05 Java Constructors, Methods, & Encapsulation

Problema 1

Given:

```
class Triangle{
    public int base;
    public int height;
    private final double ANGLE;

    public  void setAngle(double a){  ANGLE = a;  }

    public static void main(String[] args) {
        Triangle t = new Triangle();
        t.setAngle(90);
    }
}
```

Identify the correct statement(s).
You have to select 1 option

- the value of ANGLE will not be set to 90 by the setAngle method.
- An exception will be thrown at run time.
- The code will work as expected setting the value of ANGLE to 90.
- The code will not compile.

Explanation:
The given code has two problems: 1. If you declare a field to be final, it must be explicitly initialized by the time the creation of an object of the class is complete. So you can either initialize it immediately: private final double ANGLE = 0; or you can initialize it in the constructor or an instance block. 2. Since ANGLE is final, you can't change its value once it is set. Therefore the setAngle method will also not compile.

Problema 2

Given the following source code, which of the lines that are commented out may be reinserted without introducing errors?

```
abstract class Bang{
 //abstract void f();  //(0)
   final    void g(){}
 //final    void h(){} //(1)
   protected static int i;
   private int j;
}

final class BigBang extends Bang{
 //BigBang(int n) { m = n; } //(2)
   public static void main(String args[]){
      Bang mc = new BigBang();
   }
   void h(){}
 //void k(){ i++; } //(3)
 //void l(){ j++; } //(4)
   int m;
}
```

You have to select 1 option:

`final void h( ) { } //(1)`
It will fail because BigBang will try to override a final method.

`BigBang(int n) { m = n; }  //(2)`
It will fail since BigBang will no longer have a default constructor that is used in the main( ) method.

`void k( ) { i++; }  //(3)`

`void l( ) { j++; }  //(4)`
It will fail since the method will try to access a private member 'j' of the superclass.

`abstract void f( ) ;  //(0)`
If this line is inserted, then either the class BigBang will have to be declared abstract or it has to implement method f().

Explanation:
Default constructor (having no arguments) is automatically created only if the class does not define any constructors. So as soon as //2 is inserted the default constructor will not be created.

Problema 3

What will the following code print when run?

```
public class Noobs {
    public void m(int a){
        System.out.println("In int ");
    }
    public void m(char c){
        System.out.println("In char ");
    }
    public static void main(String[] args) {
        Noobs n = new Noobs();
        int a = 'a';
        char c = 6;
        n.m(a);
        n.m(c);
    }
}
```

You have to select 1 option:

- In int
  In char

- In char
  In int

- In int
  In int

- In char
  In char

- It will not compile.

Explanation:
It looks confusing but it is a simple question. Remember that whenever two methods are applicable for a method call, the one that is most specific to the argument is chosen.
In case of `m(a)`, a is an `int`, which cannot be passed as a char (because an int cannot fit into a char). Therefore, only `m(int)` is applicable.
In case of `m(c)`, c is a `char`, which can be passed as an int as well as a char. Therefore, both the methods are applicable. However, `m(char)` is most specific therefore that is chosen over m(`int`).

Problema 4

What should be placed in the two blanks so that the following code will compile without errors:

```
class XXX{
    public void m() {
        throw new RuntimeException();
    }
}
class YYY extends XXX{
    public void m() throws Exception{
      throw new Exception();
    }
}
public class TestClass {
    public static void main(String[] args) {
        ______ obj = new ______();
        obj.m();
    }
}
```

You have to select 1 option

- XXX and YYY
- XXX and XXX
- YYY and YYY
- YYY and XXX
- None of the options will make the code compile.
    - Remember that an overriding method can only throw a subset of checked exceptions declared in the throws clause of the overridden method. Here, method m in XXX does not declare any checked exception in its throws clause, therefore, method m in YYY cannot declare any checked exception in its throws clause either. Class YYY will, therefore, not compile.

Problema 5

Consider the following code:

```
public class MyClass {

  protected int value = 10;

}
```

Which of the following statements are correct regarding the field value?
You have to select 1 option:

- It cannot be accessed from any other class.
- It can be read but cannot be modified from any other class.
- It can be modified but only from a subclass of MyClass.
    - It can also be modified from any class defined in the same package.
- It can be read and modified from any class within the same package.
    - Note that since value is protected, a class in another package which extends MyClass will only inherit this variable, but it cannot read or modify the value of a variable of a MyClass instance. For example:
    - `//in different package`
    - `class X extends MyClass{`
    - `    public static void main(String[] args){`
    - `        int a = new MyClass().value; //This will not compile`
    - `//because X does not own MyClass's value.`
    - `        a = new X().value; //This will compile fine because X inherits value.`
    - `    }`
    - `}`

Pregunta 6

What should be the return type of the following method?

```
public RETURNTYPE methodX( byte by){
    double d = 10.0;
    return (long) by/d*3;
}
```

You have to select 1 option:

- int
- long
- double
- float
- byte

Explanattion:
Note that the cast (long) applies to 'by' not to the whole expression.

```
( (long) by ) / d * 3;
```

Now, division operation on long gives you a double. So the return type should be double.

Problema 7

What would be the result of attempting to compile and run the following program?

```
class TestClass{
   static TestClass ref;
   String[] arguments;
   public static void main(String args[]){
      ref = new TestClass();
      ref.func(args);
   }
   public void func(String[] args){
      ref.arguments = args;
   }
}
```

You had to select 1 option

- The program will fail to compile, since the static method main is trying to call the non-static method `func`.
    - The concept here is that a non-static method (i.e. an instance method) can only be called on an instance of that class. Whether the caller itself is a static method or not, is immaterial.
    - The main method is calling ref.func(); - this means the main method is calling a non-static method on an actual instance of the class TestClass (referred to by 'ref'). Hence, it is valid. It is not trying calling it directly such as func() or this.func(), in which case, it would have been invalid.

- The program will fail to compile, since the non-static method func cannot access the static member variable `ref`.
    - Non static methods can access static as well as non static methods of a class.

- The program will fail to compile, since the argument `args` passed to the static method `main` cannot be passed on to the non-static method `func`.
- It certainly can be.

- The program will fail to compile, since method `func` is trying to assign to the non-static member variable `'arguments'` through the static member variable ref.

- The program will compile and run successfully.

Problema 8

Identify correct option(s)
You had to select 2 options

- Multiple inheritance of state includes ability to inherit instance methods from multiple classes.
    - Methods do not have state. Ability to inherit instance methods from multiple classes is called multiple inheritance of implementation. Default methods introduce one form of multiple inheritance of implementation. A class can implement more than one interface, which can contain default methods that have the same name. However, such a class cannot be compiled. In this case, the implementing class is required to provide its own implementation of the common method to avoid ambiguity.

- Multiple inheritance of state includes ability to inherit instance fields from multiple classes.

- Multiple inheritance of type includes ability to inherit instance fields as well as instance methods from multiple classes.

- Multiple inheritance of type includes ability to implement multiple interfaces and ability to inherit static or instance fields from interfaces and/or classes.

- Multiple inheritance of type includes ability to implement multiple interfaces and/or ability to extend from multiple classes.

Explanation:
Interfaces, classes, and enums are all "types". Java allows a class to implement multiple interfaces. In this way, Java supports multiple inheritance of types.
"State", on the other hand, is represented by instance fields. Only a class can have instance fields and therefore, only a class can have a state. (Fields defined in an interface are always implicitly static, even if you don't specify the keyword static explicitly. Therefore, an interface does not have any state.) Since a class is allowed to extend only from one class at the most, it can inherit only one state. Thus, Java does not support multiple inheritance of state.

Problema 9

Given the following code, which method declarations can be inserted at line 1 without any problems?

```
public class OverloadTest{
    public int sum(int i1, int i2) { return i1 + i2; }
    // 1
}
```

You had to select 3 options

- `public int sum(int a, int b) { return a + b; }`
    - Will cause duplicate method. Variable names don't matter. Only their types.

- `public int sum(long i1, long i2) { return (int) i1; }`

- `public int sum(int i1, long i2) { return (int) i2; }`

- `public long sum(long i1, int i2) { return i1 + i2; }`

- `public long sum(int i1, int i2) { return i1 + i2; }`
    - Only the return type is different so the compiler will complain about having duplicate method sum.

Explanation:

The rule is that you cannot have methods that create ambiguity for the compiler in a class. It is illegal for a class to have two methods having same name and having same type of input parameters in the same order.
Name of the input variables and return type of the method are not looked into.
1. Option 1 is wrong because, then both the methods will be same (as their method name and the class/type and order of the input parameters will be same). So this amounts to duplicate method which is not allowed. As mentioned, name of the input parameters does not matter. Only the type of parameters and their order matters.
2. 2 is valid because the type of input parameters are different. So this is a different method and does not amount to duplication.
3 and 4 are valid for the same reason
5 is not valid because it leads to duplicate method(as their method name and the class/type and order of the input parameters will be same). Note that as mentioned in the comments, return type does not matter.

Problema 10

Which of the following methods does not return any value?
You had to select 1 option

- ```
  public doStuff() throws FileNotFoundException, IllegalArgumentException{
      //valid code not shown
  }
  ```
  - It is missing the return type. Every method must have a return type specified in its declaration. It could be a valid constructor though if the class is named doStuff because the constructors don't return anything, not even void.

- ```
  public null doStuff() throws FileNotFoundException, IllegalArgumentException{
      //valid code not shown
  }
  ```
  - null can be a return value not a return type because null is not a type.

- ```
  public doStuff() {
      //valid code not shown
  }
  ```
  - This is not a valid method because there is no return type declared. Although it can be a valid constructor if the name of the class is doStuff.

- ```
  public void doStuff() throws FileNotFoundException, IllegalArgumentException{
      //valid code not shown
  }
  ```
  - A method that does not return anything should declare its return type as void. Note that this is different from constructors. A constructor also doesn't return anything but for a constructor, you don't specify any return type at all. That is how a constructor is differentiated from a regular method.

- ```
  private doStuff() {
      //valid code not shown
  }
  ```
  - This is not a valid method because there is no return type declared. Although it can be a valid constructor if the name of the class is doStuff.

Problema 11

Consider the following code:

```
import java.util.ArrayList;

public class Student{

    ArrayList<Integer> scores;
    private double average;

    public ArrayList<Integer> getScores(){ return scores; }

    public double getAverage(){ return average; }

    private void computeAverage(){
        //valid code to compute average
        average =//update average value
    }

    public Student(){
        computeAverage();
    }

    //other code irrelavant to this question not shown

}
```

What can be done to improve the encapsulation of this class?
You had to select 2 options

- Make the class private.
- Make the scores instance field private.
  - An important aspect of encapsulation is that other classes should not be able to modify the state fields of a class directly. Therefore, the data members should be private (or protected if you want to allow subclasses to inherit the field) and if the class wants to allow access to these fields, it should provide appropriate setters and getters with public access.
- Make getScores() protected.
- Make computeAverage() public.
- Change getScores to return a copy of the scores list:
  ```
  public ArrayList<Integer> getScores(){
      return new ArrayList(scores);
  }
  ```
  - If you return the same scores list, the caller would be able to add or remove elements from it, thereby rendering the average incorrect. This can be prevented by returning a copy of the list.
  ```

Problema 12

What will be the result of attempting to compile the following program?

```
public class TestClass{
   long l1;
   public void TestClass(long pLong) { l1 = pLong ; }  //(1)
   public static void main(String args[]){
      TestClass a, b ;
      a = new TestClass();  //(2)
      b = new TestClass(5);  //(3)
   }
}
```

You had to select 1 option

- A compilation error will be encountered at (1), since constructors should not specify a return value.
  - But it becomes a valid method if you give a return type.
- A compilation error will be encountered at (2), since the class does not have a default constructor.
  - The class has an implicit default constructor since the class doesn't have any constructor defined.
- A compilation error will be encountered at (3).
  - Because (1) is a method and not a constructor. So there is no constructor that take a parameter.
- The program will compile correctly.
- It will not compile because parameter type of the constructor is different than the type of value passed to it.
  - If (1) was a valid constructor 'int' would be promoted to long at the time of passing.

Explanation:
// Yopienso que la idea es engañar que es un constructor, peor al tener void, es un metodo

The declaration at (1) declares a method, not a constructor because it has a return value.
The method happens to have the same name as the class, but that is ok.

The class has an implicit default constructor since the class contains no constructor declarations.
This allows the instantiation at (2) to work.

Problema 13

Given the following code, which of the constructors shown in the options can be added to class B without causing a compilation to fail?

```
class A{
   int i;
   public A(int x) { this.i = x; }
}
class B extends A{
   int j;
   public B(int x, int y) { super(x);  this.j = y; }
}
```

You had to select 2 options

- `B( ) { }`
- `B(int y ) { j = y; }`
- `B(int y ) { super(y*2 ); j = y; }`
- `B(int y ) { i = y; j = y*2; }`
- `B(int z ) { this(z, z); }`

Explanation:
1. Remember that an instance of a class is also an instance of its parent class. Therefore, as a part of constructing an instance of a subclass, the JVM has to initialize those parts of the instance that are inherited from the super class as well. Further, the parts inherited from the super class need to be initialized first because the subclass may depend on them. Since it is the job of a constructor to initialize an instance, a constructor of the super class has to be invoked before the constructor of the subclass can proceed. The compiler ensures that at least one constructor of the super class is invoked if you do not explicitly call a super class's constructor by adding super(); (i.e. a call to the no-args constructor) as the first line of the sub class constructor. It automatically adds this call IF and ONLY IF the subclass's constructor does not explicitly call any of the super class's constructor in the first line of its code.

Now, if the super class (which means class A in this question) does not have a no-args constructor, the call to `super()`; will fail. Hence, choices `B( ) { }`, `B(int y ) { j = y; }` and `B(int y ) { i = y; j = y*2; }` are not valid because these constructors do not explicitly invoke the super class's construct and the compiler automatically inserts a call to `super()` in them but A does not have a no-args constructor.

Choice `B(int y ) { super(y*2 ); j = y; }` is valid because it explicitly calls `super( int )`, which is available in A.

2. Instead of calling a super class's constructor using super(<args>), you can also call another constructor of the sub class in the first line (as given in choice `B(int z ) { this(z, z); }`). The first line of this constructor is a call to `this(int, int)`, which causes `B(int, int)` to be invoked and `B(int, int)` calls super(int). So the super class A is correctly instantiated before the sub class B begins initialization.

Problema 14

Consider the following class:

```
class TestClass{
    void probe(int... x) { System.out.println("In ..."); }  //1

    void probe(Integer x) { System.out.println("In Integer"); } //2

    void probe(long x) { System.out.println("In long"); } //3

    void probe(Long x) { System.out.println("In LONG"); } //4

    public static void main(String[] args){
        Integer a = 4; new TestClass().probe(a); //5
        int b = 4; new TestClass().probe(b); //6
    }
}
```

What will it print when compiled and run?
You had to select 2 options

- In Integer and In long
- In ... and In LONG, if //2 and //3 are commented out.
- In Integer and In ..., if //4 is commented out.
- It will not compile, if //1, //2, and //3 are commented out.
- In LONG and In long, if //1 and //2 are commented out.

Explanation:

To answer this type of questions, you need to know the following rules:

1. The compiler always tries to choose the most specific method available with least number of modifications to the arguments.
2. Java designers have decided that old code should work exactly as it used to work before boxing-unboxing functionality became available.
3. Widening is preferred to boxing/unboxing (because of rule 2), which in turn, is preferred over var-args.

Thus,
1.
`probe(Integer)` will be bound to `probe(Integer)` (exact match). If that is not available, it will be bound to `probe(long)`, and then with `probe(int...)` in that order of preference.
`probe(long)` is preferred over `probe(int...)` because unboxing an Integer gives an int and in pre 1.5 code `probe(long)` is compatible with an int (Rule 2).

It is never bound to `probe(Long )` because `Integer` and `Long` are different object types and there is no IS-A relation between them. (This holds true for any two wrapper classes).
It could, however, be bound to `probe(Object )` (if it existed), because Integer `IS-A` Object.

2.
`probe(int)` is bound to probe(long) (because of Rule 2) , then to `probe(Integer )` because boxing an int qives you an Integer, which matches exactly to `probe(Integer)`, and then to `probe(int...)`.

It is never bound to `probe(Long )` because int is not compatible with Long.

We advise you to run this program and try out various combinations. The exam has questions on this pattern but they are not this tough. If you have a basic understanding, you should be ok.

Problema 15

Consider the following code:

```
public class TestClass{
   public void method(Object o){
      System.out.println("Object Version");
   }
   public void method(java.io.FileNotFoundException s){
      System.out.println("java.io.FileNotFoundException Version");
   }
   public void method(java.io.IOException s){
      System.out.println("IOException Version");
   }
   public static void main(String args[]){
      TestClass tc = new TestClass();
      tc.method(null);
   }
}
```

What would be the output when the above program is compiled and run? (Assume that `FileNotFoundException` is a subclass of `IOException`, which in turn is a subclass of Exception)

- It will `print Objec`t Version
- It will `print java.io.IOException` Version
- It will `print java.io.FileNotFoundException` Version
- It will not compile.
- It will throw an exception at runtime.

Explanation:

The reason is quite simple, the most specific method depending upon the argument is called. Here, null can be passed to all the 3 methods but `FileNotFoundException` class is the subclass of `IOException` which in turn is the subclass of Object. So, `FileNotFoundException` class is the most specific class. So, this method is called. Had there been two most specific methods, it would not even compile as the compiler will not be able to determine which method to call. For example:

```
public class TestClass{
    public void method(Object o){
       System.out.println("Object Version");
    }
    public void method(String s){
       System.out.println("String Version");
    }
    public void method(StringBuffer s){
       System.out.println("StringBuffer Version");
    }
    public static void main(String args[]){
       TestClass tc = new TestClass();
       tc.method(null);
    }}
```

Here, `null` can be passed as both `StringBuffer` and `String` and none is more specific than the other. So, it will not compile.

Problema 16

Complete the following code by filling the two blanks -

```
class XXX{
    public void m() throws Exception{
        throw new Exception();
    }
}
class YYY extends XXX{
    public void m(){ }

    public static void main(String[] args) {
        ________  obj = new ______();
        obj.m();
    }
}
```

You had to select 1 option

- XXX XXX
- XXX YYY
- YYY XXX
- YYY YYY

Explanation:

This question is based on two concepts -

1. The overriding method may choose to have no `throws` clause even if the overridden method has a throws clause. Thus, the method m in YYY is valid.

2. Whether a call needs to be wrapped in a try/catch or whether the enclosing method requires a throws clause depends on the class of the reference and not the class of the actual object. This is because it is the compiler that checks whether a call needs to have exception handling and the compiler knows only about the declared class of a variable. It doesn't know about the actual object referred to by that variable (which is known only to JVM at run time).

Here, if you define `obj` of type XXX, the call `obj.m()` will have to be wrapped into a try/catch because `main()` doesn't have a throws clause. But if you define `obj` of class `YYY`, there is no need of try catch because `YYY's m()` does not throw an exception.

Now, if the declared class of obj is YYY, you cannot assign it an object of class XXX because XXX is a superclass of YYY. So the only option left is to do:

YYY `obj = new YYY();`

Problema 17

Given the following definition of class, which member variables are accessible from OUTSIDE the package com.enthu.qb?

```
package com.enthu.qb;
public class TestClass{
   int i;
   public int j;
   protected int k;
   private int l;
}
```

You have to select 2 Options:

Member variable i.
No modifier means package(or default access) and is only accessible inside the package.

Member variable j.
public things (classes, methods and fields) are accessible from anywhere.

Member variable k.
Only if the accessing class is a subclass of TestClass.

Member variable k, but only for subclasses.
protected things (methods and fields) can be accessed from within the package and from subclasses

Member variable l.
private things are accessible only from the class that has it.

Explanation:
`public > protected > package (i.e. no modifier) > private`
where `public` is least restrictive and `private` is most restrictive.
Remember:
`protected` is less restrictive than package access. So a method(or field) declared as protected will be accessible from a subclass even if the subclass is not in the same package.
The same is not true for package access.
A top level class can only have either `public` or no access modifier but a method or field can have all the four. Note that `static`, `final`, `native` and `synchronized` are not considered as access modifiers.

Problema 18

Which of the following are true regarding overloading of a method?
You had to select 1 option

- An overloading method must have a different parameter list and same return type as that of the overloaded method.
    - There is no restriction on the return type. If the parameters are different then the methods are totally different (other than the name) so their return types can be anything.

- If there is another method with the same name but with a different number of arguments in a class then that method can be called as overloaded.

- If there is another method with the same name and same number and type of arguments but with a different return type in a class then that method can be called as overloaded.
    - For overloading a method, the "signature" of the overloaded methods must be different. In simple terms, a method signature includes method name and the number and type of arguments that it takes. So if the parameter list of the two methods with the same name are different either in terms of number or in terms of the types of the parameters, then they are overloaded.
    - For example: Method m1 is overloaded if you have two methods : void m1(int k); and void m1(double d); or if you have: void m1(int k); and void m1(int k, double d);
    - Note that return type is not considered a part of the method signature.

- An overloaded method means a method with the same name and same number and type of arguments exists in the super class and sub class.
    - This is called overriding and not overloading.

Problema 19

Consider the following class...

```
class TestClass{
   int x;
   public static void main(String[] args){
      // lot of code.
   }
}
```

You had to select 1 option

- By declaring x as static, main can access this.x
- By declaring x as public, main can access this.x
- By declaring x as protected, main can access this.x
- main cannot access this.x as it is declared now.
    - Because main() is a static method. It does not have 'this'!
- By declaring x as private, main can access this.x

Explanation
// Fijate que las opciones tienen `this.x`

It is not possible to access x from main without making it static. Because main is a static method and only static members are accessible from static methods. There is no 'this' available in main so none of the this.x are valid.

Problema 20

What can be added to the following Person class so that it is properly encapsulated and the code prints 29?

```
class Person{
    //Insert code here
}
public class Employee extends Person{
        public static void main(String[] args) {
            Employee e = new Employee();
            e.setAge(29);
            System.out.println(e.getAge());
        }
}
```

You had to select 2 options

```
private int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
```

```
protected int age;
   public int getAge() {
       return age;
   }
   public void setAge(int age) {
       this.age = age;
   }
```

protected is not a valid way to encapsulate a field because any class in a package can access the field.

```
int age;
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
```

No access modifer to age means it has default access i.e. all the members of the package can access it. This breaks encapsulation.

```
private int age;
   private int getAge() {
       return age;
   }
   private void setAge(int age) {
       this.age = age;
   }
```

If you make getAge and setAge private, you cannot call them from Employee class.

```
    private int age;
    public int getAge() {
        return age;
    }
    protected void setAge(int age) {
        this.age = age;
    }
```

This is a ambiguous question because it doesn't give all the information. It really depends on the business logic of the class and the whole application whether the accessor methods (and specially the setter) should be public or protected or even private. The field should be private.

Problema 21

Consider the following class...

```
public class ParamTest {
  public static void printSum(int a, int b){
      System.out.println("In int "+(a+b));
  }

  public static void printSum(Integer a, Integer b){
      System.out.println("In Integer "+(a+b));
  }

  public static void printSum(double a, double b){
      System.out.println("In double "+(a+b));
  }

  public static void main(String[] args) {
      printSum(1, 2);
  }
}
```

What will be printed?

- In int 3
- In Integer 3
- In double 3.0
- In double 3
- It will not compile.

Explanation:
// se va con los primitivos

The call to printSum(1, 2) will be bound to printSum(int, int) because 1 and 2 are ints, which are exact match to int, int.

Note that if printSum(int, int) method were not there in the code, printSum(double, double) would have been invoked instead of printSum(Integer, Integer) because widening is preferred over boxing/unboxing.

Problema 22

Which lines contain a valid constructor in the following code?

```
public class TestClass{
   public TestClass(int a, int b) { } // 1
   public void TestClass(int a) { }   // 2
   public TestClass(String s); // 3
   private TestClass(String s, int a) { }     //4
   public TestClass(String s1, String s2) { }; //5
}
```

You had to select 3 options

- Line // 1
- Line // 2
  - Constructors cannot return anything. Not even void.
- Line // 3
  - Constructors cannot have empty bodies (i.e. they cannot be abstract)
- Line // 4
  - You may apply public, private, and protected modifiers to a constructor. But not static, final, synchronized, native and abstract. It can also be package private i.e. without any access modifier.
- Line // 5
  - The compiler ignores the extra semi-colon.

Problema 23

What will the following program print when run?

```
public class ChangeTest {
    private int myValue = 0;

    public void showOne(int myValue){
        myValue = myValue;
    }

    public void showTwo(int myValue){
        this.myValue = myValue;
    }
    public static void main(String[] args) {
        ChangeTest ct = new ChangeTest();
        ct.showTwo(200);
        System.out.println(ct.myValue);
        ct.showOne(100);
        System.out.println(ct.myValue);
    }
}
```

You had to select 1 option

- 0 followed by 100.
- 100 followed by 100.
- 0 followed by 200.
- 100 followed by 200.
- 200 followed by 200.
- 200 followed by 100

Explanation

// Las cosas claras, asigna al local, no al bueno.

There are a couple of important concepts in this question:

1. Within an instance method, you can access the current object of the same class using 'this'. Therefore, when you access this.myValue, you are accessing the instance member myValue of the ChangeTest instance.

2. If you declare a local variable (or a method parameter) with the same name as the instance field name, the local variable "shadows" the member field. Ideally, you should be able to access the member field in the method directly by using the name of the member (in this example, myValue). However, because of shadowing, when you use myValue, it refers to the local variable instead of the instance field.

In showTwo method, there are two variables accessible with the same name myValue. One is the method parameter and another is the member field of ChangeTest object. Ideally, you should be able to access the member field in the method directly by using the name myValue but in this case, the method parameter shadows the member field because it has the same name. So by doing this.myValue, you are changing the instance variable myValue by assigning it the value contained in local variable myValue, which is 200. So in the next line when you print ct.myValue, it prints 200.

Now, in the showOne method also, there are two variables accessible with the same name myValue. One is the method parameter and another is the member field of ChangeTest object. So when you use myValue, you are actually using the method parameter instead of the member field.

Therefore, when you do : myValue = myValue; you are actually assigning the value contained in method parameter myValue to itself. You are not changing the member field myValue. Hence, when you do System.out.println(ct.myValue); in the next line, it still prints 200.

Problema 24

Consider the following code:

```
public class MyClass {

  protected int value = 10;

}
```

Which of the following statements are correct regarding the field value?

You had to select 1 option

- It cannot be accessed from any other class.
- It can be read but cannot be modified from any other class.
- It can be modified but only from a subclass of MyClass.
    - It can also be modified from any class defined in the same package.
- It can be read and modified from any class within the same package.
    - Note that since value is protected, a class in another package which extends MyClass will only inherit this variable, but it cannot read or modify the value of a variable of a MyClass instance. For example:

```
//in different package
class X extends MyClass{
   public static void main(String[] args){
      int a = new MyClass().value; //This will not compile
//because X does not own MyClass's value.
       a = new X().value; //This will compile fine because X inherits value.
   }
}
```

Problema 25

Consider the following code appearing in the same file:

```
class Data {
    int x = 0, y = 0;
    public Data(int x, int y){
        this.x = x; this.y = y;
    }
}
public class TestClass {
    public static void main(String[] args) throws Exception {
        Data d = new Data(1, 1);
        //add code here
    }
}
```

Which of the following options when applied individually will change the Data object currently referred to by the variable d to contain 2, 2 as values for its data fields?

You had to select 2 options

- Add the following two statements:
  d.x = 2;
  d.y = 2;

- Add the following statement:
  d = new Data(2, 2);
  - This will create a new Data object and will not change the original Data object referred to be d.

- Add the following two statements:
  d.x += 1;
  d.y += 1;

- Add the following statement:
  d = d + 1;
  - This will not compile because Java does not allow operator overloading for user defined classes.

Problema 26

What will the following program print?

```
public class TestClass{
  static int someInt = 10;
  public static void changeIt(int a){
    a = 20;
  }
  public static void main(String[] args){
    changeIt(someInt);
    System.out.println(someInt);
  }
}
```

You had to select 1 option

- 10
- 20
- It will not compile.
- It will throw an exception at runtime.
- None of the above.

Explanation:
Theoretically, java supports Pass by Value for everything (i.e. primitives as well as Objects). Remember that:
1. Primitives are always passed by value.
2. Object "references" are passed by value. So it looks like the object is passed by reference but actually it is the value of the reference that is passed.

An example:

```
Object o1 = new Object();
```

Let us say, this object is stored at memory location 15000. Since o1 actually stores just the address of the memory location where the object is stored, it contains 15000.

Now, when you call someMethod(o1); the value 15000 is passed to the method. Therefore, this is what happens inside the method someMethod():

```
someMethod( Object localVar) {
```

localVar now contains 15000, which means localVar also points to the same memory location where the object is stored. Therefore, when you call a method on localVar, it will be executed on the same object.

However, when you change the value of localVar itself, for example, if you do localVar=null, then localVar starts pointing to a different memory location. But the original variable o1 still contains 15000 so it still points to the same object.
}

Problema 27

What will the following code print when compiled and run?

```
class X{
    public X(){
        System.out.println("In X");
    }
}
class Y extends X{
    public Y(){
        super();
        System.out.println("In Y");
    }
}
class Z extends Y{
    public Z(){
        System.out.println("In Z");
    }
}
public class Test {
    public static void main(String[] args) {
        Y y = new Z();
    }
}
```

You had to select 1 option

It will not compile.	In X	In Z	In Y	In Z
	In Y	In Y	In X	In X
	In Z	In X	In Z	In Y

There is no problem with the code. // Remember that before the fields of a subclass can be initialized by a constructor, the fields of superclass have to be initialized. Therefore, a super class constructor must first execute before a subclass constructor can execute. This order of invocation of constructors goes up the chain up from the subclass that is being created up to the top most super class, which is java.lang.Object. // The explicit call to super(); in class Y is not required because the compiler puts a call to super(); anyway if you don't explicitly call either any super class constructor using super(...) or another constructor of the same class using this(...) first ("..." refers to appropriate arguments as required for a given constructor). // The declared type of a variable is immaterial here. Only the actual class of object that is being instantiated is important. Therefore, if you instantiate class Z, Z's constructor will be invoked, but internally, that constructor will first invoke Y's constructor before executing the rest of Z's constructor. Similarly, Y's constructor will first invoke X's constructor before executing the rest of Y's constructor. Finally, X's constructor will first invoke Objects's constructor before executing the rest of X's constructor. Object class's constructor doesn't print anything so no output is generated because of that. But once that is finished, the remaining code of X constructor's is executed, which prints "In X", then the control goes back to Y's constructor, which prints, "In Y". Finally, the control goes back to Z's constructor, which prints, "In Z".

Problema 28

Consider the following class:

```
public class Test{
    public int id;
}
```

Which of the following is the correct way to make the variable 'id' read only for any other class?
You had to select 1 option

- Make 'id' private.
    - This will not allow others to read or write.
- Make 'id' private and provide a public method getId() which will return its value.
- Make 'id' static and provide a public static method getId() which will return its value.
- Make id 'protected'

Explanation:
This is a standard way of providing read only access to internal variables.

Problema 29

When a class, whose members should be accessible only to members of that class, is coded such a way that its members are accessible to other classes as well, this is called …
You had to select 1 option

- strong coupling
- weak coupling
- strong typing
- weak encapsulation
- weak polymorphism
- high cohesion
- low cohesion

Explanation:
When a class is properly encapsulated, only the members that are part of its public API are publicly accessible to other classes. Rest is all private.

Problema 30

What will the following program print?

```
public class InitTest{
   public InitTest(){
      s1 = sM1("1");
   }
   static String s1 = sM1("a");
   String s3 = sM1("2");{
      s1 = sM1("3");
   }
   static{
      s1 = sM1("b");
   }
   static String s2 = sM1("c");
   String s4 = sM1("4");
    public static void main(String args[]){
        InitTest it = new InitTest();
    }
    private static String sM1(String s){
       System.out.println(s);  return s;
    }
}
```

You had to select 1 option

- The program will not compile.
- It will print : a b c 2 3 4 1
- It will print : 2 3 4 1 a b c
- It will print : 1 a 2 3 b c 4
- It will print : 1 a b c 2 3 4

Explanation
First, static statements/blocks are called IN THE ORDER they are defined. Next, instance initializer statements/blocks are called IN THE ORDER they are defined. Finally, the constructor is called. So, it prints a b c 2 3 4 1.

Problema 31

What will the following class print when compiled and run?

```
class Holder{
   int value = 1;
   Holder link;
   public Holder(int val){ this.value = val; }
   public static void main(String[] args){
      final Holder a = new Holder(5);
      Holder b = new Holder(10);
      a.link = b;
      b.link = setIt(a, b);
      System.out.println(a.link.value+" "+b.link.value);
   }

   public static Holder setIt(final Holder x, final Holder y){
       x.link = y.link;
       return x;
   }

}
```

You had to select 1 option

- It will not compile because 'a' is final.
    - 'a' is final is true, but that only means that a will keep pointing to the same object for the entire life of the program. The object's internal fields, however, can change.
- It will not compile because method setIt() cannot change x.link.
    - Since x and y are final, the method cannot change x and y to point to some other object but it can change the objects' internal fields.
- It will print 5, 10.
- It will print 10, 10.
- It will throw an exception when run.
    - When method setIt() executes, x.link = y.link, x.link becomes null because y.link is null so a.link.value throws NullPointerException.

Problema 32

What will be the result of attempting to compile and run the following class?

```
public class InitTest{
   static String s1 = sM1("a");{
      s1 = sM1("b");
   }
   static{
      s1 = sM1("c");
   }
   public static void main(String args[]){
      InitTest it = new InitTest();
   }
   private static String sM1(String s){
      System.out.println(s);  return s;
   }
}
```

You had to select 1 option

- The program will fail to compile.
- The program will compile without error and will print a, c and b in that order when run.
- The program will compile without error and will print a, b and c in that order when run.
- The program will compile without error and will print c, a and b in that order when run.
- The program will compile without error and will print b, c and a in that order when run.

Explanation:
First, static statements/blocks are called IN THE ORDER they are defined. (Hence, a and c will be printed.) Next, instance initializer statements/blocks are called IN THE ORDER they are defined. Finally, the constructor is called. So, then it prints b.

Problema 33

How can you declare 'i' so that it is not visible outside the package test.

```
package test;
public class Test{
   XXX int i;
   /*  irrelevant code */
}
```

You had to select 2 options

- private
    - Note that the question does not require that 'i' should be accessible from test package. So private is fine.
- public
    - Marking it public will make it accessible from all classes in all packages.
- protected
    - It will make it available to a subclass even if the subclass is in a different package.
- No access modifier
- friend
    - There is no such modifier in Java.

Problema 34

What will be the output when the following program is run?

```
public class TestClass{
    char c;
    public void m1(){
        char[ ] cA = { 'a' , 'b'};
        m2(c, cA);
        System.out.println( ( (int)c)  + "," + cA[1] );
    }
    public void m2(char c, char[ ] cA){
        c = 'b';
        cA[1] = cA[0] = 'm';
    }
    public static void main(String args[]){
        new TestClass().m1();
    }
}
```

You had to select 1 option

- Compile time error.
    - c is an instance variable of numeric type so it will be given a default value of 0, which prints as empty space.
- ,m
    - Without the cast to int, c would be printed as empty space and cA[1] is 'm'
- 0,m
    - Because of the explicit cast to int in the println() call, c will be printed as 0.
- b,b
- b,m

Explanation:

Note that Arrays are Objects (i.e. cA instanceof Object is true) so are effectively passed by reference. So in m1() the change in cA[1] done by m2() is reflected everywhere the array is used.

c is a primitive type and is passed by value. In method m2() the passed parameter c is different from the instance variable 'c' because local variables (and method parameters) shadow instance variables with same name. So instance member 'c' keeps its default (i.e. 0) value.

Problema 35

Consider the following classes...

```
class Teacher{
      void teach(String student){
          /* lots of code */
      }
}
class Prof extends Teacher{
        //1
}
```

Which of the following methods can be inserted at line //1 ?
You had to select 4 options

- public void teach() throws Exception
    - It overloads the teach() method instead of overriding it.
- private void teach(int i) throws Exception
    - It overloads the teach() method instead of overriding it.
- protected void teach(String s)
    - This overrides Teacher's teach method. The overriding method can have more visibility. (It cannot have less. Here, it cannot be private.)
- public final void teach(String s)
    - Overriding method may be made final.
- public abstract void teach(String s)
    - This is wrong because class Prof has not been declared as abstract. Note that otherwise it is OK to override a method by an abstract method.

Explanation:
Note that 'protected' is less restrictive than default 'no modifier'. So choice 3 is valid.
"public abstract void teach(String s)" would have been valid if class Prof had been declared abstract.

Problema 36

What will the following code print when compiled and run?

```
interface Pow{
    static void wow(){
        System.out.println("In Pow.wow");
    }
}

abstract class Wow{

   static void wow(){
       System.out.println("In Wow.wow");
  }
}

public class Powwow extends Wow implements Pow {
    public static void main(String[] args) {
        Powwow f = new Powwow();
        f.wow();
    }
}
```

You had to select 1 option

- In Wow.wow
    - Observe that f is reference variable of type Powwow. Since Powwow extends Wow, Powwow inherits the static method wow() from Wow. Java allows a static method of a class to be invoked using a reference variable and so f.wow() invoke's Wow's wow().
    - Static methods of a interface are not inherited in the same way by an implementing class. Therefore, the static method wow() defined in the interface Pow, cannot be accessed through a reference variable. It can only be accessed using the name of the interface i.e. using Pow.wow().
- In Pow.wow
- It will print In Pow.wow if f.wow() is replaced with ((Pow)f).wow();
    - It will not compile if you make this change because it is illegal to invoke a static method of an interface using a reference variable. If you want to invoke a static method of an interface, you need to use the name of the interface. For example, Pow.wow();
    - You can use a reference variable to invoke a static method of a class or super class though. So this is valid: ((Wow)f).wow(); even though it is not a good practice to use a reference to invoke a static method.
- It will not compile.

Problema 37

What will the following code print when run?

```
class A{
    String value = "test";
    A(String val){
        this.value = val;
    }
}
public class TestClass {
    public static void main(String[] args) throws Exception {
        new A("new test").print();
    }
}
```

You had to select 1 option

- test
- new test
- It will not compile.
    - There is no method named print() defined in class A. Further, there is no such method in class Object either. To print the contents of an object you can use toString() method that returns a String: System.out.println(a.toString());
    - However, for this to print a meaningful value, class A should override the Object class's toString() method to return a meaningful String.
- It will throw an exception at run time.

Problema 38

Consider the following code appearing in the same file:

```
class Data {
    private int x = 0, y = 0;
    public Data(int x, int y){
        this.x = x; this.y = y;
    }
}
public class TestClass {
    public static void main(String[] args) throws Exception {
        Data d = new Data(1, 1);
        //add code here
    }
}
```

Which of the following options when applied individually will change the Data object currently referred to by the variable d to contain 2, 2 as values for its data fields?

You had to select 1 option

- Add the following two statements :
  d.x = 2;
  d.y = 2;
  - Note that x and y are private in class Data. Therefore, you cannot access these members from any other class.

- Add the following statement:
  d = new Data(2, 2);
  - This will create a new Data object and will not change the original Data object referred to be d.

- Add the following two statements:
  d.x += 1;
  d.y += 1;
  - Note that x and y are private in class Data. Therefore, you cannot access these members from any other class.

- Add the following method to Data class:
  public void setValues(int x, int y){
  this.x.setInt(x); this.y.setInt(y);
  }
  Then add the following statement:
  d.setValues(2, 2);
  - x is primitive int. You cannot call any methods on a primitive. so this.x.setInt(...) or this.y.setInt(...) don't make any sense.

- Add the following method to Data class:
  public void setValues(int x, int y){
  this.x = x; this.y = y;
  }
  Then add the following statement:
  d.setValues(2, 2);
  - This is a good example of encapsulation where the data members of Data class are private and there is a method in Data class to manipulate its data. Compare this approach to making x and y as public and letting other classes directly modify the values.