Written Solutions

Solution 01:

Runtime Polymorphism:

Runtime polymorphism is a process in which a call to an overridden method is resolved at runtime rather than compile time. It means if the child class provides specific implementation of a method that has been provided by one of its parent classes, then implementation of the child class will be executed.

Describing Circle and Rectangle by referencing the concepts of OOP:

The concept of generalization in OOP means that an object encapsulates common state and behaviour for a category of objects. The general object in case of Circle and Rectangle is the geometric shape. Both shapes have area and perimeter.

The concept of specialization in OOP means that an object can inherit the common state and behaviour of a generic object. However, each object needs to define its own special and particular state and behaviour. Circle and Rectangle each have separate formulas to calculate its area and perimeter.

Using inheritance we can manage generalization and specialization in OOP. Common features will be in the Shape class. Circle and Rectangle class will be derived from Shape class to implement shape specific features.

Abstraction is the process of identifying common patterns that have systematic variations. An abstraction represents the common patterns and provides a means for specifying which variation to use. Here the Shape class can be an Abstract class because there is no formula to calculate the area or perimeter of a generic shape. So, CalculateArea() and CalculatePerimeter() will be implemented in the derived Circle and Rectangle class.

Shape, Circle and Rectangle class:

```
public abstract class Shape{
   public abstract double CalculateArea();
   public abstract double CalculatePerimeter();
}

 public class Circle extends Shape{
   double radius;

   public Circle(){
       radius = 5;
   }

   @Override
   public double CalculateArea() {
       return 3.1416 * radius * radius;
   }

   @Override
   public double CalculatePerimeter() {
       return 2 * 3.1416 * radius;
   }
}

public class Rectangle extends Shape{
   double length, width;

   public Rectangle(){
       length = 5;
       width = 5;
   }

   @Override
   public double CalculateArea() {
       return length * width;
   }

   @Override
   public double CalculatePerimeter() {
       return 2 * ( length + width );
   }
}
```

Solution 02:

Stack:

A stack is a special area of a computer's memory which stores temporary variables created by a function. In stack, variables are declared, stored and initialized during runtime. It is a temporary storage memory. When the computing task is complete, the memory of the variable will be automatically erased. The stack section mostly contains methods, local variables and reference variables.

Heap:

The heap is a memory used by programming languages to store global variables. By default, all global variables are stored in heap memory space. It supports Dynamic Memory Allocation. The heap is not managed automatically and is not tightly managed by the CPU. It is more like a free-floating region of memory.

Heap and Stack dilemma:

If I need a large array (100MB), then I will allocate the memory on heap. Because the memory size of the heap is larger than the stack. Stack is a temporary storage memory where the data members are accessible only if the method that contains them is currently running. On windows, the typical maximum size for a stack is 1MB. Though the stack size can be adjustable. But it is ideal to keep the stack size small. So, if someone needs to allocate a large block of memory. It is ideal to keep it on the heap and it is also possible to keep the variable there for a long time which is not possible in case of stack. In stack, memory allocation and deallocation happens automatically.

Solution 03:

In element-wise matrix multiplication, multiplications are not dependent on each other. Every multiplication is independent. So, if we use threading and divide the multiplication on different threads, then multiplications will happen simultaneously on different threads. Thus, it will be faster than running the multiplications on a single thread. We can use Streams in java instead of normal for loop to parallelize the for loop.

Solution 04:

Problem of the previous solution:

The recursive solution is not correct. There are two excess count increments before the traverse methods. It will unnecessarily increase the node count.

Correct approach:

```
Struct Node{
     Node* left;
     Node* right;
}

int traverse(Node* node){
     if(node == NULL) return 0;

     count++;

     traverse(node->left);
     traverse(node->right);
     return count;
}
```

The time-complexity of the above method is O(n). So, the above approach will also work if the tree has millions of nodes and is heavily left-skewed.