SRN | | | | | | | | | | | |

PES University, Bangalore

(Established under Karnataka Act No. 16 of 2013)

UE19CS208A

OCT 2020: INTERNAL ASSESSMENT 1 (Special Topic) B.TECH.

UE19CS208D- Object Oriented Programming with Java

Time: 1Hrs | Answer all questions in the same order | Max Marks: 30

1	a)	Explain primitive types and non-primitive types with an example	5
	Solution:	Primitive and non-primitive data types Data type Size byte 1 byte short 2 bytes int 4 bytes long 8 bytes float 4 bytes double 8 bytes boolean 1 bit char 2 bytes Non-primitive data types are called reference types because they refer to objects. The main difference between primitive and non-primitive data types are: Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String). Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot. A primitive type has always a value, while non-primitive types can be null. A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter. The size of a primitive type depends on the data type, while non-primitive types have all the same size. Examples of non-primitive types are String, array, class, and interface.	
	b)	Explain overloading of instance methods with an example	5
	Solution:	In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*. Method overloading is one of the ways that Java supports polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in	

		the call. Here is a simple example that illustrates method overloading:
		// Demonstrate method overloading.
		class OverloadDemo {
		void test() {
		System.out.println("No parameters");
		}
		// Overload test for one integer parameter.
		void test(int a) {
		System.out.println("a: " + a);
		}
		// Overload test for two integer parameters.
		void test(int a, int b) {
		System.out.println("a and b: " + a + " " + b);
		}
		// Overload test for a double parameter
		double test(double a) {
		System.out.println("double a: " + a);
		return a*a;
		}
		}
		class Overload {
		public static void main(String args[]) {
		OverloadDemo ob = new OverloadDemo();
		double result;
		// call all versions of test()
		ob.test();
		ob.test(10);
		ob.test(10, 20);
		result = ob.test(123.25);
		System.out.println("Result of ob.test(123.25): " + result);
		}
		}	

2	a)	Explain the concept of assertion with an example	5

	Solution:	import java.util.*;
		// assert:
		// check for a boolean condition
		// not enabled by default
		// run :
		// java -ea Assert1
		
		// exception : used to indicate unusual conditons
		// cannot switch off exceptions
		
		// assertion : to catch programming bugs	

SRN | | | | | | | | | | | | |

```
//                      to check pre-condition, post-condition,
//                      invariants
//                  can be enabled or disabled at runtime
public class Assert1
{
        public static void main(String[] args)
        {
                Scanner scanner = new Scanner(System.in);
                double n;
                n = scanner.nextDouble();
                // Two forms of assertion;
                //      second is considered better
                // check whether the number is positive
                // 1. assert n >= 0;
                // 2. assert n >= 0 : "number is negative";
                assert n >= 0 : "number is negative";
                System.out.println("sqrt : " + Math.sqrt(n));


        }
}
```

b) Output of the following code: when the user enters 0 (zero) as the input. 5

```
import java.util.*;
class test
{
        public static void main(String args[])
        {
                try
                {
                        fun1();
                        System.out.println("after fun1");
                }
                catch(ArithmeticException e)
                {
                        System.out.println("catch in main()");
                }
                finally
                {
                        System.out.println("finally");
                }
                System.out.println("message1");
        }
        static void fun1()
        {
                fun2();
                System.out.println("after fun2");
        }
```

```
        static void fun2()
        {
                try
                {
                        int[] a={1,2,3};
                        int n;
                        Scanner ob= new Scanner(System.in);
                        n=ob.nextInt();
                        System.out.println(a[n]);
                        int res=10/n;
                        return;
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("catch in fun2");
                }
                finally
                {
                        System.out.println("finally2");
                }
                System.out.println("message2");
        }

}
```

Solution:

```
1
finally2
catch in main()
finally
message1
```

3	a)	Write a java program to search for an element in an array of integers (linear search). Write a function for the implementation of search.	5
	Solution:		

```
import java.util.*;
public class Example3
{
        public static void main(String[] args)
        {
                int[] a = {1, 2, 3, 4, 5};
                Scanner scanner = new Scanner(System.in);
                int e = scanner.nextInt();
                int pos = findElement(a, e);
                if(pos == -1)
                {
                        System.out.println("not found");
                }
```

```
                else
                {
                        System.out.println("found at pos : " + pos);
                }

        }
        public static int findElement(int[] x, int e)
        {
                boolean found = false;
                int pos = -1;
                int i; // why here and not in the loop
                for(i = 0; ! found && i < x.length; i++)
                {
                        if(x[i] == e)
                        {
                                found = true;
                                pos = i;
                        }
                }
                return pos;
        }
}
```

b) Explain dynamic polymorphism with an example. 5

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.
- At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed
- A superclass reference variable can refer to a subclass object. This is also known as up casting. Java uses this fact to resolve calls to overridden methods at run time.

Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed. Here is an example that illustrates dynamic method dispatch:

```
public class Example2
{
```

SRN | | | | | | | | | | | | |

```
        public static void main(String[] args)
        {
                // test 1:
                A x1 = new A();
                A x2 = new B();
                x1.foo();
                x2.foo();

        }
        public static void test(A a)
        {
                System.out.println("calling foo ");
                a.foo();
        }
}
class A
{
        public void foo() { System.out.println("I am foo of A"); }
}
class B extends A
{
        public void foo() { System.out.println("I am foo of B"); }
}
```