# LESSON 4
# RECURSION

# Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, on the basis of which all creation emerges.

# Recursion

- Recursion is a problem-solving approach that can be used to generate simple solutions to certain kinds of problems that are difficult to solve by other means.
- In the field of artificial intelligence, recursion often is used to write programs that exhibit intelligent behavior.
- Recursion reduces a problem into one or more simpler versions of itself.
- A Java method is recursive, or exhibits recursion, if in its body it calls itself.

# Steps to Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

# Simple Countdown using recursion

```
public class RecursiveCountdown{
public static void main(String[] args){
countDown(3);
}
public static void countDown(int num){
if (num <=0){
System.out.println();
}
else{
System.out.print(num);
countDown(num - 1);
}
}
}
```

# General Examples

# Recursive Algorithm for Finding the Length of a String

`if` the string is empty (has no characters)

    the length is 0

`else`

    the length is 1 plus the length of the string that excludes the first character

# Recursive Algorithm for Finding the Length of a String (cont.)

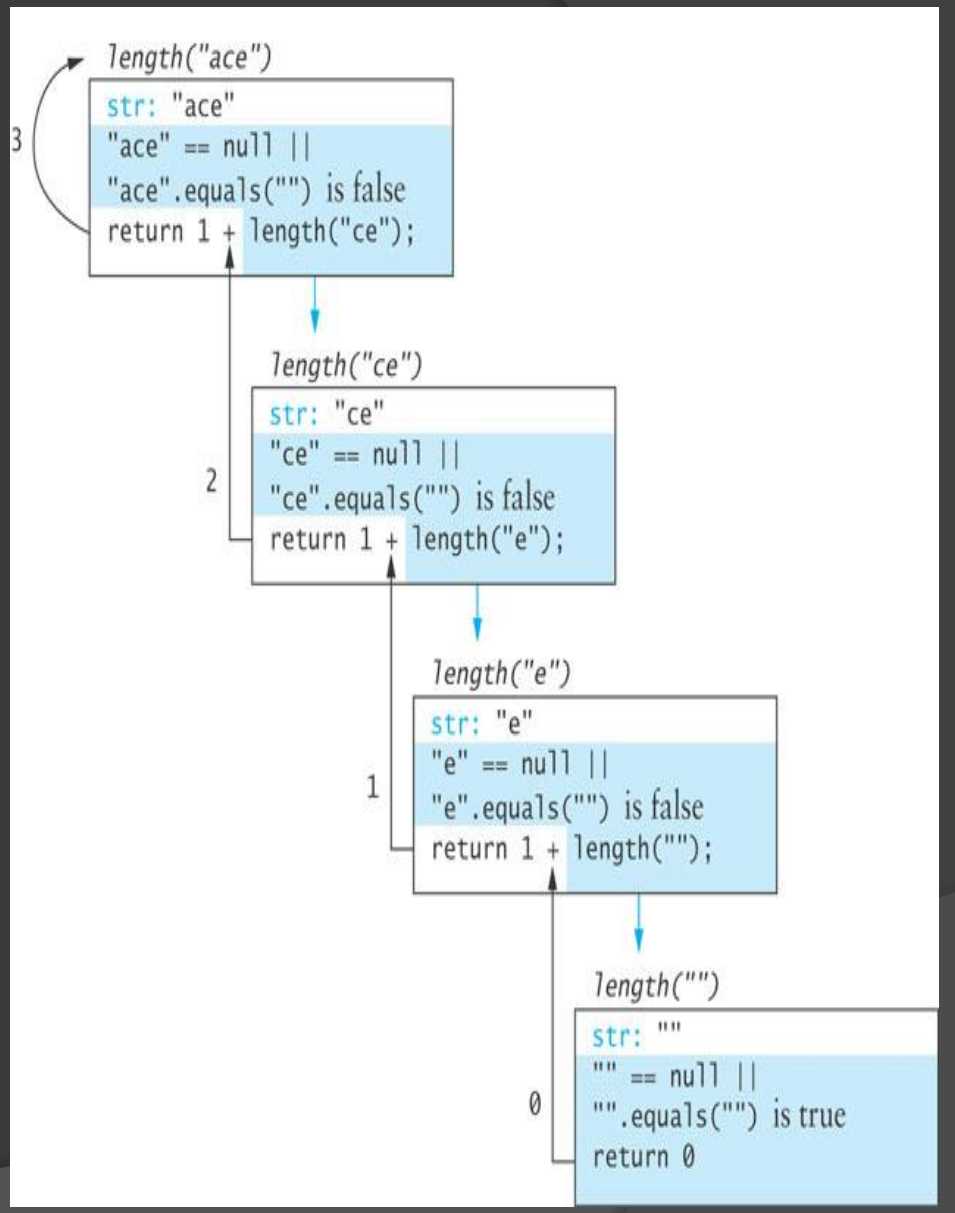
```
/** Recursive method length
    @param str The string
    @return The length of the string
*/
public static int length(String str) {
   if (str == null || str.equals(""))// base case
      return 0;
   else // Recursive case
      return 1 + length(str.substring(1));
}
```

# Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack and return will remove the activation frame from the stack and return to the previous call.

# Stack Trace

```
public static int length(String str) {
   // base case
   if (str == null || str.equals(""))
      return 0;
   else // Recursive case
      return 1 +
      length(str.substring(1));
}
```



# Write a Recursive Algorithm for Printing String Characters in Reverse

```
/** Recursive method printCharsReverse
    Problem : The argument string is displayed in reverse, one character
              per line
    @param str :The input string
*/
public static void printCharsReverse(String str) {
    if (str == null || str.equals(""))
       return;
    else {
       printCharsReverse(str.substring(1));
       System.out.println(str.charAt(0));
    }
}
```

# Main Point

When a recursion involves many redundant computations, one tries to write an iterative version of the method (using loops). Likewise, though all healing can in principle be done on the level of consciousness, when consciousness is not yet sufficiently established in its home, many steps of healing may be required to obtain the desired result.

# Mathematical & Date Structure Examples

- Factorial
- Fibonacci
- Linear Search

# Factorial Function

The factorial function on input n computes the product of the positive integers less than or equal to n.

For example, 5! = 5*4*3*2*1 = 120

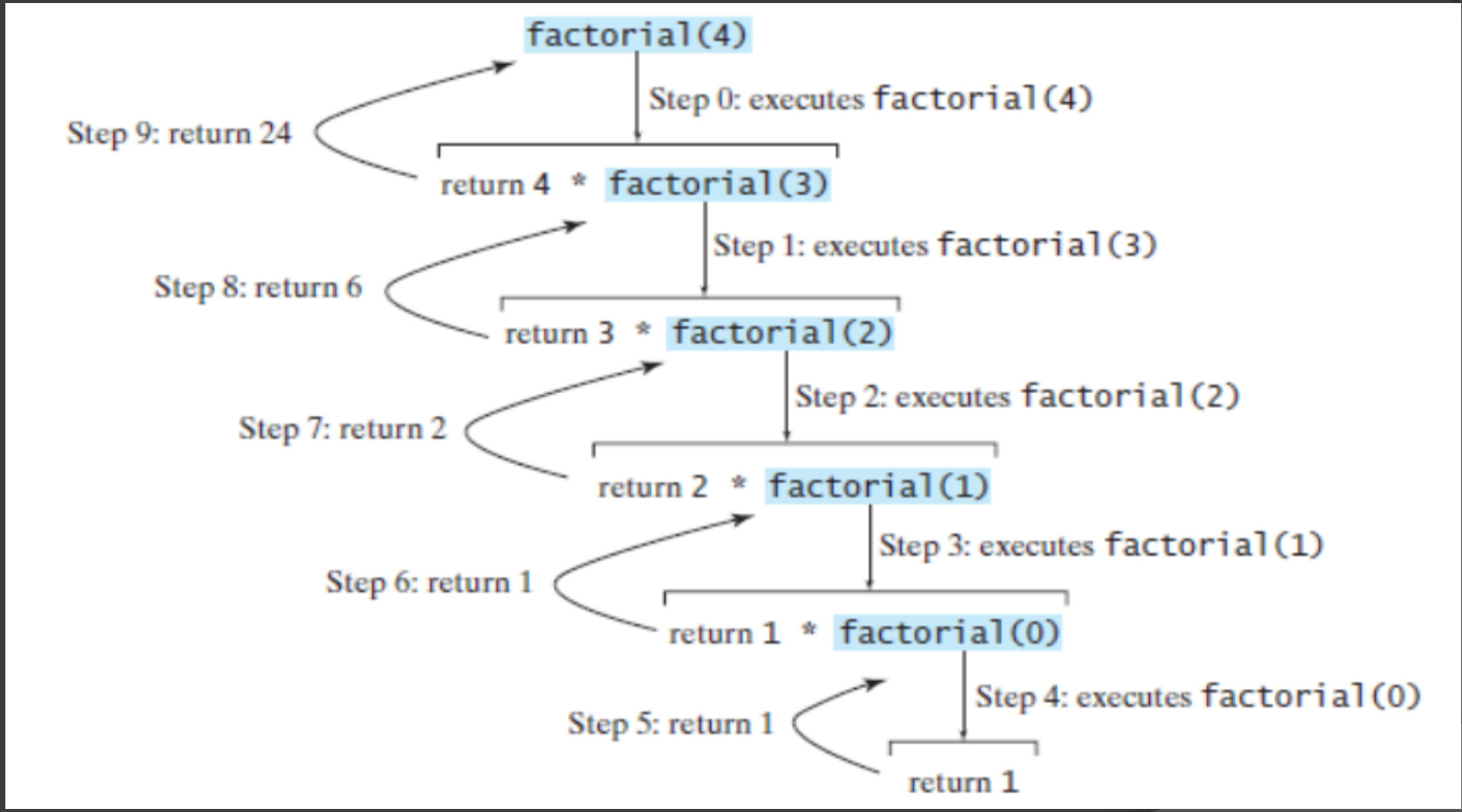
```
public static int fact(int num) {
        if(num == 0 || num == 1) {     //base case
                return 1;
        }
        return num * fact(num-1);
}
```

# Factorial Class

```
public class factorial{
	public static void main(String[] args) {
		System.out.println("Factorial of 6 = " + fact(6));
		System.out.println("Factorial of 10 = " + fact(10));
	}

	public static int fact(int num) {
		if(num == 0 || num == 1)
			return 1;
		else
			return num * fact(num - 1);
	}
}
```

# Stack Trace of Factorial



# What is the Output ?

```
public class test {
public static void main(String[] args) {
xMethod(1234567);
}
public static void xMethod(int n) {
if (n > 0) {
System.out.print(n % 10);
xMethod(n / 10);
} }
}
```

# Fibonacci Numbers

$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots,$

OR

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$

```
int fib(int n) {
        if(n == 0 || n == 1) {
                return n;
        }
        return fib(n-1) + fib(n-2);
}
```

# Recursive Implementations of a Utility Method

- Sorting, searching, and other manipulations of characters in a string or elements in arrays or lists are often done recursively. Sometimes (but not always), an implementation of such a utility provides a public method

```
public <return-value-type> thePublicMethod(params)
```

whose signature and return type make sense to potential users, and a private recursive method

```
private <ret-value-type> privateRecurMethod(otherParams)
```

which does the real work and is designed to call itself.

Example : Linear Search uses Utility Method

# Linear Search

- Searching an array can be accomplished using recursion
- Base cases for recursive search:
  - Empty array, target can not be found;

    result is -1
  - First element of the array being searched = target;

    result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

# Linear Search

```
// Utility Method - public
public static int linearSearch(Object[] items,  Object target) {
// invoke private method to pass all the arguments
return linearSearch(items,target,0);
}

// Recursive Method - private
private static int linearSearch(Object[] items, Object target, int posFirst) {
        if (posFirst == items.length) {
            return -1;
        } else if (target.equals(items[posFirst])) {
            return posFirst;
        } else {
            return linearSearch(items, target, posFirst + 1);
        }
    }
```

# main() method

```
public static void main(String[] args) {
// TODO Auto-generated method stub
Object[] i = { 10,20,15,12,65,50};
Object[] s = {"java","c#","SE","UML","WAP"};
// Utility Function Support from the client side - possible arguments
System.out.println("Searching of 12 in the list : " + linearSearch(i,12));
System.out.println("Searching of  FPP in the list : " + linearSearch(s," FPP");
}
```

# Recursion Versus Iteration

- There are similarities between recursion and iteration
- In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
- In recursion, the condition usually tests for a base case
- You can always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

# Summary

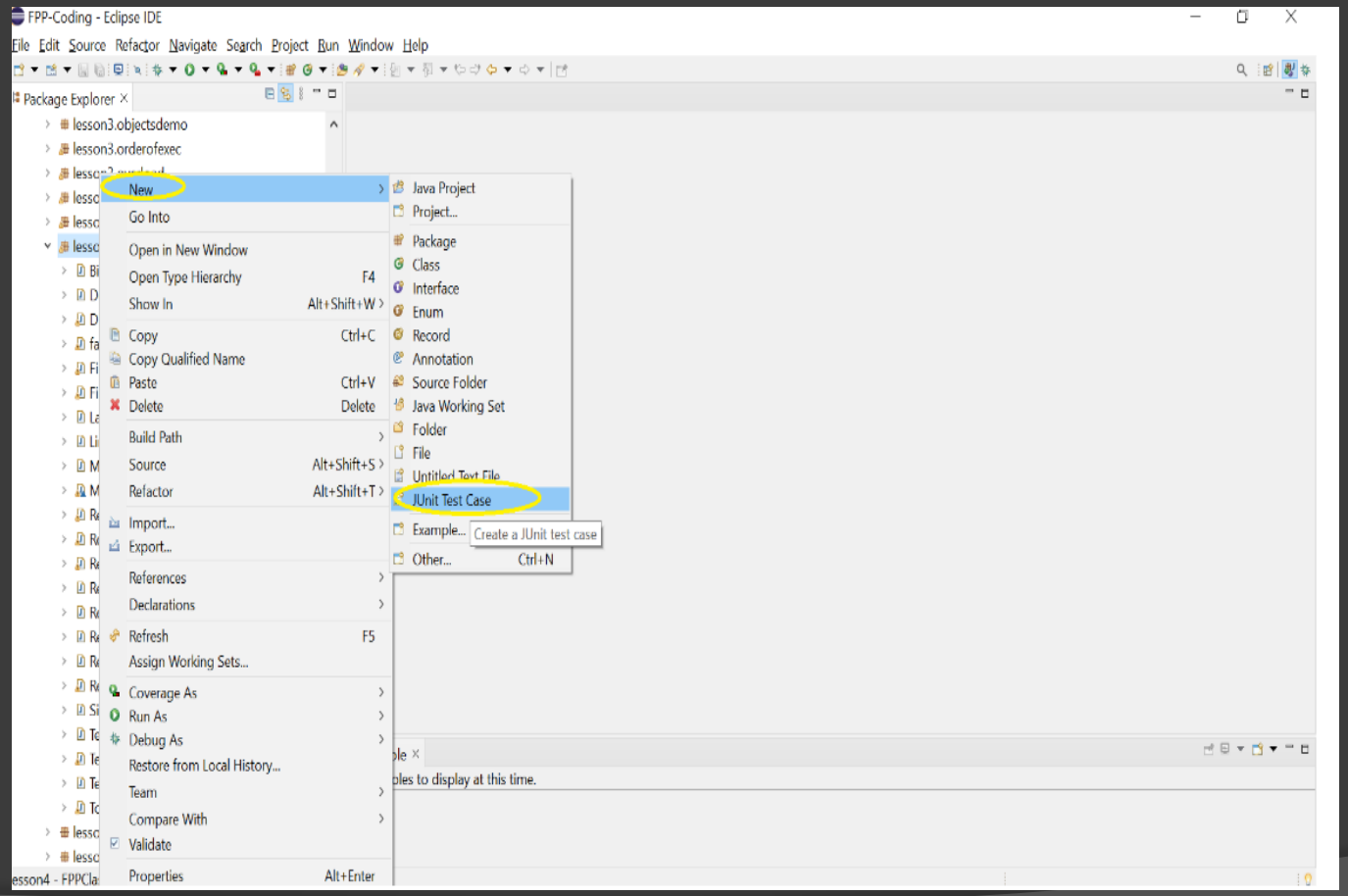
- A Java method is recursive, or exhibits recursion, if in its body it calls itself.
- A recursion is valid if the following criteria are met:
  - The method must have a base case which returns a value without making a self-call.
  - For every input to the method, the sequence of self-calls eventually leads to a self-call in which the base case is accessed.
- Sometimes recursion leads to redundant computations, which lead to slow running times (like Fibonacci). In such cases, an implementation using iteration instead of recursion should be done.
- When recursion is used to provide utility function support, the public method signature that is exposed to the client reveals only the parameters that are relevant for the client – not the special parameters that may be needed to implement the recursion.

# Including Unit Testing in Your Work Environment

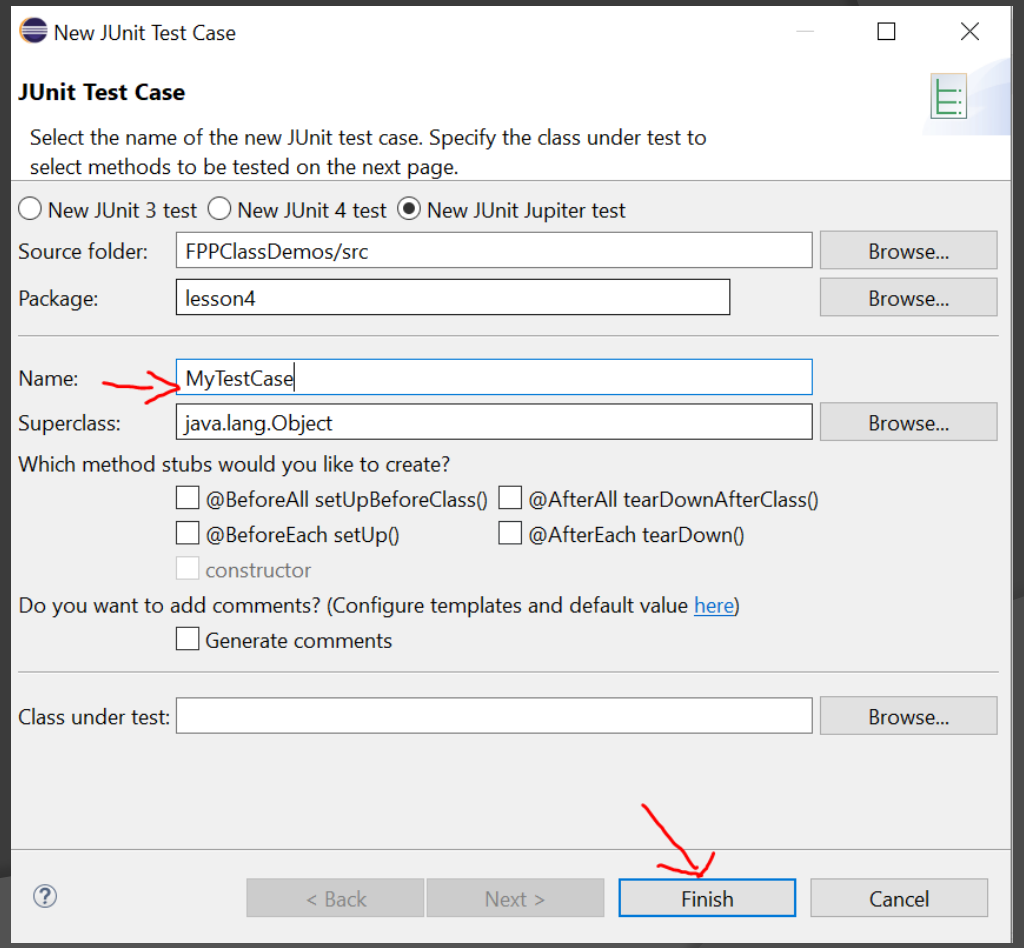
- A quick way to test your code as you develop is to run it from the main method as we have been doing
- A better way that is more reusable and useful for larger-scale team development is to have a parallel test project and use Junit.

# Junit Testing Step by Step

Step 1. Right Click on your Package→
Select New→ Junit Test Case

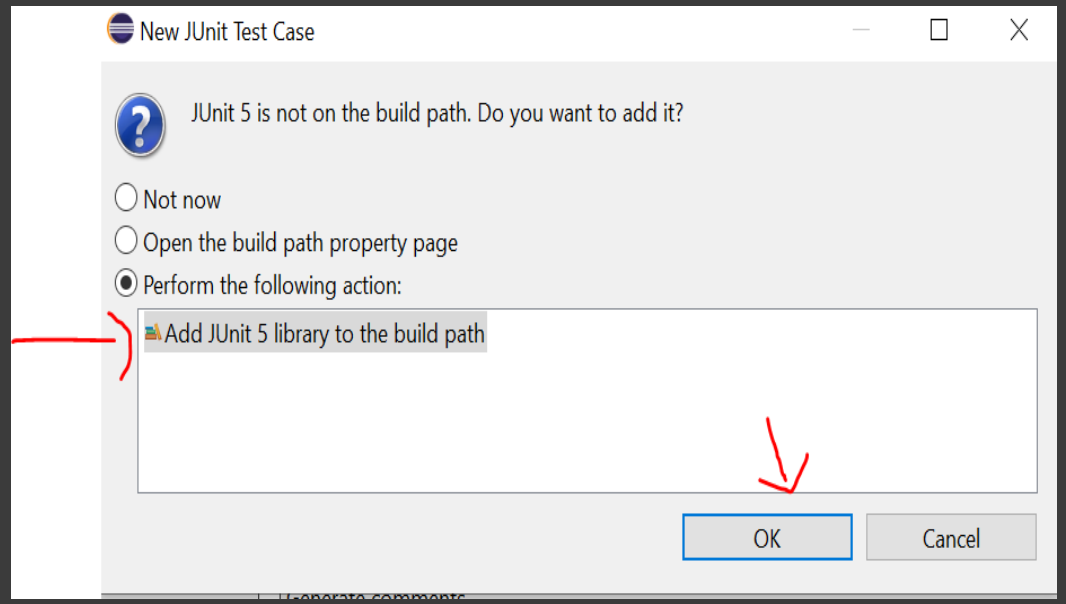



Step2: Give name for the Test Case and Select Finish

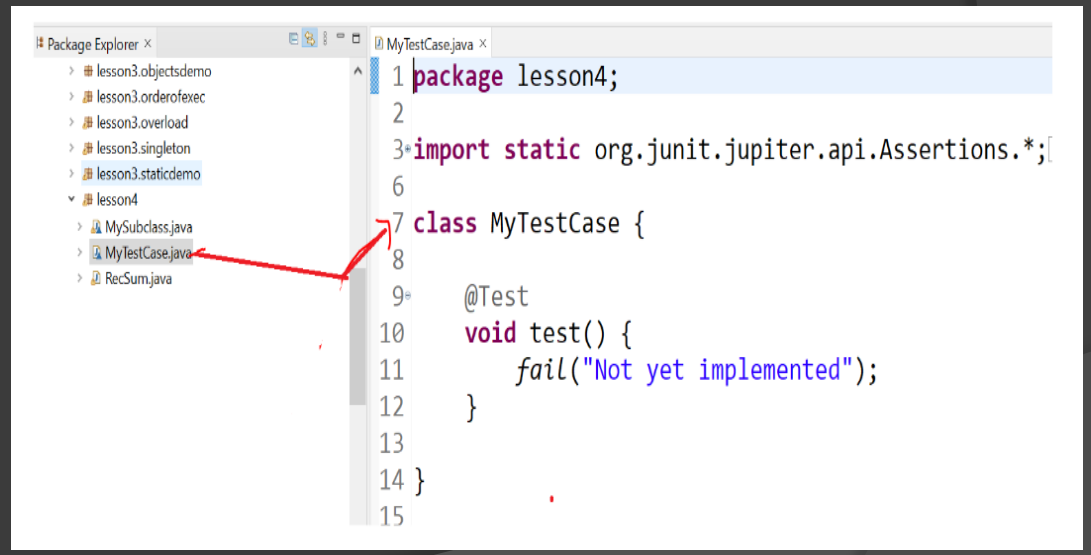


# Junit Testing Step by Step

Step 3: You will get the below screen to add the Junit library in your project



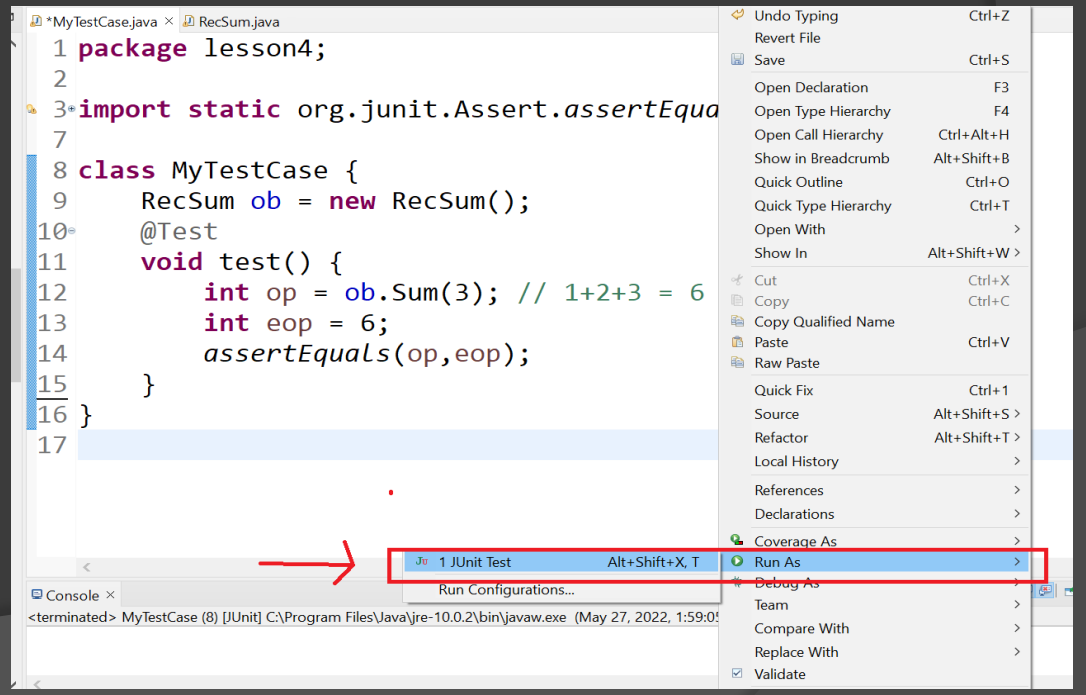

Step 4: You can see the Junit Test Case file as mentioned below



```
public class RecSum {
public static void main(String[] args) {
// if your input is 5 ==> 1+2+3+4+5
   RecSum r1 = new RecSum();
 System.out.println("Sum = " +
r1.Sum(5));
}
 public int Sum(int n){
          if (n == 1)            //base case
             return 1;
          else
    return Sum(n-1) + n; //Recursive case
          }
}
```

```
class MyTestCase {
RecSum ob = new RecSum();
@Test
void test() {
int op = ob.Sum(3); // 1+2+3 = 6
int eop = 6;
assertEquals(op,eop);
}
}
```

**Right Click, do as per the screenshot**



# Example to include more test() cases

```
public class MyTestCase {
StringFunction ob = new StringFunction();
@Test
public void test() {
        String op = ob.StringJoin("Java", "Program");
        String eop = "JavaProgram";
        assertEquals(op,eop);
}
@Test
public void test1() {
        int op = ob.Multiply(5, 2);
        int eop = 10;
        assertEquals(op,eop);
}
@Test
public void test2() {
        int op = ob.Sum(3); // 1+2+3 = 6
        int eop = 6;
        assertEquals(op,eop);
}
}
```

# Connecting the Parts of Knowledge With the Wholeness of Knowledge

*Recursion creates from self-referral activity*

1. In Java, it is possible for a method to call itself.
2. For a self-calling method to be a legitimate recursion, it must have a base case, and whenever the method is called, the sequence of self-calls must converge to the base case.

---

3. **Transcendental Consciousness:** TC is the self-referral field of existence, at the basis of all manifest existence.
4. **Wholeness moving within itself**: In Unity Consciousness, one sees that all activity in the universe springs from the self-referral dynamics of wholeness. The "base case" – the reference point – is always the Self, realized as Brahman.

# PRACTICE