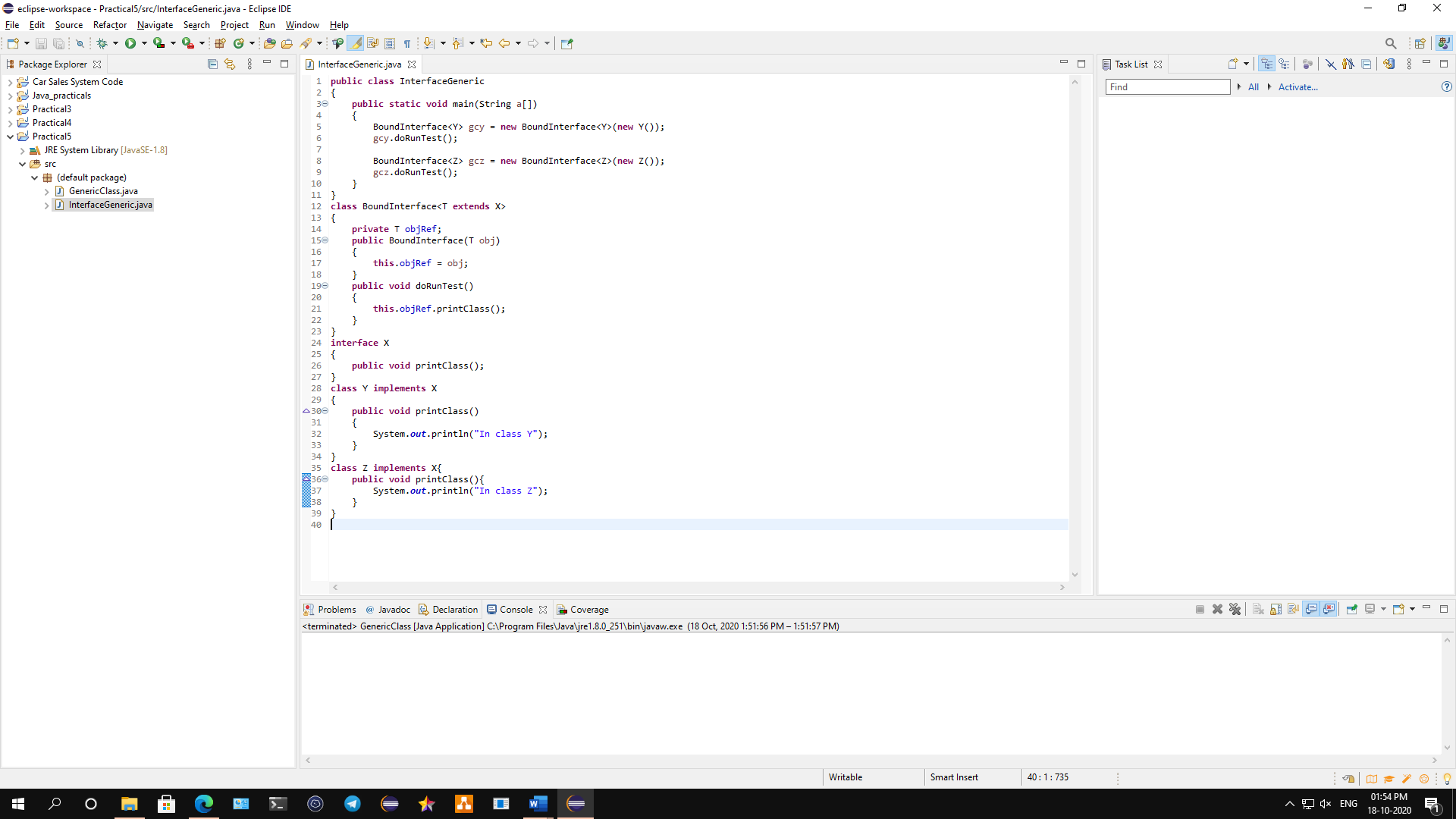


**B) Implement bounded types (implements an interface) with generics.**

**Aim: Write a java program to implement bounded types (implements an interface) with generics.**

**Description:**

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for. To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound. In this program we have defined bound interface as Y and Z. Then we use class bound that extend T and X with private obj T with public BoundInterface class. Then we have defined X interface with class Y that implements class X so it will print that X is in class Y and then we defined class Z which implements X that will show output as class X is in class Z.



**Conclusion: We have written a program to implement bounded types (implements an interface) with generics.**

**Code:**

**public** **class** InterfaceGeneric

{

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

{

BoundInterface<Y> gcy = **new** BoundInterface<Y>(**new** Y());

gcy.doRunTest();

BoundInterface<Z> gcz = **new** BoundInterface<Z>(**new** Z());

gcz.doRunTest();

}

}

**class** BoundInterface<T **extends** X>

{

**private** T objRef;

**public** BoundInterface(T obj)

{

**this**.objRef = obj;

}

**public** **void** doRunTest()

{

**this**.objRef.printClass();

}

}

**interface** X

{

**public** **void** printClass();

}

**class** Y **implements** X

{

**public** **void** printClass()

{

System.***out***.println("In class Y");

}

}

**class** Z **implements** X{

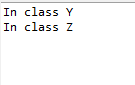
**public** **void** printClass(){

System.***out***.println("In class Z");

}

}

**Output:**

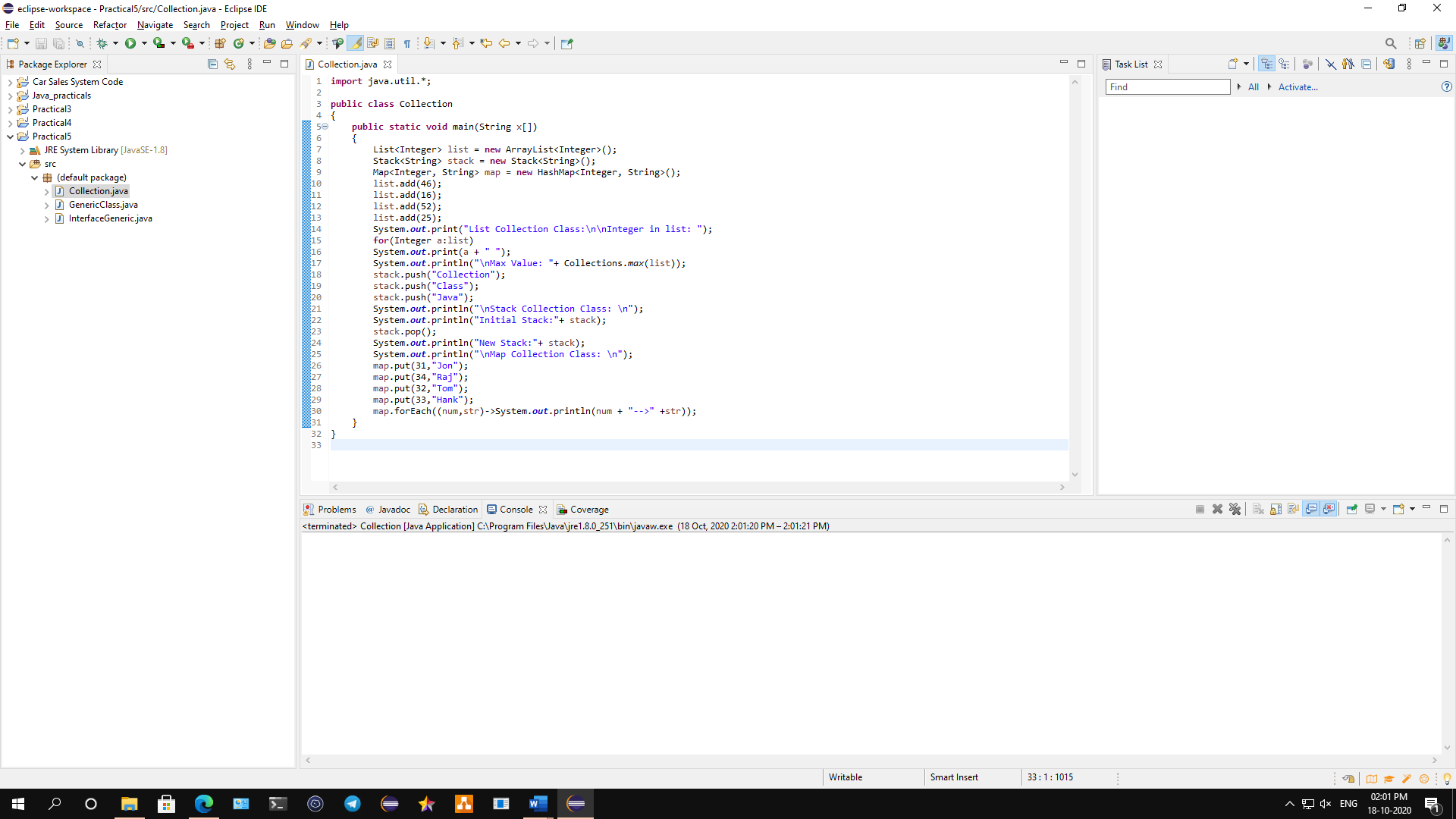


**C) Implement any three collection classes.**

**Aim: Write a java program to implement any three collection classes.**

**Description:**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion. Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes ([ArrayList](https://www.javatpoint.com/java-arraylist), Vector, [LinkedList](https://www.javatpoint.com/java-linkedlist), [PriorityQueue](https://www.javatpoint.com/java-priorityqueue), HashSet, LinkedHashSet, TreeSet). A framework is a set of [classes](https://www.geeksforgeeks.org/classes-objects-java/) and [interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) which provide a ready-made architecture. In order to implement a new feature or a class, there is no need to define a framework. However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task. **Consistent API, reduces programming effort and Increases program speed and quality are the advantages of collection frameworks. In this program we use Array as a list integer to find the list of integers with collection classes. We have use stack function to list max collection of classes and then we use pop function to pop out the list if the stack is full, then we got the map of collection class output.**



**Conclusion: We have written a java program to implement any three collection classes.**

**Code:**

**import** java.util.\*;

**public** **class** Collection

{

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

{

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

Stack<String> stack = **new** Stack<String>();

Map<Integer, String> map = **new** HashMap<Integer, String>();

list.add(46);

list.add(16);

list.add(52);

list.add(25);

System.***out***.print("List Collection Class:\n\nInteger in list: ");

**for**(Integer a:list)

System.***out***.print(a + " ");

System.***out***.println("\nMax Value: "+ Collections.*max*(list));

stack.push("Collection");

stack.push("Class");

stack.push("Java");

System.***out***.println("\nStack Collection Class: \n");

System.***out***.println("Initial Stack:"+ stack);

stack.pop();

System.***out***.println("New Stack:"+ stack);

System.***out***.println("\nMap Collection Class: \n");

map.put(31,"Jon");

map.put(34,"Raj");

map.put(32,"Tom");

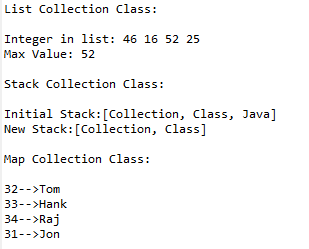
map.put(33,"Hank");

map.forEach((num,str)->System.***out***.println(num + "-->" +str));

}

}

**Output:**

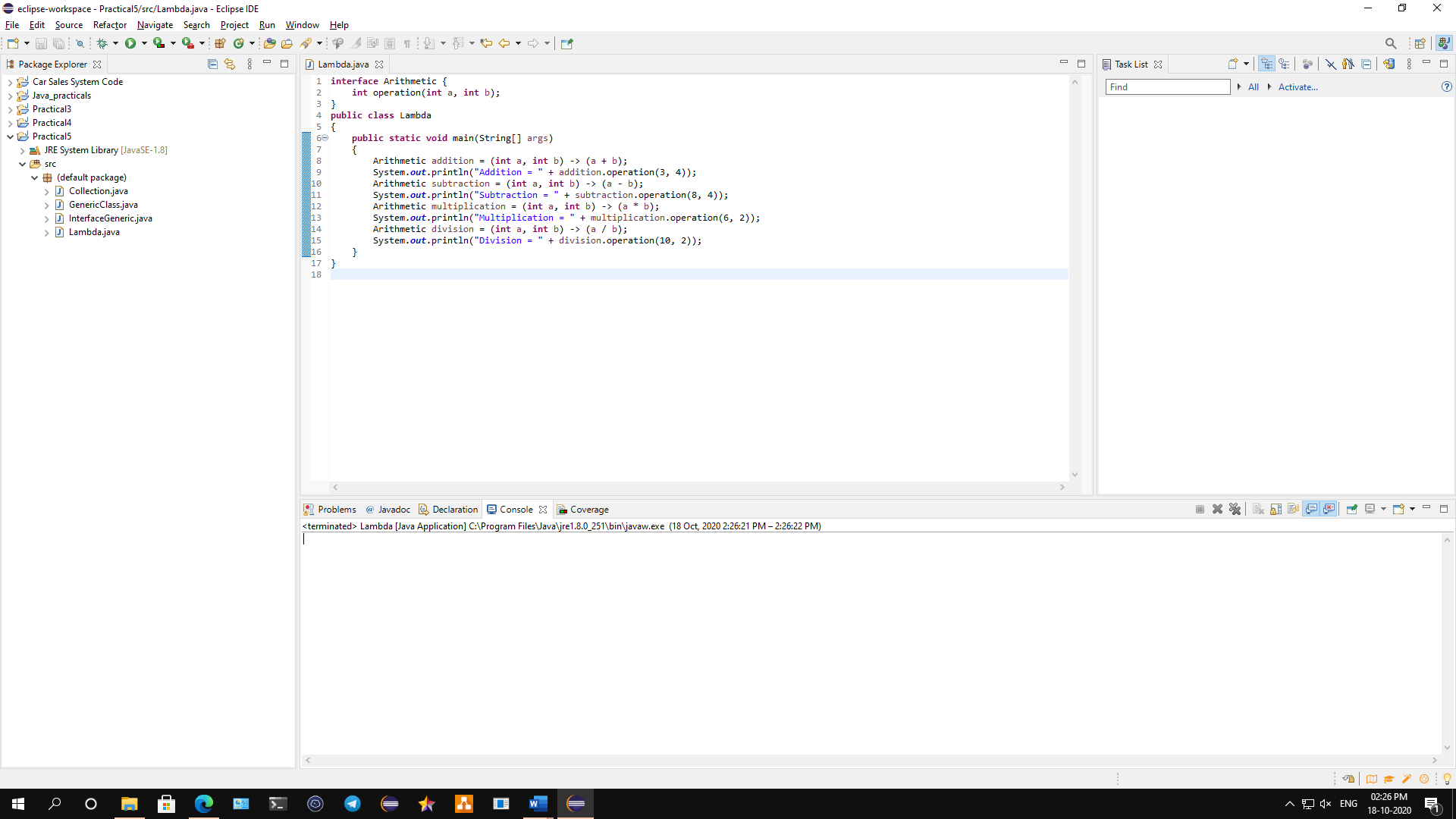


**D) Perform addition, subtraction, multiplication as well as division using Lambda Expression.**

**Aim: Write a java program to perform addition, subtraction, multiplication as well as division using Lambda Expression.**

**Description:**

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method. Lambda expressions are usually passed as parameters to a function. Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists. In this program we first defined the interface as arithmetic to find to arithmetic operations. Then we have defined Lamda as public class then we arithmetic interface to define addition, subtraction, multiplication and division operation. We will get the output once the program is done.



**Conclusion: We have written a java program to perform addition, subtraction, multiplication as well as division using Lambda Expression.**

**Code:**

**interface** Arithmetic {

**int** operation(**int** a, **int** b);

}

**public** **class** Lambda

{

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

{

Arithmetic addition = (**int** a, **int** b) -> (a + b);

System.***out***.println("Addition = " + addition.operation(3, 4));

Arithmetic subtraction = (**int** a, **int** b) -> (a - b);

System.***out***.println("Subtraction = " + subtraction.operation(8, 4));

Arithmetic multiplication = (**int** a, **int** b) -> (a \* b);

System.***out***.println("Multiplication = " + multiplication.operation(6, 2));

Arithmetic division = (**int** a, **int** b) -> (a / b);

System.***out***.println("Division = " + division.operation(10, 2));

}

}

**Output:**

