CCEE Mock- II | J2SE & OS

Total points 26/40

I hope you all came prepared for this and going to take this test seriously. Consider this as your actual CCEE and don't fall in any of the malpractices because obviously this is for your preparation purpose only. Also, do analyse the concept where you lagged in this paper. All the best.

The respondent's email (**amolgavit158121@gmail.com**) was recorded on submission of this form.

0 of 0 points

Name *

Amol Gavit

PRN (12 Digits) *

250240320013

Centre * Dropdown

Kharghar

Questions 27 of 40 points

✓ How do objects pass messages in java ? * 1/1

- ○ No tukka : "I don't want to attempt this question"
- ○ 2. they pass messages by modifying the static variables of each other`s classes.
- ○ 1. they pass messages by modifying each other`s fields.
- ◉ 3. they pass messages by calling each other`s instance methods. ✓
- ○ 4. they pass messages by calling static methods of each other`s classes .

Evaluating the Given Choices
Modifying each other's fields (Incorrect)
Directly modifying an object's fields from another object is discouraged, as it violates encapsulation principles.

Modifying static variables of each other's classes (Incorrect)
This approach involves changing shared class-level variables, but it's not a typical method of message passing.

Calling each other's instance methods (Correct)
Objects in Java pass messages by invoking instance methods of other objects. This ensures proper encapsulation and interaction.

Calling static methods of each other's classes (Incorrect)
While static methods can be invoked, they belong to the class rather than an instance, making them unsuitable for object-to-object communication.

Understanding Interfaces in the Collection Framework
An interface in Java defines a contract that classes implementing it must follow. The Java Collection Framework provides several interfaces, allowing different types of collections to be structured efficiently.

✓ Which of these is interface in the collection framework. Select the correct answers. *

- 3. Linked List
- 4. Set ✓
- No tukka : "I don't want to attempt this question"
- 1. Array
- 2. Vector

Evaluating the Given Choices
Array (Incorrect)
Array is not an interface; it's a fundamental feature of Java used for fixed-size collections.

Vector (Incorrect)
Vector is a class, not an interface. It implements List and provides synchronized access to elements.

LinkedList (Incorrect)
LinkedList is a class, not an interface. It implements the List and Deque interfaces.

Set (Correct)
Set is an interface in the Java Collection Framework, representing a collection of unique elements. Common implementations of Set include HashSet, TreeSet, and LinkedHashSet.

Understanding substring() in Java
The substring(int beginIndex, int endIndex) method extracts a portion of a string starting from beginIndex (inclusive) to endIndex (exclusive).

✓ What will be the output : *
String x="0123456789";
System.out.println(x.substring(5,8));

- No tukka : "I don't want to attempt this question"
- 1. 567 ✓
- 2. 5678
- 3. 56789
- 4. 678

Evaluating the Given Code
java
String x = "0123456789";
System.out.println(x.substring(5,8);
x.substring(5,8) takes characters from index 5 to 7 (since endIndex is exclusive).

Looking at the string "0123456789":

Index 5 → '5'

Index 6 → '6'

Index 7 → '7'

The extracted substring is "567".

✓ Which are the valid identifiers? * 1/1

a)_class
b)$value$
c)zer@
d)angstrom

- ○ No tukka : "I don't want to attempt this question"
- ○ 4. a,b,c,d,e
- ○ 2. a,b,c
- ◉ 1. a,b,d ✓
- ○ 3. c,b,d

Rules for Valid Identifiers in Java
Identifiers can contain letters (A-Z, a-z), digits (0-9), underscore (_), and dollar sign ($).

Identifiers cannot contain special characters like @, #, !, etc.

Identifiers cannot start with a digit.

Identifiers cannot be Java reserved keywords.

Evaluating the Given Choices
_class (Valid) → Allowed because it starts with an underscore.

$value$ (Valid) → Allowed because $ is a valid character.

zer@ (Invalid) → Contains @, which is not allowed.

angstrom (Valid) → A simple word with only letters, making it valid.

✓ Which of the following statements is true about exception handling in Java: *1/1

- ( ) 3.A try block must have one finally block for each catch block
- (•) 1.A try block can have many catch blocks but only one finally block ✓
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 2.A try block can have many catch blocks and many finally blocks
- ( ) 4.A try block must have at least one catch block to have a finally block

Understanding Exception Handling in Java
In Java, exception handling is managed using three key blocks:

try block: Contains code that may throw an exception.

catch block(s): Handles exceptions thrown in the try block.

finally block: Executes regardless of whether an exception occurs.

Evaluating the Given Statements
A try block can have many catch blocks but only one finally block (Correct)
A try block can have multiple catch blocks to handle different exception types.
However, it can only have one finally block, which executes after try and catch.

A try block can have many catch blocks and many finally blocks (Incorrect)
Java does not allow multiple finally blocks for a single try block.

A try block must have one finally block for each catch block (Incorrect)
The finally block is not required for every catch block. A single finally block suffices.

A try block must have at least one catch block to have a finally block (Incorrect)
A finally block can exist without a catch block.
Example:
java

```
try {
    int x = 10 / 0;
} finally {
    System.out.println("Cleanup code");
}
```

This is valid and ensures cleanup operations execute.

Understanding the Collection Interface
The Collection interface is the root interface of the Java Collections Framework. It provides fundamental methods for handling collections, such as adding, removing, and iterating over elements.

✕ Which of these methods are defined in the collection interface *

a.add(Object o)
b.retainAll(Collection c)
c.get(int index)
d.iterator()
e.indexOf(Object o)

- 2. e,c,d
- 4. a,b,c
- No tukka : "I don't want to attempt this question"
- 3. a,b,d
- (●) 1. a,c,d

Correct answer

- (●) 3. a,b,d

Evaluating the Given Methods
add(Object o) (Valid) → Defined in Collection, used to add an element.

retainAll(Collection c) (Valid) → Defined in Collection, keeps only elements present in another collection.

get(int index) (Invalid) → Not part of Collection; it belongs to List, which allows indexed access.

iterator() (Valid) → Defined in Collection, returns an iterator for traversing elements.

indexOf(Object o) (Invalid) → Not part of Collection; it belongs to List, which supports indexed searches.

Understanding Interface Extension in Java
Unlike classes, which can only extend one other class (single inheritance), interfaces can extend multiple interfaces.

This allows an interface to inherit methods from multiple sources, promoting flexibility in design.

✓ An interface can extend any number of other interfaces and can be extended by any number of interfaces

- 2. False
- 1. True
- No tukka : "I don't want to attempt this question"
- 3. Boht confusing hai

Example of an Interface Extending Multiple Interfaces
java
```
interface A {
    void methodA();
}

interface B {
    void methodB();
}

// Interface C extends both A and B
interface C extends A, B {
    void methodC();
}
```
Here, C inherits methods from both A and B, demonstrating multiple inheritance for interfaces.

Understanding Static Inner Classes in Java
A static inner class (also called a static nested class) is a class defined inside another class but declared as static.

Unlike regular inner classes, a static inner class does not require an instance of the outer class to be created.

✓ Static inner class object can be created inside its outer class. *

- 1. True
- No tukka : "I don't want to attempt this question"
- 2. False

Example of Creating a Static Inner Class Object Inside Its Outer Class
java
```
class Outer {
    static class Inner {
        void display() {
            System.out.println("Inside static inner class");
        }
    }

    public static void main(String[] args) {
        Inner obj = new Inner(); // Creating object inside Outer class
        obj.display();
    }
}
```
Output:

Inside static inner class
Here, Inner is a static inner class, and its object is created inside Outer's main method.

Understanding Final Classes and Methods in Java
A final class in Java cannot be subclassed. This means no other class can extend it.

Since a final class cannot be inherited, its methods cannot be overridden.

Even though methods in a final class do not need to be explicitly marked as final, they behave as if they are implicitly final because overriding them is impossible.

✓ Every method of final class is implicitly final. * 1/1

- (●) 1. True ✓
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 2. False
- ( ) "Only my method is final"

Example of a Final Class
java
```
final class FinalClass {
   void display() {
      System.out.println("This method cannot be overridden.");
   }
}
```
Here, display() is not explicitly marked as final, but since FinalClass itself is final, the method cannot be overridden.

Understanding NaN in Java
NaN is a special floating-point value representing an undefined or unrepresentable number.

Comparing NaN using == always returns false, as per the IEEE 754 floating-point standard.

However, equals() method correctly identifies two NaN values as equal.

✓ The following code will print *

1/1

```
Double a=new Double(Double.NaN);
Double b=new Double(Double.NaN);

if(Double.NaN==Double.NaN)
System.out.println("True");
else
System.out.println("False");


if(a.equals(b))
System.out.println("True");
else
System.out.println("false");
```

- ( ) 1. True True
- (•) 3. false True
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 4. false false
- ( ) 2. True false

Evaluating the Given Code
java
```
Double a = new Double(Double.NaN);
Double b = new Double(Double.NaN);

if (Double.NaN == Double.NaN)
   System.out.println("True");
else
   System.out.println("False");

if (a.equals(b))
   System.out.println("True");
else
   System.out.println("false");
```

Double.NaN == Double.NaN → Returns false (because NaN is unordered).
a.equals(b) → Returns true (because equals() correctly compares object values).

✓

Understanding String Comparison in Java
== compares object references, meaning it checks if two variables point to the same memory location.

.equals() compares the actual content of the strings.

✓ Which statements about the output of following program are true. *

```
public class EqualTest
{
public static void main(String[] args)
{
String s1="YES";
String s2="YES";
if(s1==s2)
System.out.println("equal");

String s3= new String ("YES");
String s4= new String("YES");

if(s3==s4)
System.out.println("s3 eq s4");
}

}
```

- (●) 2. "equal" is printed only.
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 1. "equal" is printed, "s3 eq s4" is printed.
- ( ) 4. nothing is printed.
- ( ) 3. "s3 eq s4" is printed only.

Evaluating the Given Code
java

```
public class EqualTest {
   public static void main(String[] args) {
      String s1 = "YES";
      String s2 = "YES";
      if (s1 == s2)
         System.out.println("equal");

      String s3 = new String("YES");
      String s4 = new String("YES");
      if (s3 == s4)
         System.out.println("s3 eq s4");
   }
}
```

Step-by-Step Execution
s1 == s2 → Prints "equal"

Since both s1 and s2 are string literals, Java interns them, meaning they point to the same memory location.

s1 == s2 evaluates to true, so "equal" is printed.

s3 == s4 → Does NOT print "s3 eq s4"

s3 and s4 are created using new String("YES"), meaning they are different objects in memory.

s3 == s4 evaluates to false, so "s3 eq s4" is not printed.

Understanding the Code Execution
Boolean Condition Evaluation

b1 == true evaluates to true, so the first condition in the OR (||) operation is already satisfied.

Since OR (||) short-circuits when the first condition is true, place(true) is never executed.

Output Analysis

The if condition evaluates to true, so "Khatam !!!" is printed.

The place(true) method is never called, meaning "TATA" and "BYE BYE !!" are not printed.

✕ Given the following class :

```
public static void main(String argv[]){
boolean b1=true;
if((b1==true)||place(true)){
System.out.println("Khatam !!!");
}
}
public static boolean place(boolean location){
if(location==true){
System.out.println("TATA");
}
System.out.println("BYE BYE !!");
return true;
}
}
```

What will happen when you attempt to compile and run it?

- ( ) 1. compile time error
- ( ) 4. No Output
- ( ) No tukka : "I don't want to attempt this question"
- (•) 3. Output of TATA and BYE BYE !! followed by "Khatam !!!"
- ( ) 2. Output of "Khatam !!!"

Correct answer

* Understanding the Code Execution

```
java
public static void main(String argv[]) {
   boolean b1 = true;
   if ((b1 == true) || place(true)) {
      System.out.println("Khatam !!!");
   }
}

public static boolean place(boolean location) {
   if (location == true) {
      System.out.println("TATA");
   }
   System.out.println("BYE BYE !!");
   return true;
}
```

Key Observations
Short-Circuit Behavior (|| OR Operator)
In the if condition:

```
java
if ((b1 == true) || place(true))
```

The first condition, b1 == true, evaluates to true.

Since OR (||) stops evaluating as soon as it finds true, place(true) never executes.

Impact on Output
Since place(true) is not executed, "TATA" and "BYE BYE !!" will not be printed.
The program directly prints "Khatam !!!".

Final Output
Khatam !!!

2. Output of "Khatam !!!"

What is wrong in the following class definitions? * 1/1

```
abstract class Print{abstract show( );}
class Display extends Print{}
```

- [x] 2. Wrong. Method show( ) should have a return type
- [ ] 4. Wrong. Display does not contain any members
- [ ] No tukka : "I don't want to attempt this question"
- [ ] 3. Wrong. Method show( ) is not implemented in Display
- [ ] 1. Nothing is wrong

Understanding Abstract Classes and Methods in Java
An abstract class can contain both abstract and non-abstract methods.

An abstract method must be implemented by any concrete subclass unless the subclass itself is declared abstract.

Every method in Java must have a return type, even abstract methods.

Errors in the Given Code
java
```
abstract class Print {
   abstract show(); //      Incorrect: Missing return type
}
```

class Display extends Print {} // Incorrect: Does not implement `show()`
Error in show() method

The method lacks a return type, which is mandatory in Java.

Correct declaration:

java
```
abstract void show();
```

Evaluating the Given Answer Choices
Option 2 (Correct): "Wrong. Method show() should have a return type" → This is a valid issue.

Option 3 (Correct): "Wrong. Method show() is not implemented in Display" → Since Display is a concrete class, it must implement show().

Option 4 (Incorrect): "Wrong. Display does not contain any members" → This is not necessarily an error; a class can exist without members.

Option 1 (Incorrect): "Nothing is wrong" → The code has errors.

Understanding Serialization in Java
Serialization is the process of converting an object into a byte stream so it can be saved or transmitted. However, not all fields are serialized—some modifiers prevent serialization.

✕ Class fields with following modifiers will not be serialized * 0/1

- ○ 2. static
- ○ No tukka : "I don't want to attempt this question"
- ○ 1. private
- ○ 4. protected
- ◉ 3. finaly ✕

Correct answer

- ◉ 2. static

Evaluating the Given Modifiers
static (Not Serialized)

static fields belong to the class, not individual objects.

Since serialization works on object state, static fields are ignored during serialization.

private (Serialized)

private fields can be serialized if the class implements Serializable.

protected (Serialized)

protected fields can be serialized just like private fields.

finaly (Invalid Modifier)

There is no finaly modifier in Java.

If the intended modifier was final, final fields can be serialized unless marked transient.

"If no accessibility modifier is specified for a member declaration, the member is only accessible for classes in the package of its class and subclasses of its class anywhere."

Incorrect: If no modifier is specified, the member has package-private access, meaning it is accessible only within the same package, but not necessarily in subclasses outside the package.

"You cannot specify accessibility of local variables. They are only accessible within the block in which they are declared."

Correct: Local variables do not have access modifiers like public, private, or protected. They are only accessible within the block where they are defined

✕ Which statements are true about the use of modifiers?

1.If no accessibility modifier is specified for a member declaration the member is only accessible for classes in the package of its class and subclasses of its class anywhere.

2.You cannot specify acessibility of local variable. They are only accessible within the block in which they are declared.

3.subclasses of a class must reside in the same package as the calss they extend.

4 Local variables can be decalred static.

5. Object themselves do not have any accessibilty modifiers,only the object references do.

- ○ 3. 3,4,5
- ○ No tukka : "I don't want to attempt this question"
- ◉ 2. 2,4,5
- ○ 4. 2,5
- ○ 1. 1,2,3

Correct answer

"Subclasses of a class must reside in the same package as the class they extend."

Incorrect: Subclasses can exist in different packages, but they must follow access control rules. If a superclass has default (package-private) access, it can only be extended within the same package.

"Local variables can be declared static."

Incorrect: Local variables cannot be static because static applies to class-level members, not variables inside methods.

"Objects themselves do not have any accessibility modifiers, only the object references do."

Correct: Access modifiers apply to class members (fields, methods, constructors), not objects themselves. The accessibility of an object depends on the reference used to access it

4. 2,5

Step-by-Step Execution
Compilation Check

The program compiles successfully because RuntimeException is an unchecked exception, meaning it does not require explicit handling.

There is no syntax error, except for the typo in satatic (which should be static).

Runtime Behavior

re is assigned null, meaning it does not reference a valid exception object.

When throw re; executes, Java attempts to throw null, which results in a NullPointerException.

✗ What will be result of attempting to compile & run following program * 0/1

```
public class Myclass
{
public satatic void main(String [] args)
{RuntimeException re=null;
throw re;
}
}
```

select correct answer

- 4. program will complile without error & will run & terminate without any o/p ✗
- 3. program will complile without error & will throw java.lang.NullpointerException when run
- No tukka : "I don't want to attempt this question"
- 1. program fail to compile ,since it cannot throw re
- 2. program will complile without error & will throw java.lang.RuntimeException when run

Final Output
Exception in thread "main"
java.lang.NullPointerException
at Myclass.main(Myclass.java:4)
Correct Answer:
Option 3: The program will compile without error and will throw java.lang.NullPointerException when run.

Correct answer

- 3. program will complile without error & will throw java.lang.NullpointerException when run

Understanding wait() in Java
The wait() method is part of the Object class and is used for inter-thread communication.

When a thread calls wait() on an object, it releases the object's monitor lock and enters a waiting state.

The thread remains in this state until another thread calls notify() or notifyAll() on the same object.

✓ What is the effect of issuing a wait() method on an object? * 1/1

- 3. An exception will be raised
- (●) 2. The object issuing the call to wait() will halt until another object sends a notify() or notifyAll() method ✓
- 4. The object issuing the call to wait() will be automatically synchronized with any other objects using the receiving object.
- No tukka : "I don't want to attempt this question"
- 1. If a notify() method has already been sent to that object then it has no effect

Evaluating the Given Options
"If a notify() method has already been sent to that object then it has no effect" (Incorrect)

If notify() was called before wait(), the thread will still go into waiting mode because notify() does not store a signal for future use.

"The object issuing the call to wait() will halt until another object sends a notify() or notifyAll() method" (Correct)

This is the expected behavior of wait(). The thread pauses execution until it is notified.

"An exception will be raised" (Partially Incorrect)

wait() can throw an InterruptedException if the waiting thread is interrupted, but this is not the default behavior.

"The object issuing the call to wait() will be automatically synchronized with any other objects using the receiving object" (Incorrect)

wait() only affects the thread calling it; it does not synchronize other objects automatically.

Evaluating the Given Options
"All static variables, all final variables, all instance variables" (Correct)
Inner classes can access all types of variables from the outer class.

"Only final static variables" (Incorrect)
Inner classes can access non-final variables too.

"Only final instance variables" (Incorrect)
Inner classes can access non-final instance variables as well.

"Only public variables" (Incorrect)
Inner classes can access private variables too.

✓ Which variables can an inner class access from the class which encapsulates it? *1/1

- 2. Only final static variables
- (●) 1. All static variables,all final variable ,all instance variable. ✓
- 4. only public variable
- No tukka : "I don't want to attempt this question"
- 3. Only final instance variables

✕ what will be the output of the following code? * 0/1

```
public class TestClass
{
public static void main(String args[])
{
if(0.0==-0.0)
{
System.out.println("equal");
}
else
System.out.println("unequal");
}
}
```

Understanding Floating-Point Comparison in Java
Java follows the IEEE 754 floating-point standard, which specifies that 0.0 and -0.0 are considered equal when using the == operator.

Although -0.0 has a negative sign, it is numerically equivalent to 0.0 in comparisons.

0.0 == -0.0 evaluates to true, as per IEEE 754 rules.

The program prints "equal".

- (●) 2. unequal ✕
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 1. equal
- ( ) 3. compile time error
- ( ) 4. none of these

Correct answer

- (●) 1. equal

Understanding finally Execution in Java
Normally, the finally block always executes, regardless of whether an exception occurs.

However, if System.exit(0) is called, the JVM terminates immediately, preventing any further execution—including the finally block.

✗ In following program 'finally' will execute ? * 0/1

```
try
{
int a=5;
int b=0;
int c=a/b;
System.exit(0);
}
catch(Exception e)
{
System.out.println(e);
}
finally
{
System.out.println("In Finally");
}
```

int c = a / b; causes an ArithmeticException (division by zero).

System.exit(0); is never reached because the exception occurs first.

The JVM terminates immediately when System.exit(0) is called, skipping the finally block.

Final Behavior
The program terminates before reaching finally, so "In Finally" is not printed.

Answer is correct

- ○ 1. True
- ○ No tukka : "I don't want to attempt this question"
- ◉ 2. False ✗
- ○ 3. Finally ki marzi

Correct answer

- ◉ 1. True

Step-by-Step Execution
Division by Zero (a / b)

Since b = 0, attempting a / b results in an ArithmeticException.

The exception is caught in the catch block, printing "Exception ".

Execution of finally Block

The finally block always executes, regardless of whether an exception occurs.

"Finally" is printed after "Exception ".

✓ What is the result of executing the following code, using the parameters 4 and 0: *1/1

```
public void divide(int a, int b) {

try {

int c = a / b;

} catch (Exception e) {

System.out.print("Exception ");

} finally {

System.out.println("Finally");

}
```

- ○ 3. Prints out: Exception
- ○ 4. No output
- ○ No tukka : "I don't want to attempt this question"
- ◉ 1. Prints out: Exception Finally ✓
- ○ 2. Prints out: Finally

✓ Which methods can be legally applied to a string object? * 1/1

a)equals(String)
b)equals(Object)
c)trim()
d)round()
e)toString()

- ( ) No tukka : "I don't want to attempt this question"
- (•) 4. a,b,c,e
- ( ) 2. a,e
- ( ) 3. a,b,d,e
- ( ) 1. a,b

Evaluating the Given Methods
equals(String) (Incorrect)

The equals() method does not take a String parameter specifically. Instead, it takes an Object and checks if the given object is equal to the current string.

equals(Object) (

The correct method signature is boolean equals(Object obj), which compares the content of two strings.

trim() (

The trim() method removes leading and trailing whitespace from a string.

round() (Incorrect)

round() is not a method of String. It belongs to the Math class and is used for rounding numbers.

toString() (

The toString() method returns the string representation of the object.

Understanding String Comparison in Java
The equals() method in Java compares the actual content of two strings, not their memory references.

str1 is a string literal, meaning it is stored in the string pool.

str2 is created using new String("Hello"), meaning it is stored separately in heap memory.

Despite being stored differently, both contain the same sequence of characters, so str1.equals(str2) returns true.

✓ class test * 1/1

```
{
public static void main(String[] args)
{
String str1="Hello";
String str2=new String("Hello");
if(str1.equals(str2))
System.out.println("Equal");
else
System.out.println("Not Equal");
}
}
```

- 4. Run time error
- 2. Not Equal
- No tukka : "I don't want to attempt this question"
- 3. compile time error
- (●) 1. Equal ✓

✗ class outer * 0/1

```
{
private int i=10;
public void test()
{
inner obj=new inner();
System.out.println("j="+j);
}
class inner
{
private int j=20;
}
}

public class demo
{
public static void main(String[] args)
{
outer ob=new outer();
ob.test();
}
}
```

Understanding the Issue
Inner Class Access Rules

The inner class inner is defined inside outer, meaning it has access to the private members of outer.

However, the outer class does not have direct access to the members of the inner class.

Error in test() Method

java
public void test() {
inner obj = new inner();
System.out.println("j=" + j); // Compilation Error
}
The variable j belongs to the inner class, but it is being accessed directly inside the outer class.

Since j is not a member of outer, the compiler cannot resolve the symbol j.

- 4. None of the Above
- 3. Compile successfully and print 0
- No tukka : "I don't want to attempt this question"
- 1. Compiler Error : Cannot resolve symbol j

Correct Answer:
Option 1: Compiler Error - Cannot resolve symbol j

2. Compile successfully and print 20 ✗

Correct answer

1. Compiler Error : Cannot resolve symbol j

✗ Which of the following will produce a value of 22 if x = 22.9? * 0/1

No tukka : "I don't want to attempt this question"

1. ceil(x)

4. floor(x)

2. round(x)

3. abs(x)

Correct answer

4. floor(x)

Understanding Rounding Methods in Java
Java provides several methods for rounding numbers, primarily through the Math class:

ceil(x) (Incorrect)
Math.ceil(x) rounds up to the nearest integer.
Example: Math.ceil(22.9) → 23 (not 22).

floor(x) (Correct)
Math.floor(x) rounds down to the nearest integer.
Example: Math.floor(22.9) → 22.

round(x) (Incorrect)
Math.round(x) rounds to the nearest integer based on standard rounding rules.
Example: Math.round(22.9) → 23 (since .9 rounds up).

abs(x) (Incorrect)
Math.abs(x) returns the absolute value, meaning it removes the negative sign but does not round.
Example: Math.abs(-22.9) → 22.9 (not 22).

Understanding CPU Scheduling
CPU scheduling is a fundamental concept in operating systems that determines which process gets to use the CPU at a given time. It is essential for multiprogramming, where multiple processes share CPU time efficiently.

✓ Which of the following options is the correct statement regarding CPU Scheduling? *1/1

- ( ) 2. CPU Scheduling is the basis of Mono-programming.
- ( ) 4. CPU Scheduling is the basis of Real-time OS.
- ( ) 3. CPU Scheduling is the basis of Batch OS.
- ( ) No tukka : "I don't want to attempt this question"
- (•) 1. CPU Scheduling is the basis of Multiprogramming. ✓

Evaluating the Given Options
CPU Scheduling is the basis of Multiprogramming (Correct)
Multiprogramming allows multiple processes to be loaded into memory and executed concurrently.
CPU scheduling ensures that processes take turns using the CPU, maximizing utilization.

CPU Scheduling is the basis of Mono-programming (Incorrect)
Mono-programming refers to executing one program at a time, which does not require CPU scheduling.

CPU Scheduling is the basis of Batch OS (Incorrect)
Batch OS processes jobs sequentially, often without requiring complex CPU scheduling.

CPU Scheduling is the basis of Real-time OS (Partially Correct but Not the Best Answer)
Real-time OS uses specialized scheduling algorithms to meet strict timing constraints, but CPU scheduling itself is not its defining characteristic.

Understanding Time-Sharing OS Scheduling
A time-sharing operating system allows multiple users to share CPU time efficiently.

The key requirement is fairness and quick response time, ensuring that no process monopolizes the CPU.

✓ Which scheduling algorithm is considered the most optimal choice for achieving efficient resource utilization in a time-sharing operating system? *1/1

A. First come first serve (FCFS)

B. Round robin (RR)

C. Priority scheduling

D. Shortest job next (SJN)

E. Multilevel queue

Correct answer is ?

- 4. All
- 1. Only B
- 3. A,B,C,D
- 2. Both B and E
- No tukka : "I don't want to attempt this question"

Evaluating the Given Scheduling Algorithms
First Come First Serve (FCFS) (Not Optimal)
Processes are scheduled in the order they arrive.
Issue: Long-running processes can block shorter ones, leading to poor response time.

Round Robin (RR) (Optimal Choice)
Each process gets a fixed time quantum before moving to the next.
Advantage: Ensures fair CPU distribution and prevents starvation.
Best suited for time-sharing systems.

Priority Scheduling (Not Ideal for Time-Sharing)
Processes are scheduled based on priority.
Issue: Lower-priority processes may suffer starvation.

Shortest Job Next (SJN) (Not Ideal for Time-Sharing)
The process with the shortest execution time is scheduled first.
Issue: Requires knowing job lengths in advance, which is impractical.

Multilevel Queue (Can Be Used Alongside RR)
Divides processes into multiple queues based on priority.
Advantage: Can be combined with Round Robin for better efficiency.

✓ Which condition leads to the occurrence of a page fault in a computer system? *1/1

- ( ) No tukka : "I don't want to attempt this question"
- ( ) 3. The page is currently stored in the main memory
- (•) 4. The page is not currently present in the main memory ✓
- ( ) 2. An attempt is made to perform an illegal mathematical operation
- ( ) 1. The page is intentionally modified by the application software

Understanding Page Faults in a Computer System
A page fault occurs when a program tries to access a page that is not currently loaded in the main memory (RAM). This triggers the operating system to fetch the required page from secondary storage (such as a hard disk or SSD) into RAM

Evaluating the Given Options
"The page is intentionally modified by the application software" (Incorrect)

Modifying a page does not cause a page fault. Page faults occur due to missing pages, not modifications.

"An attempt is made to perform an illegal mathematical operation" (Incorrect)

Illegal operations (like division by zero) cause exceptions, not page faults.

"The page is currently stored in the main memory" (Incorrect)

If the page is already in memory, no page fault occurs.

"The page is not currently present in the main memory" (Correct)

This is the exact definition of a page fault. The system must retrieve the missing page from disk

✕ Among the listed resources, which ones have the potential to contribute to the occurrence of deadlocks in a computer system? *0/1

- ○ No tukka : "I don't want to attempt this question"
- ◉ 3. Printers that are used for document output ✕
- ○ 4. All of the above resources can lead to deadlocks
- ○ 1. Files that can only be read and cannot be modified
- ○ 2. Programs that are shared and executed by multiple users simultaneously

Correct answer

- ◉ 2. Programs that are shared and executed by multiple users simultaneously

Understanding Deadlocks in a Computer System
A deadlock occurs when multiple processes are waiting for resources held by each other, preventing further execution. Deadlocks typically arise due to four necessary conditions:

Mutual Exclusion – A resource can only be used by one process at a time.

Hold and Wait – A process holds at least one resource while waiting for another.

No Preemption – A resource cannot be forcibly taken away from a process.

Circular Wait – A set of processes are waiting for resources in a circular chain.

Evaluating the Given Resources
Files that can only be read and cannot be modified (Can contribute to deadlocks)

If multiple processes need exclusive access to a file, they may block each other.

Programs that are shared and executed by multiple users simultaneously (Can contribute to deadlocks)

Shared programs may require synchronization mechanisms, leading to deadlocks if improperly managed.

Printers that are used for document output (Can contribute to deadlocks)

If multiple processes request exclusive access to a printer, they may enter a deadlock state.

Understanding Compaction in Memory Management
External fragmentation occurs when free memory is available but scattered in small, non-contiguous blocks.

This prevents large processes from being allocated memory, even though there is enough total free space.

✕ Which problem in memory management does compaction aim to address? *0/1

- ○ 2. Insufficient disk space
- ○ No tukka : "I don't want to attempt this question"
- ◉ 4. Internal fragmentation ✕
- ○ 1. Excessive usage of CPU resources
- ○ 3. External fragmentation

Correct answer

- ◉ 3. External fragmentation

Evaluating the Given Options
Insufficient disk space (Incorrect)
Compaction deals with RAM fragmentation, not disk space issues.

Excessive usage of CPU resources (Incorrect)
Compaction can cause CPU overhead, but its primary goal is memory optimization.

External fragmentation (Correct)
Compaction reduces external fragmentation by consolidating free memory.

Internal fragmentation (Incorrect)
Internal fragmentation occurs when allocated memory blocks are larger than needed, wasting space inside allocated regions.
Compaction does not address this issue.

Understanding Deadlock Likelihood in a System with 3 Processes and 4 Shared Resources
Deadlock occurs when multiple processes hold resources and wait indefinitely for additional resources held by other processes. The likelihood of deadlock depends on the number of processes, available resources, and maximum resource demand per process.

✓ In a system with 3 processes and 4 shared resources, where each process requires a maximum of two units, what is the likelihood of deadlock occurrence?

- No tukka : "I don't want to attempt this question"
- (●) 3. Deadlock may or may not occur
- 4. Deadlock depends on the specific scheduling algorithm used
- 2. Deadlock is impossible to occur
- 1. Deadlock is guaranteed to occur

Evaluating the Given Scenario
Number of processes: 3
Number of shared resources: 4
Maximum resource requirement per process: 2 units

Deadlock Analysis
Worst-case scenario:
Each process requests one resource initially, leaving one free resource.
If all three processes then request one more resource, but only one resource remains, at least one process will be blocked, leading to potential deadlock.

Avoidance possibility:
If resource allocation is managed carefully (e.g., using the Banker's Algorithm or ensuring at least one free resource remains), deadlock can be avoided.

Understanding Turnaround Time in Process Execution
Turnaround time refers to the total time taken by a process from its submission to completion.

It includes both:

Processing time (actual execution time on the CPU).

Waiting time (time spent in the ready queue or waiting for resources).

✓ In the context of process execution, which term refers to the total time taken by a process to complete its execution, including both the processing time and any waiting time? *1/1

- 2. Response time
- 3. Throughput
- No tukka : "I don't want to attempt this question"
- 1. Processing time
- (●) 4. Turnaround time

Evaluating the Given Options
Processing time (Incorrect)
Processing time refers only to the time a process spends executing on the CPU. It does not include waiting time.

Response time (Incorrect)
Response time is the time between process submission and the first response. It does not measure the total execution time.

Throughput (Incorrect)
Throughput refers to the number of processes completed per unit time. It does not measure the time taken by a single process.

Turnaround time (Correct)
Turnaround time accounts for both execution and waiting time, making it the correct answer.

Understanding Time Quantum in Round Robin Scheduling
Round Robin (RR) Scheduling assigns a fixed time quantum to each process in a cyclic order.

Increasing the time quantum affects response time and context switching frequency, but does not significantly impact average turnaround time.

✓ In Round Robin CPU Scheduling, what is the effect of increasing the time quantum (also known as time slice) on the average turnaround time of processes?

- (●) 2. The average turnaround time increases
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 1. The average turnaround time decreases
- ( ) 3. The average turnaround time remains constant
- ( ) 4. There is no effect on the average turnaround time

Effects of Increasing Time Quantum
If the time quantum is too small
More frequent context switches, increasing CPU overhead.
Processes take longer to complete due to excessive switching.

If the time quantum is too large
RR behaves more like First Come First Serve (FCFS).
Processes complete in fewer switches, but waiting time increases for shorter tasks.

Impact on Turnaround Time
Turnaround time is the total time from arrival to completion.

Since all processes eventually get CPU time, turnaround time remains relatively stable regardless of time quantum adjustments.

Data Section

Contains global and static variables.

Initialized data and uninitialized data are stored separately.

Text Section

Stores executable instructions of the program.

Typically a read-only section to prevent accidental modification.

✓ Which sections can a process be divided into? * 1/1

- ( ) 1. The stack section
- ( ) 2. The heap section
- ( ) 3. The data section and text section
- (●) 4. All of the above
- ( ) No tukka : "I don't want to attempt this question"

Understanding Process Memory Layout
A process in memory is divided into several distinct sections, each serving a different purpose:

Stack Section
Stores temporary data, such as function parameters, return addresses, and local variables.
Grows and shrinks dynamically as functions are called and return.

Heap Section
Used for dynamic memory allocation during runtime.
Memory is allocated using functions like malloc() in C or new in Java.

✓ The init process which always has a Pid of ? *

- 1. 3
- No tukka : "I don't want to attempt this question"
- 3. 1
- 4. 0
- 2. 4

Understanding the Init Process
The init process is the first process started by the kernel when a Unix or Linux system boots up.

It is responsible for initializing the system, managing system services, and spawning other processes.

Since it is

PID 1 (Correct)
The init process is always PID 1 in Unix/Linux systems.

PID 0 (Incorrect)
PID 0 is reserved for the scheduler or idle process, not init.

✓ In a multiplayer online game, a player character is waiting for an opponent's move to be received over the network before making its own move. In which state is the player character during this waiting period?

- 4. The "ready" state
- 3. The "running" state
- No tukka : "I don't want to attempt this question"
- 2. The "waiting" state
- 1. The "new" state

Understanding Process States in Multiplayer Games
In a multiplayer online game, a player character must wait for an opponent's move before proceeding. This situation aligns with the waiting state in process execution.

Evaluating the Given Options
"New" state (Incorrect)
The "new" state refers to a process that has just been created but has not started execution.

"Waiting" state (Correct)
The player character is waiting for an external event (opponent's move), making this the correct state.

"Running" state (Incorrect)
The process is not actively executing; it is paused until the opponent's move arrives.

"Ready" state (Incorrect)
The "ready" state means the process is prepared to run but is waiting for CPU time, which is different from waiting for an opponent's move.

This command performs the following actions:

find . -type f -name "*.txt"

Searches for all files (-type f) in the current directory (.) with the .txt extension.

-exec grep "keyword" {} ;

Executes grep "keyword" on each found file ({} represents the filename).

If a file contains "keyword", grep prints the matching lines along with the filename.

✓ What does the following command output in a bash shell? * 1/1

$ find . -type f -name "*.txt" -exec grep "keyword" {} ;

- 2. The word "keyword"
- (●) 1. The list of files in the current directory that have the extension ".txt" and contain the word "keyword" ✓
- 4. No output, the command is invalid
- 3. The word "{}"
- No tukka : "I don't want to attempt this question"

Evaluating the Given Options

"The list of files in the current directory that have the extension .txt and contain the word keyword" (Correct)

grep searches for "keyword" in each .txt file and prints matching results.

"The word keyword" (Incorrect)

grep prints entire matching lines, not just the word "keyword".

"The word {}" (Incorrect)

{} is a placeholder for filenames in -exec, but it does not appear in the output.

"No output, the command is invalid" (Incorrect)

The command is valid and will produce output if matching files exist.

Understanding the Shell Script Execution
bash
#!/bin/bash
echo "Welcome to the script"
exit 1

#!/bin/bash
This shebang specifies that the script should be executed using the Bash shell.

echo "Welcome to the script"
This prints "Welcome to the script" to the terminal.

exit 1
The script terminates with exit code 1, which indicates an error or failure.

✕ Consider the following shell script named "script.sh": *

```
#!/bin/bash

echo "Welcome to the script"

exit 1
```

What is the output when running the script?

- (●) 1. Welcome to the script
- ( ) 3. Error: Permission denied
- ( ) No tukka : "I don't want to attempt this question"
- ( ) 4. Error: Script execution terminated with exit code 1
- ( ) 2. Error: Command not found

Correct answer

- (●) 4. Error: Script execution terminated with exit code 1

Evaluating the Given Options
"Welcome to the script" (Incorrect)

While "Welcome to the script" is printed, the script terminates with exit code 1, signaling an error.

"Error: Command not found" (Incorrect)

The script does not contain an invalid command, so this error does not occur.

"Error: Permission denied" (Incorrect)

This error would occur only if the script lacks execution permissions (chmod +x script.sh would be needed).

"Error: Script execution terminated with exit code 1" (Correct)

Since the script explicitly exits with exit 1, this is the expected behavior[_{{{CITATION{{{_1{CCEE Mock- II _ J2SE & OS (AADU).pdf](file:///E:/DAC/CCEE/Test/CCEE%20Mock-%20II%20_%20J2SE%20&%20OS%20(AADU).pdf)

Understanding the Shell Script Execution
bash
#!/bin/bash
result=$((5 + "abc"))
echo "The result is: $result"
#!/bin/bash

This shebang specifies that the script should be executed using the Bash shell.

result=$((5 + "abc"))

The ((...)) syntax is used for arithmetic operations in Bash.

"abc" is not a valid number, causing an invalid arithmetic operator error.

echo "The result is: $result"

This line will not execute because the script terminates with an error.

✗ Consider the following shell script named "script.sh": * 0/1

#!/bin/bash

result=$((5 + "abc"))

echo "The result is: $result"
What is the output when running the script?

- 1. The result is: abc
- 2. The result is: 5 ✗
- No tukka : "I don't want to attempt this question"
- 3. Error: Invalid arithmetic operator
- 4. Error: Command not found

Correct answer

- 3. Error: Invalid arithmetic operator

Evaluating the Given Options
"The result is: abc" (Incorrect)

Bash does not treat "abc" as a valid operand in arithmetic operations.

"The result is: 5" (Incorrect)

The script fails before assigning a value to result, so this output is not possible.

"Error: Invalid arithmetic operator" (Correct)

Bash throws an error when attempting to perform arithmetic with a non-numeric value.

"Error: Command not found" (Incorrect)

The script does not contain an invalid command, only an arithmetic error.

Calculating Average Turnaround Time Using FCFS Scheduling
First-Come, First-Served (FCFS) scheduling executes processes in the order they arrive. The turnaround time for each process is calculated as:

Turnaround Time
=
Completion Time
–
Arrival Time

✓ Consider the following four processes with their corresponding arrival time and burst time : *1/1

Process	Arrival time	Burst time(in ms)
P1	0.0	8
P2	0.6	6
P3	3.8	4
P4	4.4	2

What is the average turnaround time (in ms) for these processes using the FCFS scheduling algorithm?

- ○ 4. None of these
- ◉ 2. 12.8 ✓
- ○ 1. 15
- ○ 3. 13
- ○ No tukka : "I don't want to attempt this question"

Feedback 0 of 0 points

Difficulty level of mock * Dropdown

Moderate

From now onwards, I will give my best in everything in my life without any excuses. Because I know, problem is the part and parcel of life. We should always look for solutions. *

I PROMISE

How was your experience ?