06 Java Working with Inheritance

Problema 1

What will be printed when the following code is compiled and run?

```
class A {
    public int getCode(){ return 2;}
}

class AA extends A {
  public long getCode(){ return 3;}
}

public class TestClass {

    public static void main(String[] args) throws Exception {
         A a = new A();
        A aa = new AA();
        System.out.println(a.getCode()+" "+aa.getCode());
    }

    public int getCode() {
        return 1;
    }
}
```

You had to select 1 option

- 2 3
- 2 2
- It will throw an exception at run time.
- The code will not compile.

Explanation:
// Parece que no hace un @Override porque está regresando un 'long', en lugar de un 'int'

Class AA is trying to override `getCode()` method of class A but its return type is incompatible with the A's `getCode`.

When the return type of the overridden method (i.e. the method in the base/super class) is a primitive, the return type of the overriding method (i.e. the method in the sub class) must match the return type of the overridden method.

In case of Objects, the base class method can have a covariant return type, which means, it can return either return the same class or a sub class object. For example, if base class method is:

```
  public A getA(){ ... }
```

a subclass can override it with:
`public AA getA(){ ... }` because AA is a subclass of A.

Problema 2

What, if anything, is wrong with the following code?

```
interface T1{
}
interface T2{
   int VALUE = 10;
   void m1();
}

interface T3 extends T1, T2{
   public void m1();
   public void m1(int x);
}
```

You had to select 1 option

- T3 cannot implement both T1 and T2 because it leads to ambiguity.
- There is nothing wrong with the code.
- The code will work fine only if VALUE is removed from T2 interface.
- The code will work fine only if m1() is removed from either T2 or T3.
- None of the above.

Explanation:
Having ambiguous fields or methods does not cause any problem by itself (except in the case of default methods) but referring to such fields or methods in an ambiguous way will cause a compile time error.

Having `m1()` in T3 is also fine because even though `m1()` is declared in `T2` as well as `T3` , the definition to both resolves unambiguously to only one `m1()`. Explicit cast is not required for calling the method `m1()`: `( ( T2) t).m1();`

`m1(int x)` is valid because it is a totally independent method declared by `T3`.

Problema 3

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

```
class A{
   final int fi = 10;
}
public class B extends A{
   int fi = 15;
   public static void main(String[] args){
       B b = new B();
       b.fi = 20;
       System.out.println(b.fi);
       System.out.println(  (  (A) b  ).fi  );
   }
}
```

You had to select 1 option

- It will not compile.
- It will print 10 and then 10
- It will print 20 and then 20
- It will print 10 and then 20
- It will print 20 and then 10

Explanation:

// Aunque sea `'final'` puede ser oculta en el hijo o clase B

Note that a final variable can be hidden. Here, although `fi` in `A` is `final`, it is hidden by `fi` of `B`. So `b.fi = 20;` is valid since B's fi is not final.

Problema 4

Given the following line of code:

```
List students = new ArrayList();
```

Identify the correct statement:

You had to select 1 option

- The reference type is List and the object type is ArrayList.
    - Since you are doing new ArrayList, you are creating an object of class ArrayList. You are assigning this object to variable "students", which is declared of class List. Reference type means the declared type of the variable.
- The reference type is ArrayList and the object type is ArrayList.
- The reference type is ArrayList and the object type is List.
- The reference type is List and the object type is List.

Problema 5

Given the following class definitions :

```
interface MyIface{};
class A {};
class B extends A implements MyIface{};
class C implements MyIface{};
```

and the following object instantiations:

```
A a = new A();
B b = new B();
C c = new C();
```

Which of the following assignments are legal at compile time?
You had to select 1 option

- b = c;
  - There is no relation between b and c.
- c = b;
  - There is no relation between b and c.
- MyIface i = c;
  - Because C implements MyIFace.
- c = (C) b;
  - Compiler can see that in no case can an object referred to by b can be of class c. So it is a compile time error.
- b = a;
  - It will fail at compile time because a is of class A and can potentially refer to an object of class A, which cannot be assigned to b, which is a variable of class B. To make it compile, you have to put an explicit cast, which assures the compiler that `'a'` will point to an object of class B (or a subclass of B) at run time. Note that, in this case, an explicit cast can take it through the compiler but it will then fail at run time because `'a'` does not actually refer to an object of class B (or a subclass of B), so the JVM will throw a ClassCastException.

Problema 6 -> Throws exception

What will the following program print when compiled and run?

```
class Game{
  public void play() throws Exception{
    System.out.println("Playing...");
  }
}

public class Soccer extends Game{
   @Override // El hijo no necesita poner a la excepción
   public void play(){
      System.out.println("Playing Soccer...");
   }
   public static void main(String[] args){
       Game g = new Soccer();
       g.play();
   }
}
```

You had to select 1 option

- It will not compile.
- It will throw an Exception at runtime.
- Playing Soccer...
- Playing...
- None of these.

Explanation:

Observe that play() in Game declares Exception in its throws clause. Further, class Soccer overrides the play() method without any throws clause. This is valid because a list of no exception is a valid subset of a list of exceptions thrown by the superclass method.

Now, even though the actual object referred to by 'g' is of class Soccer, the class of the variable g is of class Game. Therefore, at compile time, compiler assumes that g.play() might throw an exception, because Game's play method declares it, and thus expects this call to be either wrapped in a try-catch or the main method to have a throws clause for the main() method.

// Aunque no ejecuta el error, si afecta en el método, Puedes atrapar la excepción en código o en el método main()

Problema 7

Consider the following code:

```
class Super {
    static String ID = "QBANK";
    // static { System.out.print("In Super"); }
    // El bloque 'static' si se dispararía al solo utilizar el llamado del
    // atributo
}

class Sub extends Super{
    static { System.out.print("In Sub"); }
    // necesita crearse, un 'new' y solo se hará este bloque 'static' la
    // primera vez que se haga su instancia
}
public class Test{
    public static void main(String[] args){
        System.out.println(Sub.ID);
        // Aunque se diga 'Sub' nunca llega a sub, se queda en Super.ID
    }
}
```

What will be the output when class Test is run?

You had to select 1 option

- It will print In Sub and QBANK.
- It will print QBANK.
- Depends on the implementation of JVM.
- It will not even compile.
- None of the above.

Explanation:

A class or interface type T will be initialized immediately before the first occurrence of any one of the following:

- T is a class and an instance of T is created.
- T is a class and a static method declared by T is invoked.
- A static field declared by T is assigned.
- A static field declared by T is used and the field is not a constant variable (§4.12.4).
- T is a top level class (§7.6), and an assert statement (§14.10) lexically nested within T (§8.1.3) is executed.
- A reference to a static field (§8.3.1.1) causes initialization of only the class or interface that actually declares it, even though it might be referred to through the name of a subclass, a subinterface, or a class that implements an interface.
- Invocation of certain reflective methods in class Class and in package java.lang.reflect also causes class or interface initialization.
- A class or interface will not be initialized under any other circumstance.

Problema 8

Consider the following classes :

```
interface I{
}
class A implements I{
}

class B extends A {
}

class C extends B{
}
```

And the following declarations:

```
A a = new A();
B b = new B();
```

Identify options that will compile and run without error.

You had to select 1 option

- `a = (B)(I)b;`
  - class B does implement I because it extends A, which implements I. A reference of type I can be cast to any class at compile time (except if the class is final and does not implement I). Since B is-a A, it can be assigned to a.
- `b = (B)(I) a;`
  - This will fail at run time because a does not point to an object of class B.
- `a = (I) b;`
  - An I is not an A. Therefore, it will not compile.
- `I i = (C) a;`
  - It will compile because a C is-a A, which is-a I, and a reference of class A can point to an object of class C. But it will fail at runtime because a does not point to an object of class C.

Problema 9

Which of the following are valid declarations inside an interface independent of each other?

You had to select 2 options

- void compute();
    - All interface methods have to be public. No access control keyword in the method declaration also means public in an interface. (Note that the absence of access control keyword in the method declaration in a class means package protected.)
- public void compute();
- public final void compute();
    - final is not allowed.
- static void compute();
    - An interface can have a static method but the method must have a body in that case.
- protected void compute();
    - All interface methods have to be public.

Problema 10

Consider the following code:

```
class A{
 public XXX m1(int a){
   return a*10/4-30;
 }
}
class A2 extends A{
 public YYY m1(int a){
   return a*10/4.0;
 }
}
```

What can be substituted for XXX and YYY so that it can compile without any problems?
You had to select 1 option

- int, int
  - a*10/4.0; generates a double so, A2's m1() cannot return an int. (It will need a cast otherwise: return (int) (a*10/4.0);)
- int, double
  - The return type should be same for overridden and overriding method.
- double, double
  - a*10/4-30; generates an `'int'` which can be returned as a double without any cast.
- double, int
  - The return type should be same for overridden and overriding method.
- Nothing, they are simply not compatible.

Explanation:
Note that when a method returns objects (as opposed to primitives, like in this question), the principle of covariant returns applies. Meaning, the overriding method is allowed to return a subclass of the return type defined in the overridden method.
Thus, if a base class's method is: `public A m();` then a subclass is free to override it with: `public A1 m();` if A1 extends A.

```
public Object m1(int a){
   return a*10/4-30;
}
```

No regresará un Object, regresará en tipo Integer, para verlo puedes obtenerlo con:
`nombreDeObjeto.getClass().getName();`

Problema 11

Given the following classes, what will be the output of compiling and running the class Truck?

```
class Automobile{
   public void drive() {  System.out.println("Automobile: drive");   }
}

public class Truck extends Automobile{
   public void drive() {  System.out.println("Truck: drive");   }
   public static void main (String args [ ]){
      Automobile  a = new Automobile();
      Truck t  = new Truck();
      a.drive(); //1
      t.drive(); //2
      a = t;     //3
      a.drive(); //4
   }
}
//End of Code
```

You had to select 1 option

Compiler error at line 3.

It will print:
Automobile: drive
Truck: drive
Automobile: drive
in that order.

Runtime error at line 3.

It will print:
Automobile: drive
Truck: drive
Truck: drive
in that order.

It will print:
Automobile: drive
Automobile: drive
Automobile: drive
in that order.

Explanation:
Since `Truck` is a subclass of `Automobile`, `a = t` will be valid at compile time as well at runtime.
But a cast is needed to make `t = (Truck) a;` work.

This will be ok at compile time but if at run time 'a' does not refer to an object of class `Truck`, a `ClassCastException` will be thrown.

Now, method to be executed is decided at run time and it depends on the actual class of object referred to by the variable.

Here, at line 4, variable a refers to an object of class Truck. So Truck's `drive()` will be called which prints `Truck: drive`.

This is polymorphism in action!

Pregunta 12

Which is the first line that will cause compilation to fail in the following program?

```
// Filename: A.java
class A{
   public static void main(String args[]){
      A a = new A();
      B b = new B();
      a = b;  // 1
      b = a;  // 2
      a = (B) b; // 3
      b = (B) a; // 4
   }
}
class B extends A { }
```

You had to select 1 option

- At Line 1.
- At Line 2.
    - Because 'a' is declared of class A and 'b' is of B which is a subclass of A. So an explicit cast is needed.
- At Line 3.
- At Line 4.
- None of the above.

Explanation:

Casting a base class to a subclass as in : `b = (B) a;` is also called as narrowing (as you are trying to narrow the base class object to a more specific class object) and needs explicit cast.
Casting a sub class to a base class as in: `A a = b;` is also called as widening and does not need any casting.

For example, consider two classes: Automobile and Car, where Car extends Automobile
Now, `Automobile a = new Car();` is valid because a car is definitely an Automobile. So it does not need an explicit cast.

But, Car c = a; is not valid because 'a' is an Automobile and it may be a Car, a Truck, or a MotorCycle, so the programmer has to explicitly let the compiler know that at runtime 'a' will point to an object of class Car. Therefore, the programmer must use an explicit cast:

```
Car c = (Car) a;
```

Problema 13

Consider the following code:

```
class Base{
   private float f = 1.0f;
   void setF(float f1){ this.f = f1; }
}
class Base2 extends Base{
   private float f = 2.0f;
   //1
}
```

Which of the following options is/are valid example(s) of overriding?
You had to select 2 options

- `protected void setF(float f1){ this.f = 2*f1; }`
    - protected is less restrictive than default, so it is valid.
- `public void setF(double f1){ this.f = (float) 2*f1; }`
    - Since the parameter type is different, it is overloading not overriding.
- `public void setF(float f1){ this.f = 2*f1; }`
    - public is less restrictive than default, so it is valid.
- `private void setF(float f1){ this.f = 2*f1; }`
    - private is more restrictive than default, so it is NOT valid.
- `float setF(float f1){ this.f = 2*f1; return f;}`
    - return types must match.

Explanation:
An overriding method can be made less restrictive than the overridden method. The restrictiveness of access modifiers is as follows:
private>default>protected>public (where private is most restrictive and public is least restrictive).

Note that there is no modifier named default. The absence of any access modifiers implies default access.

Problema 14

Consider the following code:

```
public abstract class TestClass{
    public abstract void m1();
    public abstract void m2(){
        System.out.println("hello");
    }
}
```

Which of the following corrections can be applied to the above code (independently) so that it compiles without any error?
You had to select 2 options

- Replace the method body of m2() with a ; (semi-colon).
- Replace the ; at the end of m1() with a method body.
- Remove abstract from m2().
    - A method that has a body cannot be abstract. In other words, an abstract method cannot have a body. So either remove the method body (as in m1()) or remove abstract keyword.
- Remove abstract from the class declaration.

Problema 15

What will the following code print when compiled and run?

```
class ABCD{
   int x = 10;
   static int y = 20;
}
class MNOP extends ABCD{
   int x = 30;
   static int y = 40;
}

public class TestClass {
   public static void main(String[] args) {
     System.out.println(new MNOP().x+", "+new MNOP().y);
   }
}
```

You had to select 1 option

- 10, 40
- 30, 20
- 10, 20
- 30, 40
- 20, 30
- Compilation error.

Explanation:
Access to static and instance fields and static methods depends on the class of reference variable and not the actual object to which the variable points to.

Observe that this is opposite of what happens in the case of instance methods. In case of instance methods the method of the actual class of the object is called.

Therefore, in case of `System.out.println(new MNOP().x);` the reference is of type MNOP and so MNOP's x will be accessed.

Had it been like this:

```
   ABCD a = new MNOP();
   System.out.println(a.x);
   System.out.println(a.y);
```

ABCD's x and y would have been accessed because a is of type ABCD even though the actual object is of type MNOP.

Problema 16 -> Hello

What will be the output of compiling and running the following program:

```
class TestClass implements I1, I2{
   public void m1() { System.out.println("Hello"); }
   public static void main(String[] args){
      TestClass tc = new TestClass();
      ( (I1) tc).m1();
   }
}
interface I1{
   int VALUE = 1;
   void m1();
}
interface I2{
   int VALUE = 2;
   void m1();
}
```

You had to select 1 option

- It will print Hello.
- There is no way to access any VALUE in TestClass.
- The code will work fine only if VALUE is removed from one of the interfaces.
    - It works even now.
- It will not compile.
- None of the above.

Explanation:
Having ambiguous fields does not cause any problems but referring to such fields in an ambiguous way will cause a compile time error.

So you cannot call : `System.out.println(VALUE)` as it will be ambiguous.

as there is no ambiguity in referring the field:

```
TestClass tc = new TestClass();
System.out.println(( (I1) tc).VALUE);
```

So, any of the `VALUE` fields can be accessed by casting.

En tiempo de ejecución Java busca el método en el objeto real en el tipo de referencia.
El casting a `'I1'` permite al compilador resolver la ambigüedad mientras que
En tiempo de ejecución se hará el método de `'tc'` ya que es una instancia de TestClass

Problema 17

Consider the following code:

```
class A{
   A() {  print();   }
   void print() { System.out.print("A "); }
}
class B extends A{
   int i =   4;
   public static void main(String[] args){
      A a = new B();
      a.print();
   }
   void print() { System.out.print(i+" "); }
}
```

What will be the output when class B is run ?

You had to select 1 option

- It will print A 4
- It will print A A
- It will print 0 4
- It will print 4 4
- None of the above.

Explanation:

Note that method `print()` is overridden in class B. Due to polymorphism, the method to be executed is selected depending on the class of the actual object.

Here, when an object of class B is created, first B's default constructor (which is not visible in the code but is automatically provided by the compiler because B does not define any constructor explicitly) is called.

The first line of this constructor is a call to `super()`, which invokes A's constructor. A's constructor in turn calls `print()`.

Now, `print` is a non-private instance method and is therefore polymorphic, which means, the selection of the method to be executed depends on the class of actual object on which it is invoked.

Here, since the class of actual object is B, B's `print` is selected instead of A's `print`. At this point of time, variable `'i'` has not been initialized (because we are still in the middle of initializing A), so its default value i.e. 0 is printed.

Finally, 4 is printed.

NOTE: Try this code after declaring `i` in class B as `final` and observe the output.

Problema 18

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

```
// Filename: TestClass.java
public class TestClass{
   public static void main(String args[]){
      B c = new C();
      System.out.println(c.max(10, 20));
   }
}
class A{
   int max(int x, int y)  { if (x>y) return x; else return y; }
}
class B extends A{
  int max(int x, int y)  {  return 2 * super.max(x, y) ; }
}
class C extends B{
  int max(int x, int y)  {  return super.max( 2*x, 2*y); }
}
```

You had to select 1 option

- The code will fail to compile.
- Runtime error.
- The code will compile without errors and will print 80 when run.
- The code will compile without errors and will print 40 when run.
- The code will compile without errors and will print 20 when run.

Explanation:
// Dará hacia arriba el max con los super.max() pero el objeto es C, empieza en 'C'.

Remember the rule that in case of instance methods, it is always the class of the actual object that determines which version of an overridden method is invoked.

Here, observe that the method `max` of class `A` is overridden by class `B` and then again by class `C`. Thus, when main calls `c.max(10, 20)`, class C's `max` will be invoked with parameters 10 and 20 because, even though the declared type of `c` is `B`, the actual object referenced by `c` is of class `C`.

C's `max` calls the `max` method defined in `B` with parameters 2*10 and 2*20 i.e. 20 and 40. The `max` method in `B`, in turn, calls the `max` method in A with the parameters 20 and 40. The `max` method in `A` will return 40 to the `max()` method in `B`. The `max()` method in `B` will return 80 to the `max()` method in `C`. And finally the `max` of `C` will return 80 in `main`, which will be printed out.

Problema 19

Consider the following class hierarchy

```
class A{
   public void m1() {    }
}
class B extends A{
   public void m1() {    }
}
class C extends B{
   public void m1(){
      /*  //1
      ... lot of code.
      */
   }
}
```

You had to select 2 options

- You cannot access class A's m1() from class C for the same object (i.e. this).
- You can access class B's m1() using super.m1() from class C.
- You can access class A's m1() using ((A) this).m1() from class C.
    - Note that selection of method to be executed depends upon the actual object class. So no matter what you do, in class C you can only access C's m1() even by casting this to B or A. So, this option will not work.
- You can access class A's m1() using super.super.m1() from class C.

Explanation:

`super.super` is an invalid construct.

So, there is no way you can access `m1()` of A from C.

Problema 20

Consider the following program:

```
class Game {
  public void play() throws Exception   {
    System.out.println("Playing...");
  }
}

class Soccer extends Game {
   public void play(String ball)    {
      System.out.println("Playing Soccer with "+ball);
   }
}

public class TestClass {
   public static void main(String[] args) throws Exception  {
       Game g = new Soccer();
       // 1
       Soccer s = (Soccer) g;
       // 2
   }
}
```

Which of the given options can be inserted at //1 and //2?
You had to select 2 options

- It will not compile as it is.
    - There is no problem with the existing code.
- It will throw an Exception at runtime if it is run as it is.
    - Soccer s = (Soccer) g; is a valid because g does refer to an object of class Soccer at run time. So there will be no exception at run time.
- `g.play(); at //1 and s.play("cosco"); at //2`
    - This is valid because g is of type Game, which has the no-args play method and s is of type Soccer, which has defined play(String) method.
- `g.play(); at //1 and s.play(); at //2`
    - This is valid because g is of type Game, which has the no-args play method and s is of type Soccer, which inherits that method.
- `g.play("cosco"); at //1 and s.play("cosco"); at //2`
    - g.play("cosco") is not valid because even though the object referred to by g is of class Soccer, the reference type of g is Game, which does not have play(String) method.

// Recuerda que cuando el método es @Override es cuando decide ir por el que esté apuntando al objeto, pero si el método no es @Override entonces se tiene que hacer un ‘cast’ para acceder.

Problema 21

Consider the following classes...

```
class Car{
   public int gearRatio = 8;
   public String accelerate() {  return "Accelerate : Car";  }
}
class SportsCar extends Car{
   public int gearRatio = 9;
   public String accelerate() {  return  "Accelerate : SportsCar";  }
   public static void main(String[] args){
      Car c = new SportsCar();
      System.out.println( c.gearRatio+"  "+c.accelerate() );
                  // de la var. referencia   // del Objeto
   }
}
```

What will be printed when SportsCar is run?
You had to select 1 option

- 8 Accelerate : Car
- 9 Accelerate : Car
- 8 Accelerate : SportsCar
- 9 Accelerate : SportsCar
- None of the above.

Explanation:
The concept is : variables are hidden and methods are overridden.

Method to be executed depends on the class of the actual object the variable is referencing to.

Here, c refers to object of class SportsCar so SportsCar's accelerate() is selected.

Problema 22

What will be the result of compiling and running the following code?

```
class Base{
   public short getValue(){ return 1; } //1
}
class Base2 extends Base{
   public byte getValue(){ return 2; } //2
                                       // Crashea porque no es @Override
}
public class TestClass{
   public static void main(String[] args){
      Base b = new Base2();
      System.out.println(b.getValue()); //3
   }
}
```

You had to select 1 option

- It will print 1
- It will print 2.
- Compile time error at //1
- Compile time error at //2
- Compile time error at //3

Explanation:
In case of overriding, the return type of the overriding method must match exactly to the return type of the overridden method if the return type is a primitive.

//+----------+

(In case of objects, the return type of the overriding method may be a subclass of the return type of the overridden method.)

//+----------+

Problema 23

Consider the following code:

```
public class SubClass extends SuperClass{
     int i, j, k;
     public SubClass( int m, int n )     {  i = m ;  j = m ;  } //1
     public SubClass( int m )  {   super(m );   } //2
 }
```

Which of the following constructors MUST exist in SuperClass for SubClass to compile correctly?
You had to select 2 options

- It is ok even if no explicit constructor is defined in SuperClass
    - The //2 will fail as it needs a constructor taking an int!
- `public SuperClass(int a, int b)`
    - It is not used anywhere so it is not necessary.
- `public SuperClass(int a)`
    - Because it is called in the second constructor of SubClass.
- `public SuperClass()`
    - The default no args constructor will not be provided because SuperClass has to define one arg constructor.
    - // Este será llamado aún si el hijo no lo tiene, por eso es necesario. Se crea solo si no hay más constructores pero como si o si ya hay uno creado al ser llamado por ‘super’ también deberá añadirse este.
- only `public SuperClass(int a)` is required.
    - You'll have to explicitly define a no args constructor because it is needed in the first constructor of SubClass.

Problema 24

Which of the following are valid declarations inside an interface independent of each other?

You had to select 2 options

- void compute();
    - All interface methods have to be public. No access control keyword in the method declaration also means public in an interface. (Note that the absence of access control keyword in the method declaration in a class means package protected (default).)
- public void compute();
    - Prácticamente es lo de arriba pero explicito o escrito
- public final void compute();
    - final is not allowed.
- static void compute();
    - An interface can have a static method but the method must have a body in that case.
- protected void compute();
    - All interface methods have to be public.

Problema 25

Which one of these is a proper definition of a class TestClass that cannot be sub-classed?

You had to select 1 option

- final class TestClass { }
- abstract class TestClass { }
- native class TestClass { }
- static class TestClass { }
- private class TestClass { }

Explanation:
A final class cannot be subclassed.

Although declaring a method static usually implies that it is also final, this is not true for classes. An inner class can be declared static and still be extended. Note that for classes, final means it cannot be extended, while for methods, final means it cannot be overridden in a subclass. The native keyword can only be used on methods, not on classes and or variables.

Problema 26 -> Exceptions

How can you fix the following code to make it compile:

```
import java.io.*;
class Great {
    public void doStuff() throws FileNotFoundException{
    }
}

class Amazing extends Great {
  public void doStuff() throws IOException, IllegalArgumentException{
  }
}

public class TestClass {
    public static void main(String[] args) throws IOException{
        Great g = new Amazing();
        g.doStuff();
    }
}
```

Assume that changes suggested in a option are to be applied independent of other options.

You had to select 2 options

- Change doStuff in Amazing to throw only IllegalArgumentException.
    - IllegalArgumentException extends from RuntimeException. So you don't have to worry about it at least at compile time. You may or may not declare it in the throws clause. The caller doesn't have to catch it anyway. The overriding method in the subclass is free to not throw any checked exception at all even if the overridden method throws a checked exception. No exception is a valid subset of exceptions thrown by the overridden method.
- Change doStuff in Great to throw FileNotFoundException and IllegalArgumentException.
- Change doStuff in Amazing to throw only IOException.
- Change doStuff in Great to throw only IOException instead of FileNotFoundException.
- Replace g.doStuff() to ((Amazing) g).doStuff().

Explanation:
// Realmente es saber la jerarquía de los errores, pero un método con error padre no puede tener heredado un método @overriden con un error padre

The rule is that an overriding method cannot throw an exception that is a super class of the exception thrown by the overridden method.
Now, FileNotFoundException is a subclass of IOException. Therefore, Amazing's doStuff() cannot throw IOException if the base class's doStuff throws only FileNotFoundException.

```
FileNotFoundException  {  IOException
IllegalArgumentException   {  RuntimeException
```

Problema 27 -> Hidden Explanation

Consider the following classes :

```
class A{
    public static void sM1() {  System.out.println("In base static"); }
}

class B extends A{

Line 1 --> // public static void sM1() {  System.out.println("In sub
static"); }

Line 2 --> // public  void sM1() {  System.out.println("In sub non-static");
}

}
```

Which of the following statements are true?
You had to select 2 options

- class B will not compile if line 1 is uncommented.
    - static method sM1() can be shadowed by a static method sM1() in the subclass.
- class B will not compile if line 2 is uncommented.
    - static method cannot be overridden by a non-static method and vice versa
- class B will not compile if line 1 and 2 are both uncommented.
- Only the second option is correct.
- Only the third option is correct.

Explanation:
// Añade otro concepto para recordar de Hidden en 'static' métodos, no pueden ser @Override.
Another concept (although not related to this question but about static methods) is that static methods are never overridden. They are hidden just like static or non-static fields. For example,

```
class A{
  int i = 10;
  public static void m1(){  }
  public void m2() { }
}
class B extends A{
  int i = 20;
  public static void m1() {  }
  public void m2() { }
}
```

Here, UNLIKE m2, m1() of B does not override m1() of A, it just hides m1() of A, as proven by the following code:

```
A a  = new B();
System.out.println(a.i)  //will print 10 instead of 20
a.m1();  //will call A's m1
```

```
a.m2();  //will call B's m2 as m2() is not static and so overrides A's m2()
```

Problema 28 ->Over ‘static’

You are modeling a class hierarchy for living things. You have a class LivingThing which has an abstract method reproduce().
Now, you want to have 2 concrete subclasses of LivingThing - Plant and Animal. Both do reproduce but the mechanisms are different. What would you do?

You had to select 1 option

- Overload the reproduce method in Plant and Animal classes
- Overload the reproduce method in LivingThing class.
- Override the reproduce method in Plant and Animal classes
- Either overload or override reproduce in Plant and Animal classes, it depends on the preference of the designer.

Explanation:
This kind of scenario where the subclass HAS the behavior of the base class but implements it in a different way is called as overriding.
Here, both Plant and Animal reproduce, so they have the behavior of the base class but they do it differently, so you have to override the base class method in their code. Inheritance is always involved in overriding.

Overloading is quite different, when you want to do similar (not same) things but the inputs are different then you overload a method. For example, you may have two add methods: add(int i1, int i2) and add(ComplexNo c1, ComplexNo c2). Here both are doing similar things (that is why both are named as add) but inputs are different. Both are two entirely different methods and there is no inheritance involved.

El @OVERLOADING se puede aplicar con static en el padre y el hijo con el mismo método, serían métodos distintos, Hidden de ello.

El `‘final’` en un padre, no se le puede aplicar Hidden, no puedes tener @OVERLOADING.
Si hay herencia, si puede iniciar un hijo con ese método al ser `‘final’`

Si se puede `@OVERLOADING` con métodos `‘static’` en una misma clase.

El overloading puede tener un modificador de acceso de menor acceso que el principal.

- Para los atributos o fields, pueden ser los hijos o los padres ‘static’, no tienen que ser ambos.
- La visibilidad no importa, es decir, los modificadores de acceso.
- Static o no, siempre apuntará a la variable de referencia

Problema 29

Given the following class definition:

```
class A{
  protected int i;
  A(int i) {    this.i = i;    }

}
// 1 : Insert code here
```

Which of the following would be a valid class that can be inserted at //1 ?
You had to select 2 options

- `class B {}`
- `class B extends A {}`
  - Since class B does not have any constructor, the compiler will try to insert the default constructor, which will look like this: B(){
    //Notice that it is trying to call the no args constructor of the super class, A. super(); }
    Since A doesn't have any no-args constructor, the above code will fail to compile.
- `class B extends A {  B()  {  System.out.println("i = " + i); }  }`
  - It has the same problem as the one above.
- `class B { B() {} }`

// El problema realmente parece que las otras opciones no pueden ser válidas porque requieren que A tenga un constructor vacio, las otras clases B son independientes.

Problema 30

Which of the following lines of code that, when inserted at line 1, will make the overriding method in SubClass invoke the overridden method in BaseClass on the current object with the same parameter.

```
class BaseClass{
   public void print(String s) {  System.out.println("BaseClass :"+s); }
}
class SubClass extends BaseClass{
   public void print(String s){
      System.out.println("SubClass :"+s);
      // Line 1
   }
   public static void main(String args[]){
      SubClass sc = new SubClass();
      sc.print("location");
   }
}
```

You had to select 1 option

- `this.print(s);`
- `super.print(s);`
    - This is the right syntax to call the base class's overridden method. However, note that there is no way to call a method if it has been overriden more than once. For example, if you make BaseClass extend from another base class SubBase, and if SubBase also has the same method, then there is no way to invoke SubBase's print method from SubClass's print method. You cannot have something like super.super.print(s);
- `print(s);`
    - This will call the same method and will cause a recursion.
- `BaseClass.print(s);`
    - print is not a static method.

Problema 31 -> Donkey-Paquetes

Given:

```
//in file Movable.java
package p1;
public interface Movable {
  int location = 0;  // es public, static y final si es de 'interfaz'
  void move(int by);
  public void moveBack(int by);
}

//in file Donkey.java
package p2;
import p1.Movable;
public class Donkey implements Movable{
    int location = 200;
    public void move(int by) {
        location = location+by;
    }
    public void moveBack(int by) {
        location = location-by;
    }
}

//in file TestClass.java
package px;
import p1.Movable;
import p2.Donkey;
public class TestClass {
    public static void main(String[] args) {
        Movable m = new Donkey();
        m.move(10);  // location Donkey = 210
        m.moveBack(20);  // location Donkey = 190
        System.out.println(m.location);  // Imprime 'variable de referencia'
    }
}
```

Identify the correct statement(s).

You had to select 1 option

- Donkey.java will not compile.
- TestClass.java will not compile.
- Movable.java will not compile.
- It will print 190 when TestClass is run.
- It will print 0 when TestClass is run.

Explicación: todo está en buena sintaxis, los métodos se llaman en burro, el atributo se pide de movable.

Problema 32

Which of the following class definitions is/are legal definition(s) of a class that cannot be instantiated?

- ```
  class Automobile{
     abstract void honk();  //(1)
  }
  ```

- ```
  abstract class Automobile{
     void honk();   //(2)
  }
  ```

- ```
  abstract class Automobile{
     void honk(){};   //(3)
  }
  ```

- ```
  abstract class Automobile{
     abstract void honk(){}   //(4)
  }
  ```

- ```
  abstract class Automobile{
     abstract void honk();   //(5)
  }
  ```

You had to select 2 options

- 1
  It will not compile as one of its method is abstract but the class itself is not abstract.
- 2
  It will not compile as the method doesn't have the body and also is not declared abstract.
- 3
  This is a valid abstract class although it doesn't have any abstract method.
- 4
  An abstract method cannot have a method body. {} constitutes a valid method body.
- 5
  This is a valid abstract class

Explanation:

Here are some points to remember:

A class is uninstantiable if the class is declared `abstract`.
If a method has been declared as `abstract`, it cannot provide an implementation (i.e. it cannot have a method body) and the class containing that method must be declared `abstract`).
If a method is not declared `abstract`, it must provide a method body (the class can be `abstract` but not necessarily so).
If any method in a class is declared `abstract`, then the whole class must be declared `abstract`.
A class can still be made abstract even if it has no abstract method.
  ```

Problema 33

Consider the contents of following two files:

```
//In file A.java
package a;
public class A{
   A(){ }
   public void  print(){ System.out.println("A"); }
}

//In file B.java
package b;
import a.*;
public class B extends A{
   B(){ }
   public void  print(){ System.out.println("B"); }
   public static void main(String[] args){
      new B();
   }
}
```

What will be printed when you try to compile and run class B?
You had to select 1 option

- It will print A.
- It will print B.
- It will not compile.
    - Because A() is not accessible in B.
- It will compile but will not run.
- None of the above.

Explanation:
Note that there is no modifier for A's constructor. So it has default access. This means only classes in package a can use it. Also note that class B is in a different package and is extending from A.

In B's constructor the compiler will automatically add super() as the first line. But since A() is not accessible in B, this code will not compile.

Problema 34

Consider the following program...

```
class Super {  }
class Sub extends Super {  }
public class TestClass{
   public static void main(String[] args){
      Super s1 = new Super(); //1
      Sub s2 = new Sub();     //2
      s1 = (Super) s2;        //3
   }
}
```

Which of the following statements are correct?

You had to select 1 option

- It will compile and run without any problems.
- It will compile but WILL throw ClassCastException at runtime.
- It will compile but MAY throw ClassCastException at runtime.
- It will not compile.
- None of the above.

Explanation:

Note that `s2` is a variable of class `Sub`, which is a subclass of `Super`. `s1` is a variable of class `Super`. A subclass can ALWAYS be assigned to a super class variable without any cast. It will always compile and run without any exception.

For example, a `Dog` "IS A" `Animal`, so you don't need to cast it.

But an `Animal` may not always be a `Dog`. So you need to cast it to make it compile and during the runtime the actual object referenced by `animal` should be a `Dog` otherwise it will throw a `ClassCastException`.

Problema 35

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

```
public class TestClass{
   public static void main(String args[ ] ){
      A o1 = new C( );
      B o2 = (B) o1;
      System.out.println(o1.m1( ) );     // 1
      System.out.println(o2.i );     // 2
   }
}
class A { int i = 10;  int m1( ) { return i; } }
class B extends A { int i = 20;  int m1() { return i; } }     // 2
class C extends B { int i = 30;  int m1() { return i; } }     // 1
```

You had to select 1 option

- The program will fail to compile.
- Class cast exception at runtime.
- It will print 30, 20.
- It will print 30, 30.
- It will print 20, 20.

Explanation:
Which variable (or static method) will be used depends on the class that the variable is declared of.
Which instance method will be used depends on the actual class of the object that is referenced by the variable.

So, in line `o1.m1()`, the actual class of the object is C, so C's `m1()` will be used. So it returns 30.

In line o2.i, o2 is declared to be of class B, so B's i is used. So it returns 20.

Problema 36

Which statements, when inserted in the code below, will cause an exception at run time?

```
class B {}
class B1 extends B {}
class B2 extends B {}
public class ExtendsTest{
  public static void main(String args[]){
     B b = new B();
     B1 b1 = new B1();
     B2 b2 = new B2();
     // insert statement here
  }
}
```

You had to select 1 option

- `b = b1;`
  - There won't be a problem anytime because B1 is a B
- `b2 = b;`
  - It fails at Compile time as an object referenced by b may not be a B2, so an explicit cast will be needed.
- `b1 = (B1) b;`
  - It `will pass at compile time` but `fail at run time` as the actual object referenced by b is not a B1.
  - By casting b to (B1), you are telling the compiler that the object referred to by b will be of class B1 at runtime. The compiler accepts that because it is possible for b to refer to an object of class B1 since B1 is-a B. However, at run time, b is pointing to an object of class B. B is not B1, so the JVM will throw a ClassCastException.
- `b2 = (B2) b1;`
  - It will not compile because b1 can never point to an object of class B2.
- `b1 = (B) b1;`
  - This won't compile. By casting b1 to B, you are telling the compiler that b1 points to an object of class B. But you are then trying to assign this reference to b1, which is of class B1. Compiler will complain against this assignment because there is no guarantee that an object of class B will also be of class B1. To be able to assign an object of class B to a reference of class B1, you need to confirm to the compiler that the reference will actually point to an object of class B1. Therefore, another cast is needed. i.e. b1 = (B1) (B) b1;

NO puede haber cast de hijo a padre, marcará bien la compilación, pero habrá error de ejecución

Problema 37

Given the following interface definition, which definitions are valid?

```
interface I1{
   void setValue(String s);
   String getValue();
}
```

You had to select 2 options

- ```
  class A extends I1{
     String s;
     void setValue(String val) { s = val; }
     String getValue() { return s; }
  }
  ```
  - Classes do not extend interfaces, they implement interfaces.

- ```
  interface I2 extends I1{
     void analyse();
  }
  ```

- ```
  abstract class B implements I1{
     int getValue(int i) { return 0; }
  }
  ```

- ```
  interface I3 implements I1{
     void perform_work();
  }
  ```
  - Interfaces do not implement anything, they can extend multiple interfaces.

Explanation:
Esta explicación va para la segunda opción correcta:
The `getValue(int i)` method of class B in option c, is different than the method defined in the interface because their parameters are different.
Therefore, this class does not actually implement the method of the interface and that is why it needs to be declared abstract. Further, they have "default" access whereas the interface methods are always public.

// Para el primer caso, las interfaces no pueden implementar interfaces, para heredar requiere de la palabra ‘extends’. Solo las clases pueden implementar interfaces.

Problema 38 ‘F’

Consider that you are writing a set of classes related to a new Data Transmission Protocol and have created your own exception hierarchy derived from java.lang.Exception as follows:

```
enthu.trans.ChannelException
              +-- enthu.trans.DataFloodingException,
                    enthu.trans.FrameCollisionException
```

You have a TransSocket class that has the following method:

```
long connect(String ipAddr) throws ChannelException
```

Now, you also want to write another "AdvancedTransSocket" class, derived from "TransSocket" which overrides the above mentioned method. Which of the following are valid declaration of the overriding method?

You had to select 2 options

- `int connect(String ipAddr) throws DataFloodingException`
  - The return type must match. Otherwise the method is OK.
- `int connect(String ipAddr) throws ChannelException`
  - The return type must match. Otherwise the method is OK.
- `long connect(String ipAddr) throws FrameCollisionException`
- `long connect(String ipAddr) throws Exception`
  - This option is invalid because Exception is a super class of ChannelException so it cannot be thrown by the overriding method.
- `long connect(String str)`

Explanation:
There are 2 important concepts involved here:
1. The overriding method must have same return type in case of primitives (a subclass is allowed in case of classes) (Therefore, the choices returning int are not valid.) and the parameter list must be the same (The name of the parameter does not matter, just the Type is important).

2. The overriding method can throw a subset of the exception or subclass of the exceptions thrown by the overridden class. Having no throws clause is also valid since an empty set is a valid subset.

Problema 39

Assume the following declarations:

```
class A{ }
class B extends A{ }
class C extends B{ }

class X{
   B getB(){ return new B(); }
}

class Y extends X{
  //method declaration here
}
```

Which of the following methods can be inserted in class Y?
You had to select 2 options

- `public C getB(){ return new B(); }`
  - Its return type is specified as C, but it is actually returning an object of type B. Since B is NOT a C, this will not compile.
- `protected B getB(){ return new C(); }`
  - Since C is-a B, this is valid. Also, an overriding method can be made less restrictive. protected is less restrictive than 'default' access.
- `C getB(){ return new C(); }`
  - Covariant returns are allowed in Java 1.5, which means that an overriding method can change the return type of the overridden method to a subclass of the original return type. Here, C is a subclass of B. So this is valid.
- `A getB(){ return new A(); }`
  - An overriding method cannot return a superclass object of the return type defined in the overridden method. A subclass is allowed in Java 1.5.

Problema 40 -> UP UP UP

What can be inserted in the code below so that it will print UP UP UP?

```
public class Speak {
    public static void main(String[] args) {
        Speak s = new GoodSpeak();

        INSERT CODE HERE

    }
}
                // Speak no le proporciona nada a GoodSpeak
class GoodSpeak extends Speak implements Tone{
    public void up(){
        System.out.println("UP UP UP");
    }
}
interface Tone{
    void up();
}
```

You had to select 2 options

- `((Tone)s).up();`
- `s.up();`
  - It will not compile because the class of reference `'s'` is `'Speak'`, which does not have the method up().
- `((GoodSpeak)s).up();`
- `(GoodSpeak)s.up();`
  - Incorrect syntax. It will not compile.
- `(Tone)(GoodSpeak)s.up();`
  - Incorrect syntax. It will not compile. The following would have been valid: ((Tone)(GoodSpeak)s).up();

Explanation:
// s.up() no jala porque no está en Speak, y no es un método @Override por lo que no puede acceder, requiere estar en la clase Speak también.
// En el caso de Tone, si puede porque el método lo tiene la interface Tone, es @Override, y ejecutará en modo del Objeto
// UN modo culero pero puede ser válido es:
`((Tone)(GoodSpeak)(Speak)s).up();`
Pasa por las 3 transformaciones, pero termina siendo `Tone`, y @override del método
// Puedes implementar varias interfaces separadas por coma `','`
// de 'extends' solo puede ser una clase ya sea abstracta o normal

Problema 41 -> this

What will the following program print when run?

```
// Filename: TestClass.java
public class TestClass{
   public static void main(String args[] ){ A b = new B("good bye");  }
}
class A{
   A() { this("hello", " world");  }
      // 3 Llama al de 1 String (Lo unió en un solo "String")
   A(String s) { System.out.println(s); }   //4 imprime "good bye world"
      // 2 Llega aquí por mandar 2 strings ( añadió el "world")
   A(String s1, String s2){ this(s1 + s2); }
}
class B extends A{
   B(){ super("good bye"); };
      // 1 Recibe aquí el "good bye"
   B(String s){ super(s, " world"); }
   B(String s1, String s2){ this(s1 + s2 + " ! "); }
}
```

You had to select 1 option

- It will print "good bye".
- It will print "hello world".
- It will print "good bye world".
- It will print "good bye" followed by "hello world".
- It will print "hello world" followed by "good bye".

Explanation:

`new B("good bye");` will call class B's one args constructor which in turn calls `super(s, " world");` (i.e. class A's two args constructor) which in turn calls `this(s1 + s2);` (i.e. class A's one arg constructor with parameter "good bye world") which prints it.

Problema 42 -> URL:IOR

What will be the output of the following program ?

```
class CorbaComponent{
    String ior;  // 6 "URL://IOR" al papá
                 // 3 manda al @Override
    CorbaComponent(){ startUp("IOR"); }
    void startUp(String s){ ior  =  s; }
    void print(){ System.out.println(ior); }
}

class OrderManager extends CorbaComponent{
   OrderManager(){  }  // 2 Constructor vacío tiene super
     // @Override de startup()
     // 4 pasa aquí "IOR"
   void startUp(String s){  ior = getIORFromURL(s);   }
                    // 5 regresa "URL://IOR" al papá,
                    // hijo no tiene 'ior'
   String getIORFromURL(String s){  return "URL://"+s; }
}

public class Application{
                                                        // 1
   public static void main(String args[]){ start(new OrderManager()); }
                          // 7 print de CorbataComponent
   static void start(CorbaComponent cc){ cc.print(); }
}
```

You had to select 1 option

- It will throw an exception at run time.
- It will print IOR
- It will print URL://IOR
- It will not compile.
- None of the above.

Explanation:

Observer that method `startUp(String s)` of `CorbaComponent` is overridden by the subclass `OrderManager`.

When an object of class `OrderManager` is constructed, the default no args constructor of `CorbaComponent` is called. This constructor calls the `startUp(String s)` with "IOR" as parameter. Now, there are two eligible methods which can be called - `CorbaComponent`'s `startUp` and `OrderManager`'s `startUp`.

The method selection is done on the basis of the actual class of the object (which is `OrderManager` here). So `OrderManager`'s `startUp` is called, which sets the `ior` variable to `URL://IOR`.

Unlike instance method selection, variable (and static method) selection is done on the basis of the declared class of the variable and not on the actual class of object that it is referring to.

Problema 43 -> Interface sin p2

Consider the following class and interface definitions (in separate files):

```
public class Sample implements IInt{
   public static void main(String[] args){
      Sample s = new Sample();  //1
      int j = s.thevalue;       //2
      int k = IInt.thevalue;    //3
      int l = thevalue;         //4
   }
}
public interface IInt{
      int thevalue = 0;
}
```

What will happen when the above code is compiled and run?
You had to select 1 option

- It will give an error at compile time at line //1.
- It will give an error at compile time at line //2.
- It will give an error at compile time at line //3
- It will give an error at compile time at line //4.
- It will compile and run without any problem.

Explanation:
As a rule, fields defined in an interface are public, static, and final. The methods are public.

Here, the interface `IInt` defines `thevalue` and thus any class that implements this interface gets this field.

Therefore, it can be accessed using `s.thevalue` or just `thevalue` inside the class. Also, since it is static, it can also be accessed using `IInt.thevalue` or `Sample.thevalue`.

Problema 44 -> si

What is the result of compiling and running the following code ?

```
public class TestClass{
   static int si = 10;
   public static void main (String args[]){
      new TestClass();
   }
   public TestClass(){
      System.out.println(this);
   }
   public String toString(){
      return "TestClass.si = "+this.si;
   }
}
```

You had to select 1 option

- The class will not compile because you cannot override toString() method.
    - You sure can. toString() is defined as public and non-final method in Object class.
- The class will not compile as si being static, this.si is not a valid statement.
    - static member can be accessed by static and non-static methods both. Non-static can only be accessed by non-static.
- It will print TestClass@nnnnnnnn, where nnnnnnnn is the hash code of the TestClass object referred to by 'this'.
    - It would have been correct if toString() were not overridden. This is the behavior of the toString() provided by Object class.
- It will print TestClass.si = 10
- None of the above.

Explanation:

The `toString` method for class Object returns a String consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:
`getClass().getName() + '@' + Integer.toHexString(hashCode())`

Problema 45 -> Barbie Doll

What can be inserted at //1 and //2 in the code below so that it can compile without errors:

```
class Doll{
    String name;
    Doll(String nm){
        this.name = nm;
    }
}

class Barbie extends Doll{
    Barbie(){
        //1 this(“algo”), delega al constructor
        // de Barbie que tenga un String y este debe crear el de Doll
        // Puede mandar al constructor de doll con super(“algo”)
    }
    Barbie(String nm){
        //2 para que haga el de Doll requiere mandar como super(nm)
    }
}
  public class TestClass {
       public static void main(String[] args) {
          Barbie b = new Barbie("mydoll");
       }
   }
```

You had to select 2 options

- this("unknown"); at 1 and super(nm); at 2
- super("unknown"); at 1 and super(nm); at 2
- super(); at 1 and super(nm); at 2
    - super(); at 1 will not compile because super class Doll does not have a no args constructor.
- super(); at 1 and Doll(nm); at 2
    - super(); at 1 will not compile because super class Doll does not have a no args constructor. Doll(nm); at 2 is an invalid syntax for calling the super class's constructor.
- super("unknown"); at 1 and this(nm); at 2
    - this(nm); at 2 will not compile because it is a recursive call to the same constructor.
- Doll(); at 1 and Doll(nm); at 2
- Both are using invalid syntax for calling the super class's constructor.

Explanation:

Since the super class `Doll` explicitly defines a constructor, compiler will not provide the default no-args constructor. Therefore, each of `Barbie`'s constructors must directly or indirectly call `Doll`'s string argument constructor, otherwise it will not compile.

Although not relevant for this question, it is interesting to know that `super(name);` at //1 or //2, would not be valid because `name` is defined in the superclass and so it cannot be used by a subclass until super class's constructor has executed. For the same reason, `this(name);` cannot be used either.

Problema 46 -> Interface PuStaFi

Consider the following interface definition:

```
interface Bozo{
          int type = 0;
          public void jump();
}
```

Now consider the following class:

```
public class Type1Bozo implements Bozo{
          public Type1Bozo(){
             type = 1;
          }

          public void jump(){
             System.out.println("jumping..."+type);
          }

          public static void main(String[] args){
             Bozo b = new Type1Bozo();
             b.jump();
          }
}
```

What will the program print when compiled and run?
You had to select 1 option

- jumping...0
- jumping...1
- This program will not compile.
- It will throw an exception at runtime.

Explanation:
Fields defined in an interface are ALWAYS considered as public, static, and final. Even if you don't explicitly define them as such.
In fact, you cannot even declare a field to be private or protected in an interface.

Therefore, you cannot assign any value to 'type' outside the interface definition.

Problema 47 -> correct @Override

Which of the given statements are correct for a method that overrides the following method:

```
public Set getSet(int a) {...}
```

Assume that Set is an interface and HashSet is a class that implements Set.

You had to select 3 options

- Its return type must be declared as Set.
    - Return type may also be a subclass/subinterface. So it can also return SortedSet, TreeSet, HashSet, or any other class/interface that implements/extends a Set.
- It may return HashSet.
  (Assume that HashSet implements Set) // Usa la covarianza en esta opcion
- It can declare any Exception in throws clause
    - Since the original (overridden) method does not have any throws clause, the overriding method cannot declare any checked exceptions.
- It can declare any RuntimeException in throws clause.
    - A method can throw any ‘unchecked exception’ (that is, an exception that extends java.lang.RuntimeException or java.lang.Error) even without declaring it in its throws clause.
    - The compiler doesn't check whether a given block of code throws such an exception and whether such an exception is properly handled. That is why such exceptions are called "unchecked" exceptions.
- It can be abstract.
    - Yes, you can make it abstract!! You would have to make the class as abstract as well though.

Explanation:
To override a method in the subclass, the overriding method (i.e. the one in the subclass) MUST HAVE:
.same name
.same return type in case of primitives (a subclass is allowed for classes, this is also known as covariant return types).
.same type and order of parameters
.it may throw only those exceptions that are declared in the throws clause of the superclass's method or exceptions that are subclasses of the declared exceptions. It may also choose NOT to throw any exception.
The names of the parameter types do not matter. For example, void `methodX(int i)` is same as void `methodX(int k)`

Problema 48 -> prevent @Overriding

Which of the following method definitions will prevent overriding of that method?

You had to select 4 options

- `public final void m1()`
  - final methods cannot be overridden or hidden. That is the purpose of final keyword.
- `public static void m1()`
- `public static final void m1()`
  - Keep in mind that static methods are never overridden, they may be hidden by a static method of a subclass with the same signature.
- `public abstract void m1()`
- `private void m1()`
  - private methods are not inherited at all so there is no question of overriding or hiding a private method. A subclass is free to have a static or instance method with the same signature as that of a private instance or static method of a super class. The subclass method will neither override nor hide the superclass's method in such a case.

Problema 49 -> null

Consider the following classes:

```
class A {
      public int getCode(){ return 2;}
}

class AA extends A {
  public void doStuff() {
  }
}
```

Given the following two declarations, which of the options will compile?

```
A a = null;
AA aa = null;
```

You had to select 4 options

- `a = (AA)aa;` // innecesario pero posible.
- `a = new AA();` // Posible asignar a un nuevo hijo.
- `aa = new A();` // No puede adjuntar a un padre.
- `aa = (AA) a;` // Compila, pero error en ejecución, al ser `'null'` No hay pede en ejecución.
- `aa = a;` // No compila, aun siendo `'null'` si pide el cast.
- `((AA)a).doStuff();` // Si jala, cast correcto para ingresar al método.

Problema 50

Consider the following code:

```
interface Bar{
    void bar();
}

abstract class FooBase{

  public static void bar(){
     System.out.println("In static bar");
  }
}

public class Foo extends FooBase implements Bar {



}
```

What can be done to the above code so that it will compile without any error?

You had to select 1 option

- Add this method in class Foo - public void bar(){ };
- Make the bar method in Bar interface default like this - default void bar() { }
- Either of the two approaches presented above will work.
- Neither of the two approaches presented above will work.
- Nothing needs to be done. It will compile as it is.

Explanation:

The problem with the code is that since `Foo` extends `FooBase`, `Foo` gets the static method `bar()` from `FooBase` in its scope and since `Foo` also says it implements `Bar` interface, it needs to have an instance method `bar()` with the same signature. This causes a conflict. A class cannot have two methods with the same signature in its scope where one is static and one is instance. Therefore, class `Foo` cannot be a subclass of `FooBase` and implement `Bar` at the same time.

Making the bar method in `Bar` interface a default method will not help either. Because even though class `Foo` will not need have a definition of the bar method in `Foo` class itself, it will inherit that method from the Bar interface and the same conflict will occur.

One way to fix the problem is to make the static `bar` method in class `FooBase` private. In this case, class `Foo` will not have this method in its scope and will therefore be free to implement an instance method with the same signature.

Problema 51

Which statements concerning the following code are true?

```
class A{
  public A() {} // A1
  public A(String s) {  this();  System.out.println("A :"+s);  }  // A2
}

class B extends A{
  public int B(String s) {  System.out.println("B :"+s);  return 0; } // B1
}
class C extends B{
    private C(){ super(); } // C1
    public C(String s){  this();  System.out.println("C :"+s);  } // C2
    public C(int i){} // C3
}
```

You had to select 4 options

- At least one of the constructors of each class is called as a result of constructing an object of class C.
    - To create any object one and only one constructor of that class and each of the super classes is called. (A constructor may as well delegate the construction to another constructor of the same class by calling this(...) as the first statement, just like calling a method.)
- Constructor at //A2 will never be called in creation of an object of class C.
    - Because B has no defined constructor and so a default no-argument constructor will be called, which will call the no-argument constructor of A
- Class C can be instantiated only in two ways by users of class C.
    - Since one constructor is private, users of this class can use only the other two public constructors from outside this class.
- //B1 will never be called in creation of objects of class A, B, or C.
    - Because //B1 is not a constructor. Note that it is returning an int. A constructor does not have any return type, not even void.
- None of the constructors of class A will be called in creation of an object of class C.
- None of the constructors of class B will be called in creation of an object of class C.

Problema 52

What will the following code print when run?

```
class A {
}

class AA extends A {
}


public class TestClass {
    public static void main(String[] args) throws Exception {
        A a = new A();
        AA aa = new AA();
        a = aa;
        System.out.println("a = "+a.getClass());
        System.out.println("aa = "+aa.getClass());
    }
}
```

You had to select 1 option

- It will not compile.
- It will throw ClassCastException at runtime.
- `a = class AA`
  `aa = class AA`
- `a = class A`
  `aa = class AA`

Explanation:
getClass is a public instance method in Object class. That means it is polymorphic. In other words, this method is bound at run time and so it returns the name of the class of the actual object to which the reference points.

Here, at run time, both - a and aa, point to an object of class AA. So both will print AA.