```
1   Persona desc = new Persona("Carlos");
2   Empleado emp = (Empleado)desc;
3   System.out.println( emp.getEmpresa() );
```

- Error de compilación
- Error de ejecución ✔
- Compila y ejecuta correctamente

Línea de error: 2 ✔

Resultado de la ejecución correcta: VACIO ✔

```
1   Persona desc = new Persona("Carlos");
2   if (desc instanceof Empleado) {
3     Empleado emp = (Empleado)desc;
4     System.out.println( emp.getEmpresa() );
5   } else
4     System.out.println( desc.getDatos() );
```

- Error de compilación
- Error de ejecución
- Compila y ejecuta correctamente ✔

Línea de error: VACIO ✔

Resultado de la ejecución correcta: Carlos ✔

Supongamos que el método getDatos() en la clase Persona ha sido definido de la siguiente forma

```
public final String getDatos() { return (nombre); }
```

y eliminada su implementación de las subclases. Dado el siguiente código:

```
1  Empleado emp = new Empleado("Carlos", "lavanderia");
2  Persona pers;
3  pers = emp;
4  System.out.println( emp.getDatos() );
```

- Error de compilación
- Error de ejecución
- Compila y ejecuta correctamente ✔

Línea de error: VACIO ✔

Resultado de la ejecución correcta: Carlos ✔

Supongamos que el método getDatos() en la clase Persona ha sido definido de la siguiente forma

```
public static String getDatos() { return (nombre); }
```

y eliminada su implementación de las subclases. Dado el siguiente código:

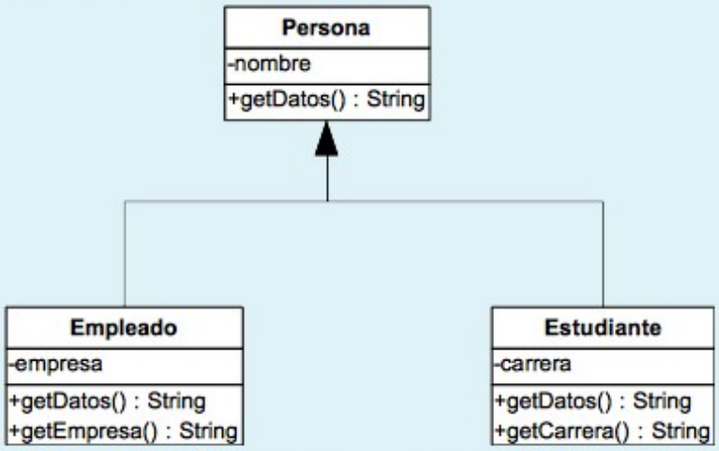
```
1  Empleado emp = new Empleado("Carlos", "lavanderia");
2  Persona pers;
3  pers = (Persona) emp;
4  System.out.println( pers.getDatos() );
```

- Error de compilación
- Error de ejecución
- Compila y ejecuta correctamente ✔

Línea de error: VACIO ✔

Resultado de la ejecución correcta: Carlos ✔

Dado el siguiente modelo



y parte de la implementación a continuación:

```
class Persona {
  public Persona(String n)    {nombre=n;}
  public String getDatos() {return nombre;}
    ...
  private String nombre;
}

class Empleado extends Persona {
  public Empleado(String n,String e)
  {  super(n); empresa=e; }
  public String getDatos()
  { return super.getDatos()+" trabaja en " + empresa; }
         ...
  private String empresa;
}

class Estudiante extends Persona {
  public Estudiante(String n,String c)
  { super(n); carrera=c; }
  public String getDatos()
  { return super.getDatos() + " estudia " + carrera; }
          ...
  private String carrera;
}
```

¿Cuál será el resultado de tratar de compilar y ejecutar los siguientes fragmentos de código? Indica si

- se produce un error de compilación
- se produce un error de ejecución
- compila y ejecuta correctamente

Si piensas que se produce algún error de compilación o de ejecución, indica en qué línea se produce el primer error (1 para la primera, 2 para la segunda, etc.). Si crees que se compila y ejecuta sin errores, indica el resultado que aparecerá en la salida estándar. Indica VACIO donde no proceda indicar respuesta.

```
1    Empleado empleado = new Empleado("Carlos","Lavandería");
2    Persona pers = new Persona("Juan");
3    empleado = pers;
4    System.out.println( empleado.getDatos() );
```

- (•) Error de compilación ✔
- ( ) Error de ejecución
- ( ) Compila y ejecuta correctamente

Línea de error: 3 ✔

Resultado de la ejecución correcta: VACIO ✔

```
1  Estudiante estudiante = null;
2  Persona pers = new Persona("José");
3  estudiante = pers;
5  System.out.println(  estudiante.getDatos() );
```

- (•) Error de compilación ✔
- ( ) Error de ejecución
- ( ) Compila y ejecuta correctamente

Línea de error: 3 ✔

Resultado de la ejecución correcta: VACIO ✔

```
1  Empleado emp = new Empleado("Carlos", "lavanderia");
2  Estudiante est = new Estudiante("Juan",”Derecho");
3  Persona pers;
4  pers = emp;
5  System.out.println( pers.getDatos() );
```

- Error de compilación
- Error de ejecución
- Compila y ejecuta correctamente ✔

Línea de error: VACIO ✔

Resultado de la ejecución correcta: Carlos trabaja en lavandería ✔

```
1   Empleado uno= new Empleado("Carlos", "lavanderia");
2   Persona desc = uno;
3   System.out.println( desc.getEmpresa() );
```

- Error de compilación ✔
- Error de ejecución
- Compila y ejecuta correctamente

Línea de error: 3 ✔

Resultado de la ejecución correcta: VACIO ✔

1. Cuando el método en la superclase y en la subclase tienen distinta signatura de tipo estamos hablando de: **Redefinición**
2. En Java, el tipo de enlace para el polimorfismo en herencia para métodos privados, finales, de clase y atributos es: **Estático**
3. En C++, el tipo de enlace por defecto para el polimorfismo en herencia para métodos virtuales es: **Dinámico**
4. El downcasting en Java siempre se especifica de forma: **Explícita**
5. Cuando el método de la subclase oculta el acceso al método de la superclase, estamos hablando de: **Shadowing**
6. En C++, donde las redefiniciones en las subclases ocultan a las definiciones de las superclases, se usa la estrategia de redefinición denominada: **Shadowing**
7. En Java, el tipo de enlace para el polimorfismo en herencia para métodos de instancia públicos y protegidos es: **Dinámico**
8. En Java, para que una clase no sea polimórfica debemos usar la palabra clave: **Final**
9. Las variables que pueden referenciar más de un tipo de objeto se denominan: **Polimórficas**
10. La sobrecarga de métodos de clase se resuelve en tiempo de: **Compilación**
11. En Java, si el método de la subclase se denomina igual que el de la superclase y tiene la misma signatura, usamos la anotación @Override, y por tanto es una: **Sobrescritura**
12. Las clases que tienen algún método con enlace dinámico se denominan: **Polimórficas**
13. Las llamadas a métodos sobrecargados se resuelven en tiempo de: **Compilación**
14. La sobrecarga de operadores es un ejemplo de sobrecarga basada en ...: **Signatura de tipos**
15. En C++, para que una clase sea polimórfica debe tener al menos un método: **Virtual**
16. Como alternativa a la sobrecarga, en C -p.ej. con la función printf-, o en Java -p.ej con un método en void F(int ... p)-, tenemos las funciones: **Poliádicas**
17. El mecanismo que permite conocer los tipos en tiempo de ejecución se denomina: **RTTI**
18. En Java, el tipo de enlace para el polimorfismo en herencia para métodos de instancia públicos y protegidos es: **Dinámico**
19. En Java, por defecto todas las clases son: **Polimórficas**
20. En Java, esta llamada usa: Object o = new String();: **Upcasting**
21. Las variables que pueden referenciar más de un tipo de objeto se denominan: **Polimórficas**
22. El método toString() en Java es un ejemplo de sobrecarga de métodos basada en ...: **Ámbito**
23. En Java, donde conviven las distintas redefiniciones que encontramos en la jerarquía de herencia, se usa la estrategia de redefinición denominada: **Mezcla**