Programación Orientada a Objetos
Enero 2010

Instrucciones del test:
Tiempo de realización: 20 minutos
Puntuación: 0.2 puntos por pregunta. Una pregunta mal resta una bien.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Si la frase es verdadera debes marcar **A**, y si es falsa **B**.

Modalidad 0

1. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base. FALSO
2. Una clase abstracta se caracteriza por no tener atributos. FALSO
3. La siguiente clase: `class S {public: virtual ~S()=0; virtual void f()=0;};` constituye una interfaz en C++. VERDADERO
4. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base. FALSO
5. Los métodos con enlace dinámico son abstractos. FALSO
6. Los constructores de las clases abstractas siempre son métodos abstractos. FALSO
7. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase. FALSO
8. Un metodo sobrecargado es aquel que tiene más de una implementación, diferenciando cada una por el ámbito en el que se declara, o por el número, orden y tipo de argumentos que admite. VERDADERO
9. Un método abstracto es un método con polimorfismo puro. FALSO
10. Todo espacio de nombres define su propio ámbito, distinto de cualquier otro ámbito. VERDADERO
11. En la sobrecarga de operadores binarios para objetos de una determinada clase, si se sobrecarga como función miembro, el operando de la izquierda siempre es un objeto de la clase. VERDADERO
12. La genericidad se considera una característica opcional de los lenguajes orientados a objetos VERDADERO
13. Hablamos de encapsulación cuando diferenciamos entre interfaz e implementación. VERDADERO
14. Una operación de clase no es una función miembro de la clase. FALSO
15. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos. VERDADERO
16. Los constructores siempre deben tener visibilidad pública. FALSO
17. En C++, si no se captura una excepción lanzada por un método, se produce un error de compilación. FALSO
18. En C++, la cláusula `throw()` tras la declaración de una función indica que ésta no lanza ninguna excepción. VERDADERO
19. Dada una clase genérica, no puede ser utilizada como clase base en herencia múltiple. FALSO
20. De una clase interfaz no se pueden crear instancias. De una clase abstracta sí. FALSO

Universidad de Alicante

Examen de Programación Orientada a Objetos

Ingeniería Informática
Ingeniería Técnica en Informática de Gestión
Ingeniería Técnica en Informática de Sistemas

15 de Enero de 2010

- *Tiempo de realización:* 1:20 horas.
- Cada ejercicio debe ir en un folio separado.
- Titulación, apellidos, nombre y DNI en todas las hojas.
- Los alumnos ERASMUS y aquéllos que estén en 5a o 6a convocatoria deben indicarlo en todas las hojas.
- Se deben justificar brevemente todas las respuestas.
- *Publicación solución examen:* **20 de Enero** en el CV
- *Publicación notas provisionales:* **1 de Febrero** en el CV
- *Revisión examen:* **5 de Febrero a las 16:00h**
  (en la Biblioteca de investigación del DLSI, Politécnica IV, módulo central, 2ª planta).
- *Publicación notas definitivas:* **6 de Febrero** en el CV

1. (1.5 puntos) Ilustra con un ejemplo en C++ cómo se declara y se define un atributo de clase público, así como las posibles formas de acceder a él.

Solución:

```
class A {
  public:
    static int a; // declaración
    void f();
//...};

int A::a=0; //definición
// Si el atributo es una constante de tipo int, char, float,...
// se puede inicializar en la declaración.

void A::f() {
  // Acceso directo dentro de una función miembro
  a=0;
}

int main() {
```

```
  // Acceso mediante del nombre de la clase
  A::a;
  // Acceso mediante un objeto de la clase (o una referencia o puntero)
  A obj; A& ref=obj;  A* p=&obj;
  obj.a; ref.a; p->a;
}
```

2. (1.5 puntos) Indica las principales ventajas del mecanismo de tratamiento de errores mediante excepciones respecto al mecanismo tradicional, basado en devolver un valor de error como resultado de un método al detectar un error. Ilustra tu respuesta con un ejemplo en C++.

Solución:

Las principales ventajas del tratamiento de errores mediante excepciones frente al mecanismo tradicional son las siguientes:

- Separa el flujo normal de una aplicación de las instrucciones de tratamiento de errores.
- Obliga al código cliente (quien invoca a los métodos que pueden detectar errores) a tratar (o ignorar) explícitamente el error. En caso contrario la aplicación abortará.
- Agrupar los tipos de errores y la diferenciación entre ellos.
- Detectar el error en un lugar (código ?servidor?) y tratarlo en otro diferente (código cliente).

Ejemplo:

```
//>>>>>>> Mecanismo tradicional de gestión de errores:
int f() { ... if (algo_va_mal) return -1; ... }

int g() {
... if (f()==-1) { //algo fue mal
    //tratar el error
  } else { /* seguir con normalidad */ } ...
  f(); // aquí simplemente se ignora un posible error de f().
}

//>>>>>> Tratamiento de errores mediante excepciones:
class Excepcion { ... };
int f() { ... throw Excepcion(); ... }

int g() {
  try {
    f(); // f() puede lanzar una excepcion
    ... // el código a continuación de la llamada no se ejecutará
```

```
      // si f() lanza una excepción
  } catch (Excepcion& ex) {
    // Aquí se trata el error producido en f();
  }

  f();
  // Si f() lanza una excepcion en esta llamada,
  // este programa abortará...
  ...
}
```

3. (1.5 puntos) Indica en qué consiste el enlace dinámico de métodos e ilústralo con un ejemplo en C++.

Solución:

La elección de qué método será el encargado de responder a un mensaje se realiza en tiempo de ejecución, en función del tipo del objeto al que hace referencia la variable mediante la cual se invoca al método, en el instante de la ejecución del mensaje.

```
class A {
  public:
    virtual void mostrar() { cout << "Soy A." << endl; }
    virtual ~A() {}
};

class B : public A {
  public:
    void mostrar() { cout << "Soy B." << endl; }
};

int main() {
  A* obj = new B();
  obj->mostrar(); // muestra 'Soy B.'
  delete obj; // llama a B::~B()
}
```

4. (1.5 puntos) Dados los ejemplos que aparecen en la figura 1, indica si se trata de un uso seguro o inseguro de la herencia y su denominación, explicando porqué es así.

Solución:

(A) Especialización *Uso seguro.* La clase derivada no modifica lo heredado de la clase base, sólo añade comportamiento. Se cumple el principio de sustitución.

(B) Especificación *Uso seguro.* La clase derivada especifica (o implementa, realiza) el interfaz de la clase base. Se cumple el principio de sustitución.

(C) Restricción *Uso inseguro.* No es una relación *es-un*, pues no todas las propiedades de la clase base sirven para la clase derivada (en concreto, el método Lapiz::afilar() no se puede aplicar a una estilográfica). No se cumple el principio de sustitución.

(D) Varianza *Uso inseguro.* También llamada herencia de conveniencia. No existe relación *es-un* entre las clases pero por comodidad decidimos adoptar la interfaz e implementación de la clase base como parte de la clase derivada, usando la herencia. En este caso, sería más correcto usar una relación de composición entre las clases.

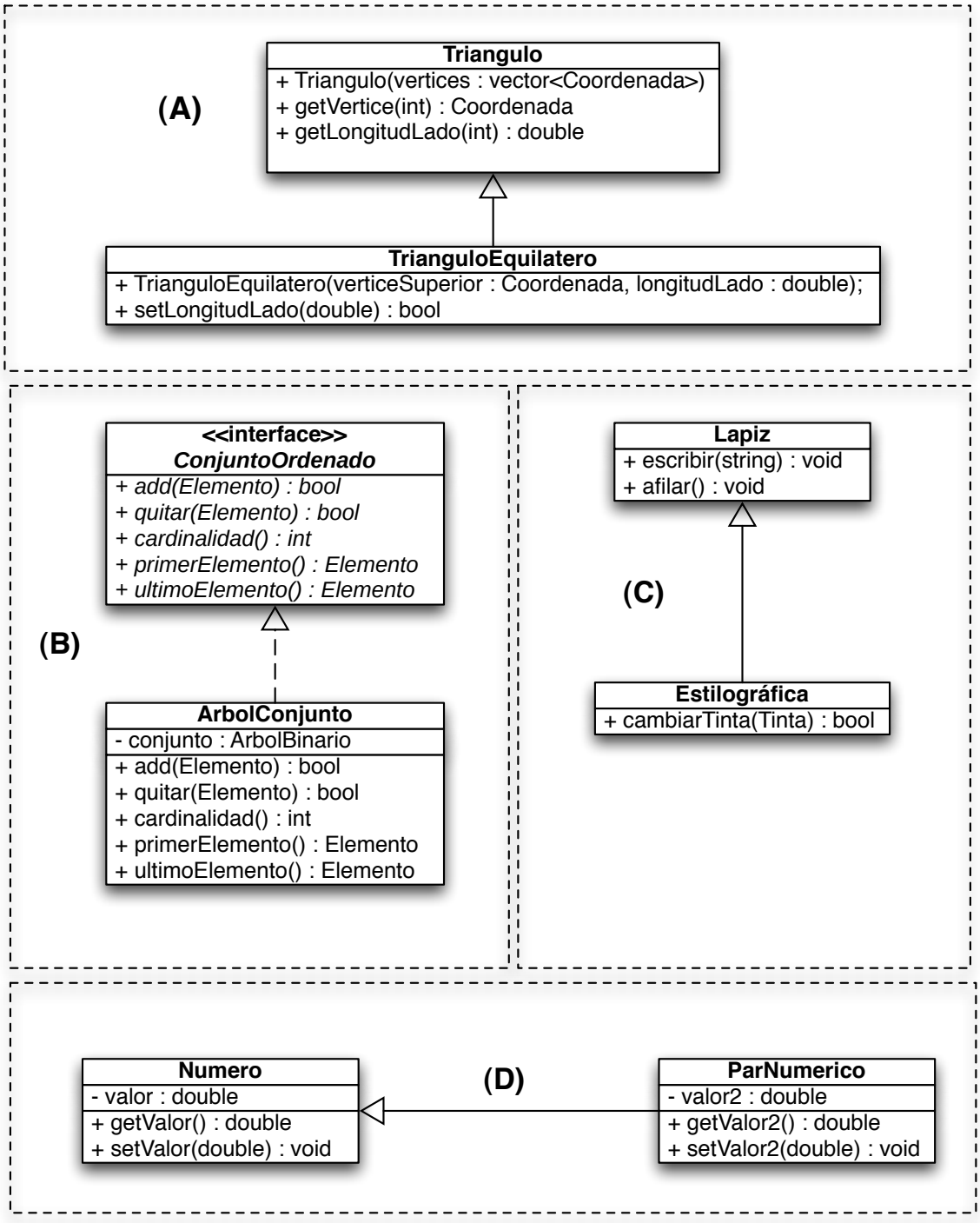


Figura 1: **Casos de herencia**

Programación Orientada a Objetos
Julio 2010

Instrucciones del test:
Tiempo de realización: 20 minutos
Puntuación: 0.2 puntos por pregunta. Una pregunta mal resta una bien.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Si la frase es verdadera debes marcar **A**, y si es falsa **B**.

Modalidad 0

1. La herencia protegida permite a los métodos de la clase derivada acceder a las propiedades privadas de la clase base. FALSO
2. Una clase abstracta se caracteriza por no tener definido ningún constructor. FALSO
3. La siguiente clase en C++: `class S {public: virtual ~S()=0;};` define una interfaz. VERDADERO
4. Los métodos virtuales son métodos abstractos. FALSO
5. Los métodos abstractos siempre tienen enlace dinámico. VERDADERO
6. Los constructores siempre tienen enlace dinámico. FALSO
7. En C++, un atributo de clase debe declararse dentro de la clase con el modificador `static`. VERDADERO
8. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica. FALSO
9. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica. FALSO
10. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A. FALSO
11. Dada la siguiente definición de clase en C++:

```
class TClase {
  public:
    TClase(int dim);
  private:    int var1;
};
```

    La instrucción `TClase c1;` no da error de compilación e invoca al constructor por defecto. FALSO
12. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. VERDADERO
13. Hablamos de encapsulación cuando agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos. VERDADERO
14. Una operación de clase sólo puede ser invocada mediante objetos constantes. FALSO
15. En C++ los constructores se pueden declarar como métodos virtuales. FALSO
16. En la misma clase, podemos definir constructores con distinta visibilidad. VERDADERO
17. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente. FALSO
18. En C++, es obligatorio especificar qué excepciones lanza una función mediante una cláusula `throw` tras la declaración de la función. FALSO
19. Dada una clase genérica, se pueden derivar de ella clases no genéricas. VERDADERO
20. Una clase interfaz no debe tener atributos de instancia. Una clase abstracta sí puede tenerlos. VERDADERO

Universidad de Alicante

Examen de Programación Orientada a Objetos

Ingeniería Informática
Ingeniería Técnica en Informática de Gestión
Ingeniería Técnica en Informática de Sistemas

16 de Julio de 2010

- *Tiempo de realización:* 1:30 horas.
- Cada ejercicio debe ir en un folio separado.
- Titulación, apellidos, nombre y DNI en todas las hojas.
- Los alumnos ERASMUS y aquéllos que estén en 5a o 6a convocatoria deben indicarlo en todas las hojas.
- Se deben justificar brevemente todas las respuestas.
- *Publicación solución examen:* **19 de Julio** en el CV.
- *Publicación notas provisionales:* **23 de Julio** en la web del DLSI.
- *Revisión examen:* **26 de Julio a las 12:00h**.
  (en la Biblioteca de investigación del DLSI, Politécnica IV, módulo central, 2ª planta).
- *Publicación notas definitivas:* **27 de Julio** en el CV.

1. (1.5 puntos) Muestra, con un ejemplo, cómo una misma excepción (el mismo objeto) puede ser capturada en diferentes lugares del código.

Solución:

```
void f() { ... throw exception(); ...}

void g() {
  try { ... f(); ... }
  catch (exception& ex) {
    // hacer algo...
    throw;  //relanzar ex
  }
}

void h() {
  try { ... g(); ... }
  catch (exception& ex) {
    // hacer algo...
  }
}
```

Otro ejemplo sería el de dos bloques try/catch anidados, donde el interno captura la excepción en primera instancia y la relanza hacia el try/catch más externo.

2. (1.5 puntos) Enuncia el principio de sustitución e ilústralo con un ejemplo en C++.

Solución:

> Debe ser posible utilizar cualquier objeto instancia de una clase derivada en el lugar de cualquier objeto instancia de su clase base sin que la semántica del programa escrito en los términos de la clase base se vea afectado.

```
class A {
  public:
    virtual void mostrar()
    { cout << "Soy A." << endl; }
    virtual ~A() {}
};

class B : public A {
  public:
    void mostrar()
    { cout << "Soy B." << endl; }
};

int main() {
  A* obj = new B();
  obj->mostrar(); // muestra 'Soy B.'
  delete obj; // llama a B::~B()
}
```

3. (1.5 puntos) ¿Qué significa que un método está sobrecargado? Ilustra tu respuesta con un ejemplo en C++.

Solución:

Que existen diferentes implementaciones del mismo método, donde todas tienen el mismo nombre, por ejemplo *sumar*, pero se distinguen entre ellas bien por el ámbito en el que se definen o bien por el número, orden y/o tipo de sus argumentos (parte izquierda de su signatura de tipo). Por ejemplo,

```
class Ejemplo{
public:
  int sumar(int);

  // diferente número de argumentos:
  int sumar(int, int);

  //double sumar(int);
```

```
  //Error: dentro de una misma clase dos métodos
  //no pueden diferir solamente en el tipo de retorno
};

// Versión de ámbito global
int sumar(int);
```

4. (1.5 puntos) Justifica por qué a menudo es necesario el uso de clases abstractas en un sistema. Ilustra tu respuesta con un ejemplo en C++.

Solución:

Uno de los objetivos de la programación orientada a objetos es el reuso de código. Una de las técnicas básicas de reuso en POO es la herencia. Cuando diferentes clases de objetos comparten ciertas características en común (atributos, métodos, relaciones con objetos de otras clases,...), una forma natural de agrupar dichas propiedades comunes es definir una clase base que las agrupe. Ocurre, sin embargo, que a menudo, a nivel de la clase base, ciertas características, por ejemplo, ciertas operaciones, pueden declararse pero no definirse, es decir, no existe (no la conocemos) una implementación de dichas operaciones a nivel de la clase base. A este tipo de clase, donde algunas operaciones se dejan sin implementar, se le llama clase abstracta, ya que declara métodos abstractos (sin implementación). Estos métodos serán implementados por las clases derivadas. He aquí un ejemplo, en C++, donde queremos definir una serie de formas geométricas que tiene ciertas cosas en común, por ejemplo, que se puede escribir (pintar) su representación en un flujo de salida (ostream):

```
class Forma {
  public:
    virtual void pintar(ostream&) = 0;
    // pintar() no se puede implementar aquí, pues no sabemos
    // que tipo de forma pintar ¿círculo, rectángulo, triángulo,...?
};

class Circulo : public Forma {
  double radio;
public:
  void pintar(ostream& os) {
  // sabemos cómo se representa un círculo
  // por tanto podremos implementar este método aquí
  }
};
```

De esta forma, podemos escribir código que trabaje con formas, sin preocuparnos de que tipo de formas concretas gestione la aplicación en una de sus ejecuciones.

```
Forma* f = new ... Circulo, Rectangulo, Triangulo,...
f->pintar(cout); // se ejecutará Circulo::pintar, o Rectangulo::pintar(),...
```

Examen de Programación Orientada a Objetos

Enero 2011

Test - Modalidad 0

Instrucciones del test:
Tiempo de realización: 20 minutos
Puntuación: 0.2 puntos por pregunta. Una pregunta mal resta una bien.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Si la frase es verdadera debes marcar **A**, y si es falsa **B**.

1. La herencia pública permite a los métodos definidos en una clase derivada acceder a las propiedades privadas de la clase base. FALSO
2. Una clase abstracta se caracteriza por declarar al menos un método abstracto. VERDADERO
3. La siguiente clase: `class S {public: Object *o;};` constituye una clase interfaz en C++. FALSO
4. Los métodos abstractos siempre tienen enlace dinámico. VERDADERO
5. Los constructores siempre son métodos virtuales. FALSO
6. Un método tiene polimorfismo puro cuando tiene como argumentos al menos una variable polimórfica. VERDADERO
7. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A. FALSO
8. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes. VERDADERO
9. Una operación de clase sólo puede ser invocada mediante objetos constantes. FALSO
10. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos. VERDADERO
11. Dada una clase genérica, no se pueden derivar de ella clases genéricas. FALSO
12. Una clase interfaz no puede tener instancias. VERDADERO
13. Una clase abstracta siempre tiene como clase base una clase interfaz. FALSO
14. No se puede definir un bloque *catch* sin su correspondiente bloque *try*. VERDADERO
15. Tras la ejecución de un bloque *catch*, termina la ejecución del programa. FALSO
16. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo. VERDADERO
17. El *downcasting* estático siempre es seguro. FALSO
18. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases. VERDADERO
19. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes. VERDADERO
20. Una operación de clase no puede tener enlace dinámico. VERDADERO

Universidad de Alicante

Examen de Programación Orientada a Objetos

Ingeniería Informática
Ingeniería Técnica en Informática de Gestión
Ingeniería Técnica en Informática de Sistemas

24 de Enero de 2011

- *Tiempo de realización:* 1 hora 30 minutos.
- Cada ejercicio debe ir en un folio separado.
- Titulación, apellidos, nombre y DNI en todas las hojas.
- Los alumnos ERASMUS y aquéllos que estén en 5a o 6a convocatoria deben indicarlo en todas las hojas.
- Se deben justificar brevemente todas las respuestas.
- *Publicación solución examen:* **10 de Febrero** en el CV.
- *Publicación notas provisionales:* **10 de Febrero** en la web del DLSI.
- *Revisión examen:* **14 de Febrero a las 10:00h**.
  (en la Biblioteca de investigación del DLSI, Politécnica IV, módulo central, 2ª planta).
- *Publicación notas definitivas:* **15 de Febrero** en el CV.

1. (2 puntos) Ilustra con un ejemplo en C++ cómo se declara y se define un atributo de clase público, así como las posibles formas de acceder a él.

Solución:

```
class A {
  public:
    static int a; // declaración
    void f();
//...};

int A::a=0; //definición. La inicializacíon es opcional.
// Si el atributo es una constante de tipo int, char, float,...
// se puede inicializar en la declaración.

void A::f() {
  // Acceso directo dentro de una función miembro
  a=0;
}

int main() {
  // Acceso mediante del nombre de la clase
  A::a;
  // Acceso mediante un objeto de la clase (o una referencia o puntero)
  A obj; A& ref=obj;  A* p=&obj;
```

```
  obj.a; ref.a; p->a;
}
```

2. (1 punto) Explica cuáles son las similitudes y/o diferencias entre un método abstracto y un método virtual.

Solución:

Ambos tipos de métodos

- tienen enlace dinámico,
- son métodos de instancia,

A diferencia de un método virtual,

- un método abstracto puede carecer de implementación.
- un método abstracto debe ser sobreescrito obligatoriamente en clases derivadas no abstractas.
- un método abstracto convierte a la clase en que se define en abstracta, impidiendo que se puedan crear instancias de esa clase.

(Otras diferencias menos relevantes, como la forma en que se declaran/definen en C++, también se han considerado respuestas correctas).

3. Estudia detenidamente el diagrama de clases de la figura 1, que representa una parte de un sistema de venta de artículos por catálogo. Teniendo en cuenta que se han omitido los constructores y el resto de la forma canónica de las clases,
   (a) (1.5 puntos) Escribe la declaración (.h) de la clase *LineaTicket*, incluyendo al menos un constructor y el resto de la forma canónica de la clase, declarando además los métodos que sean constantes como tales.

Solución:

```
class Articulo;

class LineaTicket
{
  public:
    LineaTicket(const Articulo& articulo, int cantidad);
    LineaTicket(const LineaTicket&);
    ~LineaTicket();
    LineaTicket& operator=(const LineaTicket&);
```

```
    int getCantidad() const;
    void setCantidad(int);
    Articulo* getArticulo() const;
    double calculaImporte() const;

  private:
    int cantidad;
    Articulo* articulo;

};
```

(b) (0.5 puntos) Escribe la declaración (.h) de la clase *ERoturaStock*, que se usará para lanzar una excepción cuando se soliciten más unidades de las disponibles en el stock, de forma que al ser lanzada guarde las unidades disponibles y las solicitadas por el usuario.

Solución:

```
class ERoturaStock {
  public:
    ERoturaStock(int cantidad=1, int unidades=0);
    ERoturaStock(const ERoturaStock&);
    ~ERoturaStock();
    ERoturaStock& operator=(const ERoturaStock&);
    int getUnidadesDisponibles() const;
    int getCantidadSolicitada() const;
  private:
    int unidadesDisponibles;
    int cantidadSolicitada;
};
```

(c) (1 punto) Implementa `Producto::vender(int cantidad) throw (ERoturaStock)` de forma que:

1. tras finalizar la operación, si todo ha ido bien, el nuevo stock sea *stock - cantidad*
2. se lance una excepción *ERoturaStock* si el stock resultante es negativo.

Solución:

```
void
Producto::vender(int cantidad) throw (ERoturaStock)
{
  if (stockActual < cantidad)
    throw ERoturaStock(cantidad, stockActual);

  stockActual -= cantidad;
}
```

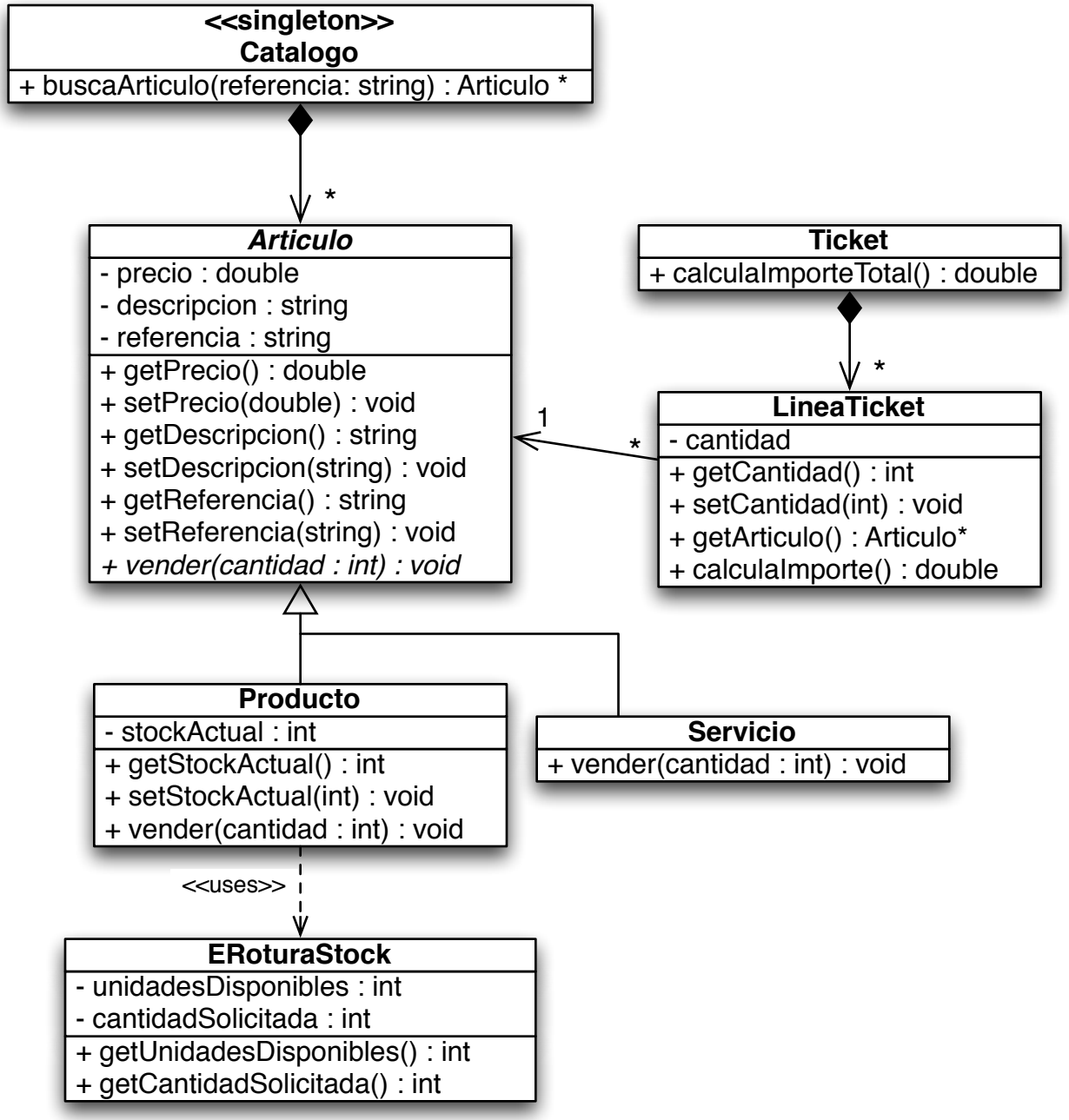


Figura 1: **Tickets**

Examen de Programación Orientada a Objetos
1 de Julio de 2011

Test - Modalidad 0

Instrucciones del test:
Tiempo de realización: 20 minutos
Puntuación: 0.2 puntos por pregunta. Una pregunta mal resta una bien.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Si la frase es verdadera debes marcar **A**, y si es falsa **B**.

1. Los métodos definidos en una clase derivada nunca pueden acceder a las propiedades privadas de una clase base. VERDADERO
2. Una clase abstracta se caracteriza por no tener definido ningún constructor. FALSO
3. La siguiente clase: `class S {public: S(); S(const S &s); virtual ~S();};` constituye una interfaz en C++. FALSO
4. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base. FALSO
5. Los métodos virtuales son métodos abstractos. FALSO
6. Los métodos abstractos son métodos con enlace dinámico. VERDADERO
7. Los constructores de las clases abstractas son métodos con enlace dinámico. FALSO
8. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase. FALSO
9. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica. FALSO
10. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica. FALSO
11. En C++ no podemos hacer sobrecarga de operadores para tipos predefinidos. VERDADERO
12. En C++, si no se define un constructor sin argumentos explícitamente, el compilador proporciona uno por defecto. FALSO
13. En la misma clase, podemos definir constructores con distinta visibilidad. VERDADERO
14. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente. FALSO
15. La instrucción `throw` permite lanzar como excepción cualquier tipo de dato. VERDADERO
16. Las funciones genéricas no se pueden sobrecargar. FALSO
17. Dada una clase genérica, se pueden derivar de ella clases no genéricas. VERDADERO
18. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor. FALSO
19. Una clase interfaz no puede tener atributos de instancia. Una clase abstracta sí puede tenerlos. VERDADERO
20. La herencia de interfaz se implementa mediante herencia pública. VERDADERO

Universidad de Alicante

Examen de Programación Orientada a Objetos

Ingeniería Informática
Ingeniería Técnica en Informática de Gestión
Ingeniería Técnica en Informática de Sistemas

1 de Julio de 2011

- *Tiempo de realización:* 1 hora 30 minutos.
- Cada apartado debe ir en un folio separado.
- Titulación, apellidos, nombre y DNI en todas las hojas.
- Los alumnos ERASMUS y aquéllos que estén en 5a o 6a convocatoria deben indicarlo en todas las hojas.
- Se deben justificar brevemente todas las respuestas.
- *Publicación solución examen:* **11 de Julio** en el CV.
- *Publicación notas provisionales:* **15 de Julio** en la web del DLSI.
- *Revisión examen:* **19 de Julio a las 10:00h**.
  (en la Biblioteca de investigación del DLSI, Politécnica IV, módulo central, 2ª planta).
- *Publicación notas definitivas:* **20 de Julio** en el CV.

1. Estudia detenidamente el diagrama de clases de la figura 1, que representa una parte de un sistema de venta de artículos por catálogo. Teniendo en cuenta que se han omitido los constructores y el resto de la forma canónica de las clases, y que el método *vender()* se ocupa de mantener actualizado el stock de productos,

   (a) (1.5 puntos) Escribe la declaración (.h) de la clase *Articulo*, incluyendo al menos un constructor, el constructor de copia y el resto de la forma canónica de la clase y declarando los métodos que sean constantes como tales.

Solución:

```
// includes de sistema...
using namespace std;

class Articulo {
public:
Articulo();
virtual ~Articulo();
Articulo(const Articulo&);
Articulo* operator=(const Articulo&);

double getPrecio() const;
void setPrecio(double);

string getDescripcion() const;
void setDescripcion(string);
```

```
string getReferencia() const;
void setReferencia(string);

virtual void vender(int cantidad) = 0;

protected:
double precio;
string descripcion;
string referencia;

};
```

(b) (1 punto) *LineaTicket* tiene un constructor sobrecargado. Impleméntalo, así como el constructor de copia y el resto de la forma canónica de la clase.

Solución:

```
LineaTicket::LineaTicket(Articulo& art, int ctd)
: articulo(&art), cantidad(ctd) {
  if (cantidad<0)
    cantidad=0;
}

LineaTicket::LineaTicket(const LineaTicket& t)
: articulo(t.articulo), cantidad(t.cantidad) {
}

LineaTicket::~LineaTicket() {
  articulo=NULL;
  cantidad=0;
}

LineaTicket&
LineaTicket::operator=(const LineaTicket& t) {
  if (this != &t) {
    articulo=t.articulo;
    cantidad=t.cantidad;
  }
  return *this;
}
```

(c) (1.5 puntos) Escribe la declaración (.h) de la clase *Catalogo*, incluyendo al menos un constructor.

Solución:

```
// includes de sistema...
```

```
using namespace std;
#include "EArticuloNoExiste.h"
#include "Articulo.h" // o declaración 'forward'

class Catalogo {

  public:
    static Catalogo* getInstancia();
    Articulo* buscaArticulo(string referencia) const
        throw (EArticuloNoExiste);
    ~Catalogo();

  private:
    Catalogo();

    static Catalogo* instancia;
    vector<Articulo*> articulos;
};
```

(d) (2 puntos) Implementa (.cc) la clase *Catalogo*, incluyendo el constructor declarado en el apartado anterior, teniendo en cuenta que el método *buscaArticulo()* lanza una excepción de tipo *EArticuloNoExiste* si no encuentra el artículo pedido.

Solución:

```
// includes de sistema...
using namespace std;
#include "Catalogo.h"

Catalogo* Catalogo::instancia=NULL;

Catalogo::Catalogo() : articulos() {}

Catalogo::~Catalogo() {
  for (int i=0; i<articulos.size(): i++)
    delete articulos[i];
  //opcional:
  articulos.clear();
}

Catalogo* Catalogo::getInstancia() {
  if (instancia==NULL)
    instancia = new Catalogo();
    // o cualquier otro constructor apropiado
  return instancia;
```

```
}

Articulo* Catalogo::buscaArticulo(string referencia)
  const throw (EArticuloNoExiste) {
  vector<Articulo*>::const_iterator it = articulos.begin();

  while (it != articulos.end()) {
    if ((*it)->getReferencia() == referencia)
      return *it;
    it++;
  }
  throw EArticuloNoExiste(referencia);
  // Suponemos la existencia de un constructor EArticuloNoExiste(string).
}
```

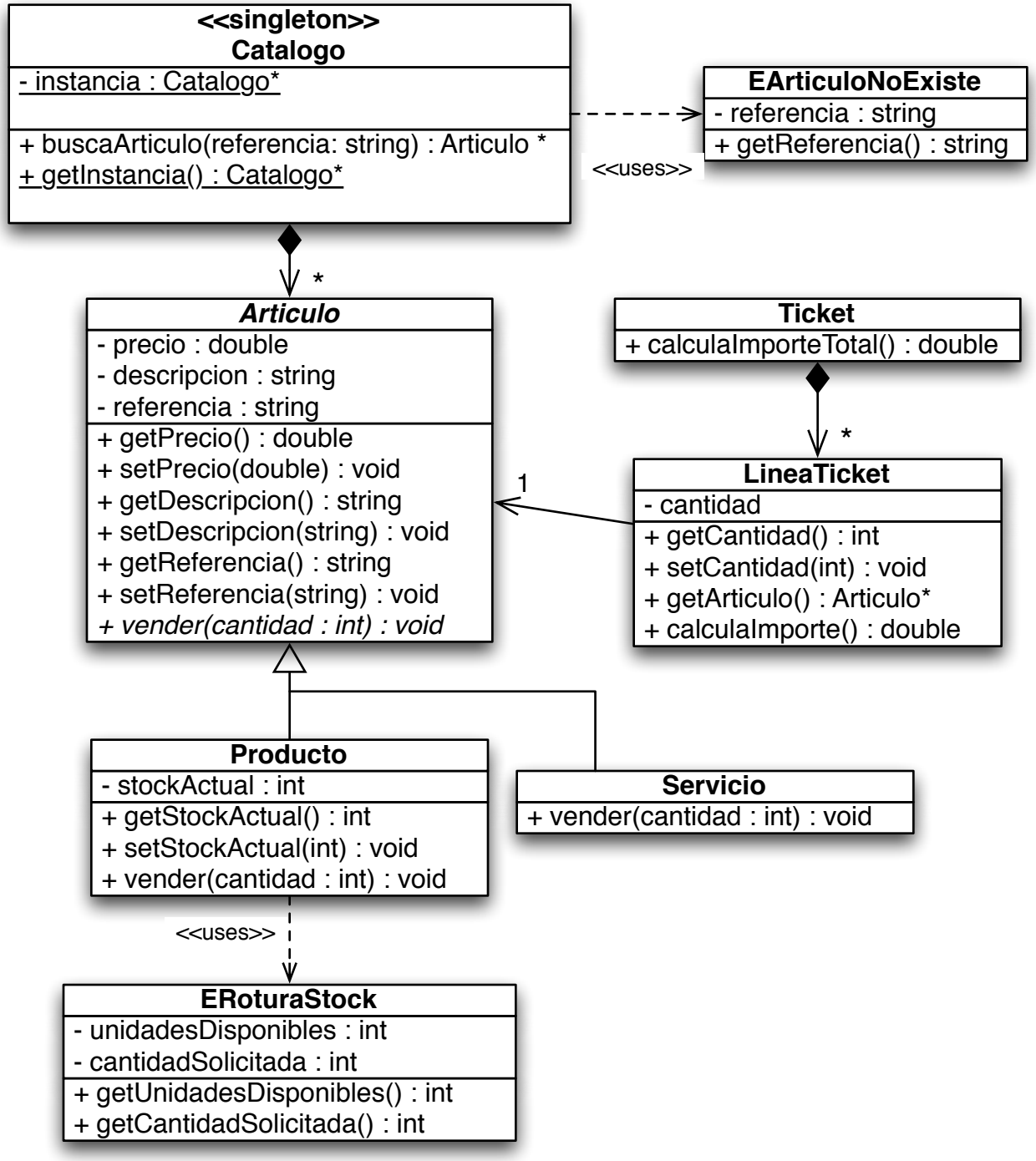


Figura 1: **Tickets**

Programación Orientada a Objetos

Enero 2009

Instrucciones del test:
Tiempo de realización: una hora.
Puntuación: 0.4 puntos por pregunta. 3 preguntas mal restan una bien. Sólo una opción es la correcta.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Señala las respuestas correctas.

Modalidad 0

1. ¿Qué tipo de herencia en C++ permite a una clase derivada acceder a los métodos privados de una de sus clases base?
   A. Protegida o privada, pero no pública.
   B. Pública.
   C. Privada.
   D. Ninguna de las anteriores.
2. ¿Qué caracteriza a una clase abstracta en C++?
   A. Que no tiene atributos.
   B. Que no tiene definido ningún constructor.
   C. Que tiene al menos un método virtual.
   D. Ninguna de las anteriores.
3. ¿Cuál de las siguientes clases constituye una interfaz en C++?
   A. `class S {public: Object *o;};`
   B. `class S {public: S(); S(const S &s); virtual ~S();};`
   C. `class S {void f()=0;};`
   D. `class S {public: virtual void f()=0;};`
4. ¿Puede invocarse desde un método de una clase derivada un método con idéntica signatura de una de sus superclases?
   A. Nunca; de hecho, así es como funciona la sobrecarga.
   B. En general, siempre podrá invocarse dicho método mediante una sintaxis que depende del lenguaje de programación.
   C. Solo si el método de la superclase es virtual.
   D. Solo si el método de la clase derivada es virtual.
5. Dada una implementación correcta de la práctica 2, ¿qué cambios adicionales habría que realizar en los métodos de las clases `MenuSinSal` y `MenuCompleto` para que sigan comportándose como se pide en el enunciado, si declaráramos como protegidos los atributos de la clase `Menu`?
   A. Ningún cambio: el código seguiría funcionando sin problemas.
   B. Sería necesario cambiar a protegida la herencia de las clases `MenuSinSal` y `MenuCompleto`.
   C. Sería necesario reescribir algunos métodos.
   D. Ninguna de las anteriores.
6. Basándonos en el diagrama de clases adjunto y el siguiente código:

```
Titular p; p.toString();
```

A. Se invoca a `Titular::toString()` mediante enlace dinámico.

B. Se invoca únicamente a `Persona::toString()`.

C. Se invoca a `Titular::toString()` mediante enlace estático.

D. Ninguna de las anteriores.

7. Elige la opción correcta:

A. Los métodos virtuales tienen que ser abstractos.

B. Los métodos abstractos tienen que ser virtuales.

C. Los métodos puros pero no abstractos tienen que ser virtuales.

D. Ninguna de las anteriores es correcta.

8. Elige la opción correcta:

A. Los constructores de las clases abstractas tienen que ser virtuales.

B. Los constructores siempre son abstractos.

C. Los constructores virtuales son siempre abstractos.

D. Ninguna de las anteriores.

9. Basándonos en el diagrama de clases adjunto:

```
Menu *m = new MenuCompleto("P", "S");
Menu *m2 = new MenuSinSal("P1", "S1");
MenuSinSal *m3 = m2->clonar();
```

A. El código anterior compila perfectamente.

B. El código anterior necesita de un downcasting para poder compilar.

C. El código anterior necesita de un upcasting para poder compilar.

D. Ninguna de las anteriores.

10. Supongamos el siguiente código:

```
class A { public: void F() {std::cout << "A"; } };
class B : public A { public: virtual void F() {std::cout << "B"; } };
class C : public B { public: void F() {std::cout << "C"; } };

main() {  A* c = new C;  c->F();  delete c; c=NULL; }
```

A. Se imprime A por pantalla;

B. Se imprime **B** por pantalla;

C. Se imprime **C** por pantalla;

D. Se imprime **ABC** por pantalla;

11. Señala la respuesta correcta:

A. Un atributo de clase ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.

B. Un atributo de clase debe declararse dentro de la clase con el modificador `const`.

C. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.

D. Ninguna de las anteriores.

12. Supongamos que hemos implementado una clase que se llama Linea. Si queremos que funcionen las siguientes líneas de código:

```
Linea l1;  double d = l1;
```

A. Implementaremos la sobrecarga del operador = para la clase Linea.

B. Implementaremos la sobrecarga del operador = para la clase double.

C. Implementaremos la sobrecarga de conversion a double en la clase Linea.

D. Todas las anteriores son ciertas.

13. Las funciones sobrecargadas son aquéllas que:

A. Se les puede pasar diferente cantidad de argumentos, dependiendo lo que nos interese sin que para ello debamos implementar cada caso.

B. Tienen más de una implementación diferenciando cada una por el número, orden y tipo de argumentos.

C. Tienen más de una implementación que, en general, pueden distinguirse únicamente por el tipo devuelto.

D. Reciben como argumento al menos una variable polimórfica.

14. Cuando sobrecargamos un operador:

A. Podemos hacerlo incluso para los tipos predefinidos.

B. Sólo podemos sobrecargar operadores para tipos definidos por el usuario. Debemos respetar las reglas de asociatividad y precedencia y podemos crear nuevos operadores.

C. No podemos hacer sobrecarga de tipos predefinidos. No podemos crear nuevos operadores. Debemos respetar las reglas de asociatividad y precedencia.

D. Ninguna de las anteriores.

15. ¿Qué implicaciones tiene declarar un dato miembro de una clase como privado?

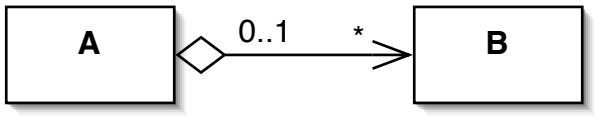
A. Indica que ese atributo sólo es accesible por las funciones de ese módulo de programa.

B. Indica que sólo lo puede usar la función donde se declara.

C. Indica que sólo se puede acceder a ese atributo o método desde las funciones miembro de la clase donde se declara.

D. Ninguna de las antetiores.

16. Dado el diagrama UML:



A. Refleja la existencia de una relación todo-parte en la que el objeto compuesto maneja la creación/destrucción de los objetos componente.

B. Refleja la existencia de una asociación binaria unidireccional donde los objetos de tipo A no están necesariamente asociados a algún B.

C. Refleja la existencia de una relación todo-parte en la que la existencia del objeto parte no depende de la existencia del objeto todo que la contiene.

D. Al implementarlo en C++, la clase A tiene que destruir la memoria de los objetos B que dependan de ella, para que no quede memoria sin liberar.

17. Hablamos de encapsulación cuando:

A. Omitimos ciertos detalles de implementación.

B. Diferenciamos entre interfaz e implementación.

C. Agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos.

D. B y C son correctas.

18. Las operaciones de clase:
    A. Sólo pueden manejar atributos de clase.
    B. No son funciones miembro de la clase.
    C. Sólo pueden ser invocadas por objetos constantes.
    D. Ninguna de las anteriores.

19. En cuanto a los constructores:
    A. En C++, si no se define un constructor sin parámetros explícitamente, el compilador proporciona uno por defecto.
    B. Son funciones miembro constantes.
    C. Siempre tienen visibilidad pública.
    D. Ninguna de las anteriores.

20. Los atributos de instancia, si son constantes:
    A. Se les asigna su valor inicial directamente cuando se declaran.
    B. Se les asigna su valor inicial en el cuerpo del constructor.
    C. Se les asigna su valor inicial mediante inicializadores en el constructor.
    D. Se les asigna su valor inicial fuera de la clase.

21. Las excepciones
    A. Si se ignoran, se produce un error de compilación.
    B. Si se ignoran, el compilador genera mensajes de advertencia (*warnings*).
    C. Si se ignoran, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente.
    D. Ninguna de las anteriores.

22. La instrucción `throw`
    A. sólo permite lanzar objetos de clase `exception` o de clases derivadas de ella.
    B. permite lanzar como excepción cualquier tipo de dato.
    C. no se utiliza nunca sin especificar qué excepción debe lanzar.
    D. no se puede usar dentro de un bloque `catch`.

23. En C++,
    A. es obligatorio especificar qué excepciones lanza una función mediante una cláusula `throw` tras la declaración de la función.
    B. la cláusula `throw()` tras la declaración de una función indica que ésta no lanza ninguna excepción.
    C. una excepción de usuario es una clase definida por el usuario que deriva de la clase `exception`.
    D. Ninguna de las anteriores.

24. Las funciones genéricas
    A. No se pueden sobrecargar ni sobreescribir.
    B. No se pueden sobreescribir pero sí sobrecargar.
    C. No se pueden sobrecargar pero sí sobreescribir.
    D. Ninguna de las anteriores.

25. Dada una clase genérica
    A. No se pueden derivar de ella clases genéricas.
    B. No se pueden derivar de ella clases no genéricas.
    C. Puede ser utilizada como clase base en herencia múltiple.
    D. Ninguna de las anteriores.

Programación Orientada a Objetos

Enero 2009

Instrucciones del test:
Tiempo de realización: una hora.
Puntuación: 0.4 puntos por pregunta. 3 preguntas mal restan una bien. Sólo una opción es la correcta.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Señala las respuestas correctas.

Modalidad 1

1. ¿Puede invocarse desde un método de una clase derivada un método con idéntica signatura de una de sus superclases?
   A. Nunca; de hecho, así es como funciona la sobrecarga.
   B. En general, siempre podrá invocarse dicho método mediante una sintaxis que depende del lenguaje de programación.
   C. Solo si el método de la superclase es virtual.
   D. Solo si el método de la clase derivada es virtual.

2. Dada una implementación correcta de la práctica 2, ¿qué cambios adicionales habría que realizar en los métodos de las clases `MenuSinSal` y `MenuCompleto` para que sigan comportándose como se pide en el enunciado, si declaráramos como protegidos los atributos de la clase `Menu`?
   A. Ningún cambio: el código seguiría funcionando sin problemas.
   B. Sería necesario cambiar a protegida la herencia de las clases `MenuSinSal` y `MenuCompleto`.
   C. Sería necesario reescribir algunos métodos.
   D. Ninguna de las anteriores.

3. Basándonos en el diagrama de clases adjunto y el siguiente código:

   ```
   Titular p; p.toString();
   ```

   A. Se invoca a `Titular::toString()` mediante enlace dinámico.
   B. Se invoca únicamente a `Persona::toString()`.
   C. Se invoca a `Titular::toString()` mediante enlace estático.
   D. Ninguna de las anteriores.

4. ¿Qué tipo de herencia en C++ permite a una clase derivada acceder a los métodos privados de una de sus clases base?
   A. Protegida o privada, pero no pública.
   B. Pública.
   C. Privada.
   D. Ninguna de las anteriores.

5. ¿Qué caracteriza a una clase abstracta en C++?
   A. Que no tiene atributos.
   B. Que no tiene definido ningún constructor.
   C. Que tiene al menos un método virtual.
   D. Ninguna de las anteriores.

6. ¿Cuál de las siguientes clases constituye una interfaz en C++?

A. `class S {public: Object *o;};`

B. `class S {public: S(); S(const S &s); virtual ~S();};`

C. `class S {void f()=0;};`

D. `class S {public: virtual void f()=0;};`

7. Supongamos el siguiente código:

```
class A { public: void F() {std::cout << "A"; } };
class B : public A { public: virtual void F() {std::cout << "B"; } };
class C : public B { public: void F() {std::cout << "C"; } };

main() {  A* c = new C;  c->F();  delete c; c=NULL; }
```

A. Se imprime A por pantalla;

B. Se imprime **B** por pantalla;

C. Se imprime **C** por pantalla;

D. Se imprime **ABC** por pantalla;

8. Señala la respuesta correcta:

A. Un atributo de clase ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.

B. Un atributo de clase debe declararse dentro de la clase con el modificador `const`.

C. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.

D. Ninguna de las anteriores.

9. Supongamos que hemos implementado una clase que se llama Linea. Si queremos que funcionen las siguientes líneas de código:

```
Linea l1;  double d = l1;
```

A. Implementaremos la sobrecarga del operador = para la clase Linea.

B. Implementaremos la sobrecarga del operador = para la clase double.

C. Implementaremos la sobrecarga de conversion a double en la clase Linea.

D. Todas las anteriores son ciertas.

10. Elige la opción correcta:

A. Los métodos virtuales tienen que ser abstractos.

B. Los métodos abstractos tienen que ser virtuales.

C. Los métodos puros pero no abstractos tienen que ser virtuales.

D. Ninguna de las anteriores es correcta.

11. Elige la opción correcta:

A. Los constructores de las clases abstractas tienen que ser virtuales.

B. Los constructores siempre son abstractos.

C. Los constructores virtuales son siempre abstractos.

D. Ninguna de las anteriores.

12. Basándonos en el diagrama de clases adjunto:

```
Menu *m = new MenuCompleto("P", "S");
Menu *m2 = new MenuSinSal("P1", "S1");
MenuSinSal *m3 = m2->clonar();
```

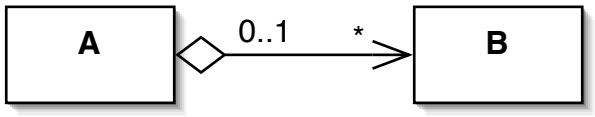
A. El código anterior compila perfectamente.

B. El código anterior necesita de un downcasting para poder compilar.

C. El código anterior necesita de un upcasting para poder compilar.

D. Ninguna de las anteriores.

13. Dado el diagrama UML:



A. Refleja la existencia de una relación todo-parte en la que el objeto compuesto maneja la creación/destrucción de los objetos componente.

B. Refleja la existencia de una asociación binaria unidireccional donde los objetos de tipo A no están necesariamente asociados a algún B.

C. Refleja la existencia de una relación todo-parte en la que la existencia del objeto parte no depende de la existencia del objeto todo que la contiene.

D. Al implementarlo en C++, la clase A tiene que destruir la memoria de los objetos B que dependan de ella, para que no quede memoria sin liberar.

14. Hablamos de encapsulación cuando:

A. Omitimos ciertos detalles de implementación.

B. Diferenciamos entre interfaz e implementación.

C. Agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos.

D. B y C son correctas.

15. Las operaciones de clase:

A. Sólo pueden manejar atributos de clase.

B. No son funciones miembro de la clase.

C. Sólo pueden ser invocadas por objetos constantes.

D. Ninguna de las anteriores.

16. Las funciones sobrecargadas son aquéllas que:

A. Se les puede pasar diferente cantidad de argumentos, dependiendo lo que nos interese sin que para ello debamos implementar cada caso.

B. Tienen más de una implementación diferenciando cada una por el número, orden y tipo de argumentos.

C. Tienen más de una implementación que, en general, pueden distinguirse únicamente por el tipo devuelto.

D. Reciben como argumento al menos una variable polimórfica.

17. Cuando sobrecargamos un operador:

A. Podemos hacerlo incluso para los tipos predefinidos.

B. Sólo podemos sobrecargar operadores para tipos definidos por el usuario. Debemos respetar las reglas de asociatividad y precedencia y podemos crear nuevos operadores.

C. No podemos hacer sobrecarga de tipos predefinidos. No podemos crear nuevos operadores. Debemos respetar las reglas de asociatividad y precedencia.

D. Ninguna de las anteriores.

18. ¿Qué implicaciones tiene declarar un dato miembro de una clase como private?

A. Indica que ese atributo sólo es accesible por las funciones de ese módulo de programa.

B. Indica que sólo lo puede usar la función donde se declara.

C. Indica que sólo se puede acceder a ese atributo o método desde las funciones miembro de la clase donde se declara.

D. Ninguna de las antetiores.

19. La instrucción `throw`

A. sólo permite lanzar objetos de clase `exception` o de clases derivadas de ella.

B. permite lanzar como excepción cualquier tipo de dato.

C. no se utiliza nunca sin especificar qué excepción debe lanzar.

D. no se puede usar dentro de un bloque `catch`.

20. En C++,

A. es obligatorio especificar qué excepciones lanza una función mediante una cláusula `throw` tras la declaración de la función.

B. la cláusula `throw()` tras la declaración de una función indica que ésta no lanza ninguna excepción.

C. una excepción de usuario es una clase definida por el usuario que deriva de la clase `exception`.

D. Ninguna de las anteriores.

21. Las funciones genéricas

A. No se pueden sobrecargar ni sobreescribir.

B. No se pueden sobreescribir pero sí sobrecargar.

C. No se pueden sobrecargar pero sí sobreescribir.

D. Ninguna de las anteriores.

22. En cuanto a los constructores:

A. En C++, si no se define un constructor sin parámetros explícitamente, el compilador proporciona uno por defecto.

B. Son funciones miembro constantes.

C. Siempre tienen visibilidad pública.

D. Ninguna de las anteriores.

23. Los atributos de instancia, si son constantes:

A. Se les asigna su valor inicial directamente cuando se declaran.

B. Se les asigna su valor inicial en el cuerpo del constructor.

C. Se les asigna su valor inicial mediante inicializadores en el constructor.

D. Se les asigna su valor inicial fuera de la clase.

24. Las excepciones

A. Si se ignoran, se produce un error de compilación.

B. Si se ignoran, el compilador genera mensajes de advertencia (*warnings*).

C. Si se ignoran, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente.

D. Ninguna de las anteriores.

25. Dada una clase genérica

A. No se pueden derivar de ella clases genéricas.

B. No se pueden derivar de ella clases no genéricas.

C. Puede ser utilizada como clase base en herencia múltiple.

D. Ninguna de las anteriores.

Programación Orientada a Objetos

Enero 2009

Instrucciones del test:
Tiempo de realización: una hora.
Puntuación: 0.4 puntos por pregunta. 3 preguntas mal restan una bien. Sólo una opción es la correcta.
En la *hoja de respuestas*, no olvides rellenar los datos y firma. Indica D.N.I y modalidad.
Señala las respuestas correctas.

Modalidad 2

1. Supongamos el siguiente código:

```
class A { public: void F() {std::cout << "A"; } };
class B : public A { public: virtual void F() {std::cout << "B"; } };
class C : public B { public: void F() {std::cout << "C"; } };

main() {  A* c = new C;  c->F();  delete c; c=NULL; }
```

   A. Se imprime A por pantalla;
   B. Se imprime **B** por pantalla;
   C. Se imprime **C** por pantalla;
   D. Se imprime **ABC** por pantalla;

2. Señala la respuesta correcta:
   A. Un atributo de clase ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.
   B. Un atributo de clase debe declararse dentro de la clase con el modificador `const`.
   C. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
   D. Ninguna de las anteriores.

3. Supongamos que hemos implementado una clase que se llama Linea. Si queremos que funcionen las siguientes líneas de código:

```
Linea l1;  double d = l1;
```

   A. Implementaremos la sobrecarga del operador = para la clase Linea.
   B. Implementaremos la sobrecarga del operador = para la clase double.
   C. Implementaremos la sobrecarga de conversion a double en la clase Linea.
   D. Todas las anteriores son ciertas.

4. Elige la opción correcta:
   A. Los métodos virtuales tienen que ser abstractos.
   B. Los métodos abstractos tienen que ser virtuales.
   C. Los métodos puros pero no abstractos tienen que ser virtuales.
   D. Ninguna de las anteriores es correcta.

5. Elige la opción correcta:
   A. Los constructores de las clases abstractas tienen que ser virtuales.
   B. Los constructores siempre son abstractos.

C. Los constructores virtuales son siempre abstractos.

D. Ninguna de las anteriores.

6. Basándonos en el diagrama de clases adjunto:

```
Menu *m = new MenuCompleto("P", "S");
Menu *m2 = new MenuSinSal("P1", "S1");
MenuSinSal *m3 = m2->clonar();
```

A. El código anterior compila perfectamente.

B. El código anterior necesita de un downcasting para poder compilar.

C. El código anterior necesita de un upcasting para poder compilar.

D. Ninguna de las anteriores.

7. ¿Puede invocarse desde un método de una clase derivada un método con idéntica signatura de una de sus superclases?

A. Nunca; de hecho, así es como funciona la sobrecarga.

B. En general, siempre podrá invocarse dicho método mediante una sintaxis que depende del lenguaje de programación.

C. Solo si el método de la superclase es virtual.

D. Solo si el método de la clase derivada es virtual.

8. Dada una implementación correcta de la práctica 2, ¿qué cambios adicionales habría que realizar en los métodos de las clases `MenuSinSal` y `MenuCompleto` para que sigan comportándose como se pide en el enunciado, si declaráramos como protegidos los atributos de la clase `Menu`?

A. Ningún cambio: el código seguiría funcionando sin problemas.

B. Sería necesario cambiar a protegida la herencia de las clases `MenuSinSal` y `MenuCompleto`.

C. Sería necesario reescribir algunos métodos.

D. Ninguna de las anteriores.

9. Basándonos en el diagrama de clases adjunto y el siguiente código:

```
Titular p; p.toString();
```

A. Se invoca a `Titular::toString()` mediante enlace dinámico.

B. Se invoca únicamente a `Persona::toString()`.

C. Se invoca a `Titular::toString()` mediante enlace estático.

D. Ninguna de las anteriores.

10. ¿Qué tipo de herencia en C++ permite a una clase derivada acceder a los métodos privados de una de sus clases base?

A. Protegida o privada, pero no pública.

B. Pública.

C. Privada.

D. Ninguna de las anteriores.

11. ¿Qué caracteriza a una clase abstracta en C++?

A. Que no tiene atributos.

B. Que no tiene definido ningún constructor.

C. Que tiene al menos un método virtual.

D. Ninguna de las anteriores.

12. ¿Cuál de las siguientes clases constituye una interfaz en C++?

A. `class S {public: Object *o;};`

B. `class S {public: S(); S(const S &s); virtual ~S();};`

C. `class S {void f()=0;};`

D. `class S {public: virtual void f()=0;};`

13. La instrucción `throw`

A. sólo permite lanzar objetos de clase `exception` o de clases derivadas de ella.

B. permite lanzar como excepción cualquier tipo de dato.

C. no se utiliza nunca sin especificar qué excepción debe lanzar.

D. no se puede usar dentro de un bloque `catch`.

14. En C++,

A. es obligatorio especificar qué excepciones lanza una función mediante una cláusula `throw` tras la declaración de la función.

B. la cláusula `throw()` tras la declaración de una función indica que ésta no lanza ninguna excepción.

C. una excepción de usuario es una clase definida por el usuario que deriva de la clase `exception`.

D. Ninguna de las anteriores.

15. Las funciones genéricas

A. No se pueden sobrecargar ni sobreescribir.

B. No se pueden sobreescribir pero sí sobrecargar.

C. No se pueden sobrecargar pero sí sobreescribir.

D. Ninguna de las anteriores.

16. En cuanto a los constructores:

A. En C++, si no se define un constructor sin parámetros explícitamente, el compilador proporciona uno por defecto.

B. Son funciones miembro constantes.

C. Siempre tienen visibilidad pública.

D. Ninguna de las anteriores.

17. Los atributos de instancia, si son constantes:

A. Se les asigna su valor inicial directamente cuando se declaran.

B. Se les asigna su valor inicial en el cuerpo del constructor.

C. Se les asigna su valor inicial mediante inicializadores en el constructor.

D. Se les asigna su valor inicial fuera de la clase.

18. Las excepciones

A. Si se ignoran, se produce un error de compilación.

B. Si se ignoran, el compilador genera mensajes de advertencia (*warnings*).

C. Si se ignoran, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente.

D. Ninguna de las anteriores.

19. Dada una clase genérica

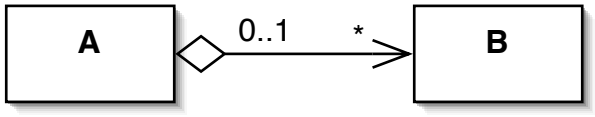
A. No se pueden derivar de ella clases genéricas.

B. No se pueden derivar de ella clases no genéricas.

C. Puede ser utilizada como clase base en herencia múltiple.

D. Ninguna de las anteriores.

20. Dado el diagrama UML:



A. Refleja la existencia de una relación todo-parte en la que el objeto compuesto maneja la creación/destrucción de los objetos componente.

B. Refleja la existencia de una asociación binaria unidireccional donde los objetos de tipo A no están necesariamente asociados a algún B.

C. Refleja la existencia de una relación todo-parte en la que la existencia del objeto parte no depende de la existencia del objeto todo que la contiene.

D. Al implementarlo en C++, la clase A tiene que destruir la memoria de los objetos B que dependan de ella, para que no quede memoria sin liberar.

21. Hablamos de encapsulación cuando:

A. Omitimos ciertos detalles de implementación.

B. Diferenciamos entre interfaz e implementación.

C. Agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos.

D. B y C son correctas.

22. Las operaciones de clase:

A. Sólo pueden manejar atributos de clase.

B. No son funciones miembro de la clase.

C. Sólo pueden ser invocadas por objetos constantes.

D. Ninguna de las anteriores.

23. Las funciones sobrecargadas son aquéllas que:

A. Se les puede pasar diferente cantidad de argumentos, dependiendo lo que nos interese sin que para ello debamos implementar cada caso.

B. Tienen más de una implementación diferenciando cada una por el número, orden y tipo de argumentos.

C. Tienen más de una implementación que, en general, pueden distinguirse únicamente por el tipo devuelto.

D. Reciben como argumento al menos una variable polimórfica.

24. Cuando sobrecargamos un operador:

A. Podemos hacerlo incluso para los tipos predefinidos.

B. Sólo podemos sobrecargar operadores para tipos definidos por el usuario. Debemos respetar las reglas de asociatividad y precedencia y podemos crear nuevos operadores.

C. No podemos hacer sobrecarga de tipos predefinidos. No podemos crear nuevos operadores. Debemos respetar las reglas de asociatividad y precedencia.

D. Ninguna de las anteriores.

25. ¿Qué implicaciones tiene declarar un dato miembro de una clase como private?

A. Indica que ese atributo sólo es accesible por las funciones de ese módulo de programa.

B. Indica que sólo lo puede usar la función donde se declara.

C. Indica que sólo se puede acceder a ese atributo o método desde las funciones miembro de la clase donde se declara.

D. Ninguna de las antetiores.

BÁSICOS.

1. La forma canónica en C++ incluye el constructor copia, el destructor, el operador asignación, el constructor por defecto y cualquier otro constructor parametrizado. F

2. En JAVA la forma canónica incluye constructor, equals, hashCode, toString y clone. F

3. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base. F

4. Independientemente del tipo de herencia la clase base siempre podrá acceder a lo público, protegido y default heredado pero no a lo privado. F

5. Una clase abstracta se caracteriza por declarar todos sus métodos de instancia como abstractos. F

6. En JAVA, la clase deberá contener al menos un método abstracto para ser declarada como tal, en C++ no. F

7. La siguiente sentencia class S{ public final void f() {}} constituye una interfaz en JAVA. F

8. La siguiente sentencia interface S{ void f(); } constituye una interfaz en C++. F

9. Desde un método de una clase derivada nunca puede invocarse un método implementando con idéntica signatura de su clase. F

10. En Java los métodos de instancia con polimorfismo puro pero no abstractos tienen enlace dinámico. V

11. Una operación de clase solo puede acceder directamente a atributos de clase. V

12. Una operación de instancia puede acceder directamente a atributos de clase y de instancia. V

13. Una operación de clase no es una función miembro de la clase. F

14. En la misma clase podemos definir constructores con distinta visibilidad. V

15. El modificador static es correcto para JAVA pero no para C++ cuando lo usamos con constructores. F

16. Un buen diseño orientado a objetos se caracteriza por un alto acoplamiento y una baja cohesión. F

17. Mediante la herencia de implementación heredamos la implementación de los métodos de la clase base pero no la interfaz de esta. F

18. La genericidad es un tipo de polimorfismo. V

19. El polimorfismo es un tipo de genericidad. F

20. En JAVA y en C++ todas las clases son polimorficas. F

21. El downcasting en JAVA y en C++ es siempre dinámico. F

22. Si la conversión, downcasting, es fuera de la jerarquía de herencia JAVA dará error de ejecución. F

23. El downcasting implica deshacer el principio de sustitución. V

GESTIÓN DE ERRORES.

24. En Java, si no se captura una excepción lanzada por un método da error de compilación. F

25. La instrucción throw en JAVA solo permite lanzar objetos que son de la clase throwable o clases derivadas de esta. V

26. Uno de los objetivos del tratamiento de errores mediante excepciones es el manejo de errores del resto del código. V

27. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido error y continúa su ejecución normalmente. F

28. En JAVA, siempre es obligatorio especificar que excepciones verificadas (checked exceptions) lanza un método mediante una cláusula throws tras la lista de argumentos. F

29. Si se produce una excepción el método que la provoca se cancela y se continúa la ejecución en el método que llamo a este. V

30. Si se produce una excepción en un constructor el objeto se construirá con los valores por defecto. F

31. Todas las excepciones son checked exception salvo las runtime que son uncheked exception. V

32. La cláusula throws de un método incluirá todas las excepciones uncheked exception que puedan producirse en este y no estén dentro del bloque try catch que las capture. F

33. El orden de las excepciones en los bloques catch no es relevante.

34. Podemos poner un bloque finally sin poner bloques catch. F

35. El bloque finally solo se ejecutará si se produce alguna excepción en el bloque try al que esté asociado. F

36. IllegalArgumentException, ArrayIndexOutOfBoundsException, ClassCastException y IOException son excepciones del tipo runtimeException y por tanto no es necesario capturarlas ni indicar throws en el método en el que se provoquen. F

GENERICIDAD.

37. La genericidad se considera una característica opcional de los lenguajes. V

38. La genericidad se considera una característica opcional de los lenguajes orientados a objetos. V

39. No se puede derivar una clase genérica de una clase no genérica. F

40. En JAVA no podemos crear interfaces genéricas. F

41. En los métodos genéricos solo podremos usar los métodos definidos en Object. V

42. En la genericidad restringida solo podremos usar los métodos de Object al implementar los métodos. F

43. Los métodos genéricos no se pueden sobrecargar ni sobrescribir. F.

REFLEXIÓN.

44. La API de reflexión de JAVA incluye métodos para obtener la signatura de todos los métodos. V

45. La reflexión permite que un programa obtenga información sobre si mismo en tiempo de ejecución. V

46. Para usar reflexión en Java hemos de conocer el nombre de las clases en tiempo de compilación. F

47. En Java el concepto de meta-clase se representa con la clase Class. V

48. Con el uso de la reflexión solo podemos invocar métodos de instancia. F

49. La reflexión sólo es útil para trabajar con componentes JavaBeans. F

50. La reflexión es demasiado compleja para usarla en aplicaciones de propósito general. F

51. La reflexión reduce el rendimiento de las aplicaciones. F

52. La reflexión no puede usarse en aplicaciones certificadas con el estándar 100% Pure Java. F

53. Mediante reflexión podemos saber cuales son las clases derivadas de una clase dada. V

54. Mediante reflexión no podemos saber cual es el método que se está ejecutando en un determinado momento. V

55. Podemos usar reflexión para encontrar un método heredado (solo hacia arriba) y reducir código condicional. V

REFACTORIZACIÓN.

56. La refactorización debe hacerse siempre apoyandonos en un conjunto de tests completo y robusto. V

57. Una clase con un gran número de métodos y atributos es candidata a ser refactorizada. V

58. Los métodos grandes (con muchas instrucciones) son estructuras que sugieren la posibilidad de una refactorización. V

59. En la refactorización se permite que cambie la estructura interna de un sistema software aunque varíe su comportamiento externo. F

60. Un ejemplo de refactorización sería mover un método arriba o abajo en la jerarquía de herencia. V

61. El cambio de una sentencia condicional por el uso de polimorfismo es un ejemplo de refactorización. V.

62. Hacer el código más fácil de entender no es un motivo suficiente para refactorizarlo. **F**.

63. Existe un catálogo de refactorizaciones comunes, de forma que el programador no se ve obligado a usar su propio criterio y metodología para refactorizar el código. V

64. El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utiliza. V

65. Con el uso de reflexión solo podemos acceder a métodos de instancia. F

FRAMEWORKS.

66. Los frameworks no contienen implementación alguna, únicamente un conjunto de interfaces que deben ser implementados por el usuario del framework. F

67. Un framework invoca mediante enlace dinámico a nuestra implementación de interfaces propios de framework. V

68. Hibernate es un framework para congelar en memoria el estado de nuestra aplicación. F

69. JDBC es un framework de Java que usan los fabricantes de sistemas de gestión de bases de datos para ofrecer un acceso estandarizado a las bases de datos. V

70. El usuario de un framework implementa al componente declarado de los interfaces de framework mediante herencia de implementación. F

71. Un frameworks es un conjunto de clases cuyos métodos invocamos para que realicen unas tareas a modo de caja negra. F

72. En Java el acceso a bases de datos se hace con librerías propietarias de SGBD cuyos interfaces no tienen ningún estándar. F

73. El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de interfaz. V

74. Para poder utilizar un framework, es necesario crear clases que implementen todas las interfaces declaradas en el framework. V

OTROS.

75. Una librería de clases proporciona una funcionalidad completa, es decir, no requiere que el usuario implemente o herede nada. V

76. El polimorfismo es una forma de reflexión. F

77. En el proceso de diseño de un sistema de software se debería intentar aumentar el acoplamiento y la cohesión. F

78. Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación. V

79. Todo espacio de nombre define su propio ámbito, distinto de cualquier otro. V

80. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea. V

81. En el diseño mediante tarjetas CRC utilizamos una tarjeta para cada clase. V

82. Una tarjeta CRC contiene el nombre de una clase, su lista de responsabilidades y su lista de colaboradores. V

83. La robustez de un sistema software es un parámetro de calidad intrínseco. F

84. El principio abierto-cerrado indica que un componente software debe estar abierto a su extensión y cerrado a su modificación. V

85. Con el diseño orientado a objetos es perfectamente posible y deseable hacer uso de variables globales. F

86. En el diseño por contrato son dos componentes fundamentales las pre y pos condiciones. V

1. Un atributo de clase ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de la clase derivada.

2. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro.

3. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación del objeto receptor del mensaje.

4. En el cuerpo de una operación de clase se puede acceder únicamente a atributos/operaciones de clase.

5. En C++ el constructor copia permite argumentos tanto por referencia como por valor.

6. En C++ el constructor copia se invoca de forma implícita cuando devolvemos un valor o pasamos un objeto como argumento por valor.

7. Las declaraciones A a(b); y A a = b; (siendo b un objeto de tipo A) invocan ambas al constructor de copia de la clase A.

8. En C++ el constructor copia solo pertenece a la forma canónica de la clase si esta procesa memoria dinámica.

9. La forma canónica en JAVA incluye el constructor, constructor copia, equals, hashcode, toString y el método clone.

10. En JAVA y en C++ solo podremos especificar como visibilidad para un atributo/operación son: prívate , public y protected.

11. El método clone() que proporciona JAVA por defecto, realiza una 'Deep copy' que copia bit a bit el contenido del objeto pasado como parámetro en el nuevo objeto.

12. Tanto JAVA como C++ proporcionan un constructor copia que realiza una Shallow copy del objeto pasado como parametro.

13. Tanto JAVA como C++ proporcionan un método clone que devuelve una copia Shallow copy del objeto que invoca el método.

14. El método clone que se nos proporciona por defecto invoca al clone de la clase base, así como, el método constructor copia que C++ proporciona por defecto.

15. C++ siempre proporciona un constructor por defecto (sin parámetros) de oficio, JAVA no.

16. C++ siempre proporciona un constructor por defecto (sin parámetros) mientras que no definamos ningún otro constructor.

17. JAVA siempre proporciona un constructor por defecto (sin parámetros) de oficio, mientras que no definamos ningún otro constructor.

18. El orden de ejecución de los constructores en la herencia es primero la derivada y luego la base.

19. Un atributo privado en la clase base no es directamente accesible en las derivadas sea cual sea el tipo de herencia, tanto en JAVA como en C++.

20. Un atributo public en base es accesible en main desde un objeto derivado sea cual sea el tipo de herencia.

21. En JAVA la herencia solo puede ser pública, mientras que en C++ solo puede ser privada o protegida.

22. La herencia pública implica herencia de implementación y de interfaz mientras que la herencia privada o protegida implica herencia de implementación pero no de interfaz.

23. La herencia de implementación siempre implica herencia de interfaz.

24. La existencia de una relación todo-parte entre dos clases implica necesariamente que el objeto todo maneja la creación/destrucción de los objetos de tipo parte.

25. Las relaciones todo-parte (tiene-un) son la agregación y la composición.

26. Las relaciones todo-parte son relaciones persistentes como la herencia.

27. En la composición el objeto todo maneja la creación/destrucción de los objetos de tipo parte al contrario que la agregación que no la maneja.

28. La única relación no persistente es la relación de uso, ya que no se materializa mediante referencias.

29. Todo son relaciones de clase excepto la relación de uso que es relación de objeto.

30. La agregación es una relación mucho más restrictiva que la herencia.

31. La agregación y la asociación pueden ser bidireccionales.

32. En JAVA un atributo de clase público puede ser accedido desde fuera de la clase a través de una referencia de la clase o mediante el nombre de la clase.

33. En C++ un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito.

34. Implementar la forma canónica de una clase es una condición necesaria (aunque no suficiente) para controlar que el valor de un atributo de clase que cuenta el número de instancias de dicha clase esté siempre en un estado consistente.

35. Un constructor en JAVA, acepta cualquier tipo de modificador (static, final, public, etc).

36. En C++ los métodos virtuales puros, pueden tener implementación pero siempre harán abstracta a la clase que los contenga.

37. En JAVA los métodos abstractos pueden tener implementación pero siempre harán abstracta a la clase que los contenga.

38. La sentencia Derivada d = d2.clone(); es correcta si d y d2 son de tipo Derivada.

39. Una clase abstracta siempre tiene que tener alguna clase que derive de ella.

40. En una composición un objeto componente puede forma parte de más de un objeto compuesto.

41. En una composición un objeto compuesto puede poseer varios componentes del mismo tipo al contrario que en la agregación.

42. Ni la asociación ni la agregación manejan la creación/destrucción de objetos de tipo parte.

43. Los destructores se pueden invocar como si fueran métodos y acepta cualquier número de parámetros.

44. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada pero no modificar los métodos heredados de la clase base.

45. El puntero this es un puntero que no puede cambiar de valor y que contiene la dirección del objeto receptor del mensaje, además existe en cualquier método definido dentro de la clase.

46. La interpretación de un mismo mensaje puede variar en función del receptor del mensaje y de la información adicional que lo acompañe.

47. La relación de uso es una relación no persistente.

48. En C++ es posible definir un constructor de copia invocando únicamente al operador asignación.

49. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO.

50. Para que se pueda realizar una herencia múltiple en C++, es necesario que no coincida ninguno de los nombre de atributo entre las clases base involucradas.

51. La signatura de tipo incluye el tipo devuelto por el método, el tipo el numero y el orden de los parámetros y el nombre del método.

52. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de herencia.

53. El puntero this no es una variable polimórfica porque es constante y no se puede cambiar su valor.

54. No se puede derivar de una clase no genérica a una genérica.

55. No se puede derivar una clase genérica a otra no genérica.

56. Las instrucciones para el manejo de excepciones nos permiten mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores.

57. No pueden haber asociados varios bloques catch a un bloque try.

58. En C++ no se pueden definir sobrecargas de operadores fuera de la clase a no ser que sean funciones amigas.

59. Definir dos métodos que se llamen igual pero que tengan distinto tipo devuelto se llama sobrecarga.

60. Para sobrecargar dos métodos dentro del mismo ámbito es necesario que difieran en el número, tipo **y** orden de sus argumentos.

61. En la sobrecarga basada en ámbito los métodos deben diferir en el tipo, orden **o** numero de argumentos.

62. En la sobrecarga nunca se podrá cambiar el tipo devuelto por un método.

63. Los métodos de sobrecarga son el refinamiento y el reemplazo.

64. En JAVA toda clase es potencialmente polimórfica en C++ solo si posee algún método virtual.

65. Ser abstracto implica enlace dinámico.

66. Un método con enlace dinámico siempre será abstracto.

67. En JAVA se puede elegir entre enlazar un método de forma estática, no poniendo virtual delante del método.

68. Los miembros de instancia final de una clase se pueden inicializar en el cuerpo de los constructores.

69. JAVA inicializa todos los atributos de una clase, incluyendo los atributos constantes.

70. Si A tiene una relación de uso con B, no contendrá datos de tipo B pero algunos de sus métodos recibirán objetos de tipo B como parámetros, accederán a sus variables privadas o usarán algún método de la clase B.

71. En algunos lenguajes una clase es una instancia de otra clase, llamada metaclase.

72. El diagrama de clases define las clases, sus propiedades y como se relacionan unas con otras, proporcionando una vista estática de los elementos que conforman el software.

73. Los componentes de una tarjeta CRC son el nombre de la clase, sus responsabilidades y los colaboradores con los que esta clase interactúa.

74. Los bloques try catch no se pueden anidar.

75. Si en el bloque try de un catch, que captura una checked exception, es imposible que se produzca dicha excepción el código no compilará, indicando que el bloque catch es inalcanzable.

76. Si en la sentencia throws de un método especificamos una excepción que es imposible que se lance en el cuerpo de dicho método el código no compilará, indicando que el método nunca lanzara la excepción.

77. Cuando un método sobrescribe a otro, este podrá especificar lista de excepciones en su declaración mediante la sentencia throws.

78. Cuando un método sobrescribe a otro, si indica una excepción de tipo checked dicha excepción (o una compatible) deberá aparacer en la lista throws del método sobrescrito.

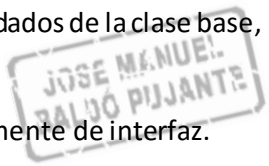
79. Cuando un método sobrescribe a otro, el nuevo método podrá cambiar la visibilidad del método heredado sin restricciones.

80. En JAVA solo existe la sobrescritura, la redefinición y la ocultación son características de C++. 😐

81. Cuando una clase sobrescribe el comportamiento de un método heredado estaremos haciendo herencia de interfaz.

82. Una clase que sobrescriba todos los métodos heredados de la clase base, realiza herencia de interfaz. 😐

83. Cuando una clase sobrescribe algunos de los métodos heredados de la clase base, estaremos únicamente ante una herencia de implementación.

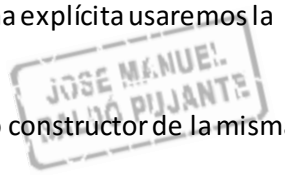


84. La herencia privada y protegida de C++ son herencia únicamente de interfaz.

85. Los constructores no se heredan.

86. Los constructores siempre se refinan ya sea de forma explícita o implícita.

87. Para invocar al método constructor de la clase base de forma explícita usaremos la sentencia super.Base(parámetros).



88. No podemos invocar desde un constructor de la clase a otro constructor de la misma clase.

89. Tanto en JAVA como en C++ los constructores se pueden invocar como si fueran métodos, objeto.constructor(parámetros).

90. C++: En la herencia primero se destruyen los objetos bases y luego las derivadas, es decir, primero se ejecutan los destructores de la base y luego los de las derivadas.

91. Un objeto derivada al que se accede a través de una referencia a clase base sólo se puede manipular usando haciendo uso de la interfaz de la clase base.

92. C++ soporta herencia multiple de implementación y de interfaz, sin embargo, JAVA solo soporta herencia multiple de interfaz.

93. Dos son los problemas que nos podemos encontrar en la herencia múltiple en C++, la colisión de nombres y la duplicación de propiedades.

94. La colisión de nombres se podrá resolver mediante el operador de ámbito.

95. La duplicación de propiedades se resuelve usando el atributo virtual.

96. La herencia de interfaz garantiza siempre el principio de sustitución.

97. Mientras que la especialización y la especificación son un uso seguro de la herencia, la generalización la restricción y la varianza no lo son.

98. La herencia privada en C++ implementa un tipo de herencia de construcción que si preserva el principio de sustitución

99. La herencia de construcción o herencia de implementación pura, no cumple el principio de sustitución.

100. La herencia ralentiza la velocidad de ejecución.

101. La regla del cambio y la regla del polimorfismo nos servirán para diferenciar si estamos ante una relación IS-A o una relación HAS-A.

102. La composición es una técnica generalmente más sencilla que la herencia y más flexible, resistente en cuanto a cambios.

103. En la sobrecarga en signatura de tipos, en los parámetros, no se considerarán tipos distintos, si el parámetro en el nuevo método es derivado del parámetro del sobrescrito.

104. Al sobrescribir un método en clase derivada, podemos cambiar el tipo de retorno del método a un subtipo del especificado en la clase base.

105. Si usamos sobrecarga basada en sigunatura de tipos, los métodos tienen enlace estático (static) y los tipos de los parámetros son diferentes pero relacionados por herencia, si invocamos al método usando una referencia de tipo Base, como parámetro, pero que apunta a un derivado, se invocará al método que recibe una Base en su definición.

106. JAVA permite la sobrecarga de operadores, C++ no.

107. En la sobrecarga de operadores, no se puede cambiar la precedencia, asociatividad o aridad de estos.

108. Para sobrecargar un operador como método de clase (función miembro), el operando de la izquierda deberá ser del tipo de la clase.

109. En JAVA el enlazado de métodos es siempre dinámico, aunque un método definido com o final, se comportaría como estático.

110. Tanto la redifinición como el shadowing tienen enlazado estático.

111. Podemos utilizar el atributo final para indicar que no podemos heredar de dicha clase.

112. En JAVA toda variable es potencialmente polimórfica, en C++ solo aquellas que poseen métodos virtuales.

113. Las variables this y super son variables polimórficas.

114. El Downcasting es el polimorfismo inverso.

115. C++ y JAVA soportan downcasting estatico y dinámico.

116. En el downcasting dinámico la comprobación de la conversión se hará en tiempo de compilación.

117. El downcasting en JAVA es siempre seguro.

118. Un método polimórfico o con polimorfismo puro es aquel que no recibe variables potencialmente polimórficas.

119. ArrayList<Integer> e; y ArrayList<String> s; delvolverán el mismo tipo, si reciben el mensaje getClass().

120. ArrayList<Object> a = new ArrayList<String> (); y Object [] a = new String [3]; son declaraciones correctas.

121. La implementación de un framework sigue el principio de Hollywood: “no nos llames, nosotros te llamaremos”.

122. Cualquier operador puede ser sobrecargado como función miembro.

123. Los miembros de instancia final (constantes) de una clase, se pueden inicializar en el constructor de dicha clase.

124. Los miembros de clase final (constantes) de una clase, se pueden inicializar en el constructor de dicha clase.

125. Los miembros públicos de instancia de una clase son accesibles en cualquier función o método usando el nombre de la clase y el operador de ámbito en C++ y el nombre de la clase y el operador punto en JAVA.

126. Los miembros protected solo son accesibles por los métodos de esa clase.

127.En JAVA y C++ los miembros prívate de una clase son accesibles únicamente por los métodos de esa clase.

128. Si en una relación todo-parte las partes pueden cambiar de objeto todo se aproximará más a ser una agregación que una composición.

129. De una clase abstracta no se podrán crear objetos pero si punteros o referencias.

130. El modelo de herencia en C++ es merge.

131. El atributo virtual se hereda.

132. En C++ el atributo virtual sobre un método cambia su enlazado a dinámico y además se hereda.

133. Tres son los tipos de polimorfismo en jerarquías de herencia ocultación, redefinición y sobrecarga.

134. En la sobrescritura en el método definido en la derivada debe tener la misma signatura que el de la base.

135. En la redifinición (sin virtual en base) puede cambiar únicamente la parte derecha de la signatura.

136. En C++ la sobrecarga de operadores es conmutativa.

137. Las operaciones consultoras obtienen información sobre el estado del objeto (en C++ deben ser declaradas const), mientras que las ordenes modifican el estado del objeto (en C++ no podrán ser declaradas const).

138. C++ y JAVA son lenguajes débilmente tipados.

139. List<Fruit> flist = new ArrayList<Apple>(); Es una asignación correcta siempre que Apple sea derivado de Fruit.

140. List<?> flist = new ArrayList<Apple>(); Es una asignación correcta.

141. Si accedemos a una colección mediante una referencia comodín podremos consultar y añadir elementos a dicha colección.

142. JUnit es una librería.

143. En JAVA podremos crear objetos de tipo Collection, List, Set, SortedSet, Map y SortedMap.

144. Si usamos el ArrayList y el TreeSet estaremos usando el JCF, como un framework.

145. Implementar la interfaz comparator y llamar al método sort implica usar el JCF como un framework.

146. Para poder usar las librerías de del JDK, en algún caso, nos veremos obligados a sobrescribir algún método abstracto.

147. Los toolkits son frameworks.

148. JDBC es una librería.

149. Los desarrolladores usaremos las librerías de los fabricantes que implementan en framework JDBC.

150.

1. La herencia de interfaz se implementa mediante herencia pública.	V
2. Una interfaz no puede tener atributos de instancia. Una clase abstracta si puede tenerlos.	V
3. No se puede definir un bloque catch sin su correspondiente bloque try.	V
4. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo.	V
5. El downcasting siempre es seguro.	F
6. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes.	V
7. En el diseño mediante tarjetas CRC, utilizamos una tarjeta por cada jerarquía de herencia.	F
8. Un espacio de nombres es un ámbito con nombre.	V
9. Un sistema de tipos de un lenguaje asocia a cada tipo una expresión.	V
10. Hacer que el código sea más fácil de entender no es un motivo suficiente para refactorizarlo.	F
11. En Java, los tipos genéricos sólo se pueden aplicar a clases e interfaces.	F
12. En Java, una clase genérica puede ser parametrizada empleado más de un tipo.	V
13. Sean dos clases Base e Hija. La clase Hija hereda de Base. En Java, cuando asignamos un objeto de la clase Hija a una referencia a Base haciendo conversión de tipo explícita estamos haciendo object slicing.	F
14. Una de las principales fuentes de problemas cuando utilizamos herencia múltiple es que las clases bases hereden de un ancestro común.	V
15. Los lenguajes de programación soportan el reemplazo y el refinamiento como métodos de sobrescritura, pero no hay ningún lenguaje que proporcione ambas técnicas (JAVA solo soporta reemplazo y C++ sólo soporta refinamiento).	F
16. Una interfaz puede implementar otra interfaz.	F
17. En java es obligatorio indicar que un método de una clase derivada sobrescribe un método de la clase base con la misma signatura.	F
18. En la sobrecarga basada en ámbito los métodos pueden diferir únicamente en el tipo devuelto.	V
19. En la herencia pública la clase derivada podrá acceder a los atributos privados de la clase base de la que hereda.	F
20. Una clase abstracta se caracteriza por no tener ningún constructor.	F
21. El cambio de una condicional por el uso de polimorfismo es un ejemplo de refactorización.	V
22. Un atributo siempre tiene visibilidad pública.	F
23. El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utilice.	V
24. El usuario de un framework implementa el comportamiento	F

declarado en los interfaces del framework mediante herencia de implementación	
25. El proceso de diseño de un sistema software se debería intentar aumentar la cohesión y reducir el acoplamiento.	V
26. Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar cerradas a la extensión y abiertas a la modificación.	F
27. La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización.	V
28. Existe un catálogo de refactorizaciones comunes de forma que el programador no se ve obligado a usar su propio criterio y metodología para refactorizar código.	V
29. Arraylist es una implementación en el Java Collection Framework de la interfaz List.	V
30. Con el uso de reflexión solo podemos invocar métodos de instancia.	F
31. La siguiente clase class S{ public Object obj; } constituye una interfaz en Java.	F
32. Desde un método de una clase derivada solamente puede invocarse un método implementado con idéntica signatura de una de sus superclases si el método en la clase base tiene enlace dinámico.	F
33. Los métodos con enlace dinámico son métodos abstractos.	F
34. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica.	F
35. La genericidad se considera una característica opcional de los lenguajes orientados a objetos.	V
36. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.	V
37. En java, podemos definir constructores con distinta visibilidad en la misma clase.	V
38. Los métodos genéricos no se puede sobre sobrecargar ni sobrescribir.	F
39. Una interfaz no tiene instancia. Por ejemplo, dada la interfaz Comprable en Java, no podemos hacer new Comparable().	V
40. Una interfaz en Java obliga a que las **clases no abstractas** que la implementan definan todos los métodos que la interfaz declara.	V

1. Los métodos definidos en una clase derivada nunca pueden acceder a las propiedades privadas de una clase base.
2. Una clase abstracta se caracteriza por no tener definido ningún constructor.
3. La siguiente clase: `class S {public: S(); S(const S &s); virtual ~S();};` constituye una interfaz en C++.
4. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base.
5. Los métodos virtuales son métodos abstractos.
6. Los métodos abstractos son métodos con enlace dinámico.
7. Los constructores de las clases abstractas son métodos con enlace dinámico.
8. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
9. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica.
10. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica.
11. En C++ no podemos hacer sobrecarga de operadores para tipos predefinidos.
12. En C++, si no se define un constructor sin argumentos explícitamente, el compilador proporciona uno por defecto.
13. En la misma clase, podemos definir constructores con distinta visibilidad.
14. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente.
15. La instrucción `throw` permite lanzar como excepción cualquier tipo de dato.
16. Las funciones genéricas no se pueden sobrecargar.
17. Dada una clase genérica, se pueden derivar de ella clases no genéricas.
18. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor.
19. Una clase interfaz no puede tener atributos de instancia. Una clase abstracta sí puede tenerlos.
20. La herencia de interfaz se implementa mediante herencia pública.

VERDADERO
FALSO
FALSO
FALSO
FALSO
VERDADERO

FALSO
FALSO
FALSO
FALSO
VERDADERO
FALSO

VERDADERO
FALSO
FALSO
VERDADERO
FALSO
VERDADERO

FALSO
VERDADERO
VERDADERO

1. La herencia pública permite a los métodos definidos en una clase derivada acceder a las propiedades privadas de la clase base.
2. Una clase abstracta se caracteriza por declarar al menos un método abstracto.
3. La siguiente clase: `class S {public: Object *o;};` constituye una clase interfaz en C++.
4. Los métodos abstractos siempre tienen enlace dinámico.
5. Los constructores siempre son métodos virtuales.
6. Un método tiene polimorfismo puro cuando tiene como argumentos al menos una variable polimórfica.
7. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A.
8. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
9. Una operación de clase sólo puede ser invocada mediante objetos constantes.
10. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos.
11. Dada una clase genérica, no se pueden derivar de ella clases genéricas.
12. Una clase interfaz no puede tener instancias.
13. Una clase abstracta siempre tiene como clase base una clase interfaz.
14. No se puede definir un bloque *catch* sin su correspondiente bloque *try*.
15. Tras la ejecución de un bloque *catch*, termina la ejecución del programa.
16. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo.
17. El *downcasting* estático siempre es seguro.
18. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases.
19. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes.
20. Una operación de clase no puede tener enlace dinámico.

FALSO
VERDADERO
FALSO
VERDADERO
FALSO

VERDADERO
FALSO
VERDADERO
FALSO

VERDADERO
FALSO
VERDADERO
FALSO
VERDADERO
FALSO

VERDADERO
FALSO
VERDADERO
VERDADERO
VERDADERO

1. La herencia protegida permite a los métodos de la clase derivada acceder a las propiedades privadas de la clase base.
2. Una clase abstracta se caracteriza por no tener definido ningún constructor.
3. La siguiente clase en C++: `class S {public: virtual ~S()=0;};` define una interfaz.
4. Los métodos virtuales son métodos abstractos.
5. Los métodos abstractos siempre tienen enlace dinámico.
6. Los constructores siempre tienen enlace dinámico.
7. En C++, un atributo de clase debe declararse dentro de la clase con el modificador `static`.
8. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica.
9. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica.
10. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A.
11. Dada la siguiente definición de clase en C++:

```
class TClase {
  public:
    TClase(int dim);
  private:    int var1;
};
```

    La instrucción `TClase c1;` no da error de compilación e invoca al constructor por defecto.
12. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
13. Hablamos de encapsulación cuando agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos.
14. Una operación de clase sólo puede ser invocada mediante objetos constantes.
15. En C++ los constructores se pueden declarar como métodos virtuales.
16. En la misma clase, podemos definir constructores con distinta visibilidad.
17. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continua su ejecución normalmente.
18. En C++, es obligatorio especificar qué excepciones lanza una función mediante una cláusula `throw` tras la declaración de la función.
19. Dada una clase genérica, se pueden derivar de ella clases no genéricas.
20. Una clase interfaz no debe tener atributos de instancia. Una clase abstracta sí puede tenerlos.

1 F	6F	11F	16V
2F	7V	12V	17F
3 V	8F	13V	18F
4F	9F	14F	19V
5V	10F	15F	20V

1. Un mensaje tiene cero o más argumentos. Por el contrario, una llamada a procedimiento/función tiene uno o más argumentos.
2. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña.
3. En una agregación, la existencia de un objeto *parte* depende de la existencia del objeto *todo* que lo contiene.
4. Una forma de mejorar el diseño de un sistema es reducir su acoplamiento y aumentar su cohesión.
5. Los inicializadores en C++ tienen el formato *nombreAtributo(valor)* y se colocan entre la lista de argumentos y el cuerpo de cualquier método, permitiendo asignar valores a los atributos de instancia de la clase en la que se define dicho método.
6. Un atributo estático ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.
7. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
8. La herencia múltiple se produce cuando de una misma clase base heredan varias clases derivadas.
9. Un atributo protegido en la clase base es también protegido en cualquier clase que derive de dicha clase base, independientemente del tipo de herencia utilizado.
10. Cuando se crea un objeto de una clase D que deriva de una clase B, el orden de ejecución de los constructores es siempre *B() D()*.
11. Una clase abstracta es una clase que no permite definir instancias de ella.
12. Un interfaz es una clase abstracta con al menos un método de instancia abstracto.
13. Un método de clase (estático) no puede tener enlace dinámico.
14. Un método virtual en C++ siempre tiene enlace estático.
15. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases.
16. La llamada a un método sobrescrito se resuelve en tiempo de compilación.
17. El *downcasting* estático siempre es seguro.
18. Los métodos definidos en una clase genérica son a su vez genéricos.
19. No se puede definir un bloque *catch* sin su correspondiente bloque *try*.
20. La instrucción *throw* (en C++) sólo permite lanzar objetos de clase *exception* o de clases derivadas de ella.

1F	6F	11V	16F
2V	7F	12F	17F
3F	8F	13V	18V
4V	9F	14F	19V
5F	10V	15V	20F

1. **En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro**

2. **En una composición un objeto componente puede formar parte de más de un compuesto.**

3. **La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o de la información adicional que le acompaña.**

4. **Los destructores en C++ pueden aceptar cualquier número de parámetros.**

5. **Los TAD´s representan una perspectiva orientada a datos, mientras que los objetos reflejan una perspectiva orientada a servicios.**

6. **Un atributo estático ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.**

7. **Un atributo privado en la clase base no es directamente accesible en la clase derivada, independientemente del tipo de herencia utilizado.**

8. **Un constructor acepta cualquier tipo de modificador (static, const, public, etc)**

9. **Una clase es una especificación abstracta de una estructura de datos y de las operaciones que se pueden realizar con ella.**

10. **Una operación constante sólo puede ser invocada por un objeto constante.**

1F
2F
3V
4F
5V
6F
7V
8F
9F
10F

1. **El puntero this es un puntero constante al objeto que recibe el mensaje**

2. **Implementar la forma canónica ortodoxa de una clase es una condición necesaria (aunque no suficiente) para controlar que el valor de un atributo de clase que cuenta el número de instancias de dicha clase esté siempre en un estado consistente.**

3. **La existencia de una relación todo-parte entre dos clases implica necesariamente que el objeto ´todo´ maneja la creación/destrucción de objetos de tipo ´parte´**

4. **La forma canónica de la clase está formada por el constructor, el constructor de copia, el destructor y el operador de asignación**

5. **Si la clase no proporciona un constructor sin parámetros, el compilador en C++ genera uno de oficio**

6. **Tanto composición como herencia son mecanismos de reutilización del software**

7. **Un atributo de clase debe declararse dentro de la clase con el modificador const**

8. **Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito**

9. **Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada, pero no modificar los métodos heredados de la clase base**

10. **Una interfaz es la definición de un protocolo para cierto comportamiento, sin especificar la implementación de dicho comportamiento**

1V
2F
3F
4V
5F
6V
7F
8V
9F
10V

1. **C++ sólo permite heredar cuando la clase hija es un subtipo de la clase padre (herencia como implementación de la generalización)**

2. **El constructor de copia permite argumentos tanto por referencia como por valor.**

3. **El estado de un objeto es el conjunto de valores de atributos y métodos que han sido invocados sobre él.**

4. **En las jerarquías de herencia en C++, si la clase base define un operador de asignación y la clase derivada no lo redefine, al invocar a dicho operador con objetos de la clase derivada se invocará al código de la clase base.**

5. **La herencia es más flexible en cuanto a posibles cambios en la naturaleza de los objetos que la composición**

6. **La herencia privada en C++ es un tipo de herencia insegura porque no preserva el principio de encapsulación.**

7. **La relación de herencia es una relación de clases no persistente.**

8. **Los inicializadores en C++ tienen el formato nombre_atributo(valor) y se colocan entre la lista de argumentos y el cuerpo de cualquier método. Estos inicializadores permiten asignar valores a los atributos de la clase en la que se define dicho método.**

9. **Un objeto se caracteriza por poseer un estado, un comportamiento y una identidad.**

10. **Una clase abstracta siempre tiene que tener alguna clase que derive de ella.**

1F
2F
3F
4F
5F
6F
7F
8F
9V
10F

1. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base.
2. Una clase abstracta se caracteriza por no tener atributos.
3. La siguiente clase: `class S {public: virtual ~S()=0; virtual void f()=0;};` constituye una interfaz en C++.
4. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base.
5. Los métodos con enlace dinámico son abstractos.
6. Los constructores de las clases abstractas siempre son métodos abstractos.
7. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
8. Un metodo sobrecargado es aquel que tiene más de una implementación, diferenciando cada una por el ámbito en el que se declara, o por el número, orden y tipo de argumentos que admite.
9. Un método abstracto es un método con polimorfismo puro.
10. Todo espacio de nombres define su propio ámbito, distinto de cualquier otro ámbito.
11. En la sobrecarga de operadores binarios para objetos de una determinada clase, si se sobrecarga como función miembro, el operando de la izquierda siempre es un objeto de la clase.
12. La genericidad se considera una característica opcional de los lenguajes orientados a objetos
13. Hablamos de encapsulación cuando diferenciamos entre interfaz e implementación.
14. Una operación de clase no es una función miembro de la clase.
15. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos.
16. Los constructores siempre deben tener visibilidad pública.
17. En C++, si no se captura una excepción lanzada por un método, se produce un error de compilación.
18. En C++, la cláusula `throw()` tras la declaración de una función indica que ésta no lanza ninguna excepción.
19. Dada una clase genérica, no puede ser utilizada como clase base en herencia múltiple.
20. De una clase interfaz no se pueden crear instancias. De una clase abstracta sí.

FALSO	FALSO	VERDADERO	VERDADERO	VERDADERO	VERDADERO
FALSO	FALSO	FALSO	VERDADERO	FALSO	FALSO
VERDADERO	FALSO	VERDADERO	VERDADERO	FALSO	FALSO
	FALSO		FALSO		

1. Cuando usamos la *varianza* estamos haciendo un uso inseguro de la herencia de implementación.

2. En la sobrecarga de operadores como función miembro, el operando de la izquierda puede ser un objeto de la clase o cualquier otro objeto, mientras que en las funciones amigas siempre es un objeto de la clase.

3. Cuando creamos un objeto en C++ mediante una variable automática el constructor se autoinvoca. También se autoinvoca el destructor del mismo al salir del ámbito de la función donde se creó.

4. Si para una clase genérica llamada Pila<T> declaramos las siguientes funciones:

```
virtual void apilar(T* pt);
virtual void apilar(T t);
```

   Es un caso de sobrescritura ya que el tipo devuelto es el mismo en ambos métodos.

5. De una clase abstracta se pueden crear referencias a objetos de la clase.

6. En un atributo de clase se reserva espacio en memoria para una copia de él por cada objeto de su clase creado.

7. Declarar un dato miembro de una clase como private indica que sólo se puede acceder a ese atributo desde las funciones miembro de la clase.

8. Una clase abstracta siempre tiene que tener alguna clase que derive de ella.

9. El polimorfismo debido a la sobrecarga de funciones siempre se da en relaciones de herencia.

10. A los atributos de instancia si son constantes se les asigna su valor inicial fuera de la clase.

11. Toda sentencia que aparece después de un punto del programa en el que ocurre una excepción, en ningún caso se ejecuta.

12. Si utilizamos los mecanismos de manejo de excepciones disminuye la eficiencia del programa incluso si no se llega a lanzar nunca una excepción.

13. Cuando se captura una excepción y ésta pertenece a una jerarquía de clases, el primer bloque catch debe comenzar con la clase del nivel más alto de la jerarquía.

14. Hablamos de shadowing cuando el método a invocar se decide en tiempo de compilación.

15. A diferencia de otros lenguajes de programación en C++ la sobreescritura en relaciones de herencia se debe indicar de forma explícita en la clase padre .

VERDADERO FALSO FALSO
FALSO FALSO VERDADERO
VERDADERO FALSO FALSO
FALSO FALSO VERDADERO
VERDADERO FALSO VERDADERO

1. La encapsulación es un mecanismo que permite separar de forma estricta interfaz e implementación.
2. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña.
3. Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito.
4. Es posible definir un constructor de copia invocando en su cuerpo al operador de asignación.
5. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO.
6. Para que se pueda realizar una herencia múltiple en C++, es necesario que no coincida ninguno de los nombres de atributo entre las clases base involucradas.
7. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada, pero no modificar los métodos heredados de la clase base.
8. En las jerarquías de herencia en C++, si la clase base define un operador de asignación y la clase derivada no lo redefine, al invocar a dicho operador con objetos de la clase derivada se invocará al método de la clase base.
9. Dada una clase abstracta siempre debe existir alguna clase que derive de ella.
10. La signatura de tipo de un método incluye el tipo devuelto por el método.
11. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases.
12. En C++, un destructor no puede ser virtual.
13. El puntero *this* no es una variable polimórfica porque es constante y no se puede cambiar su valor.
14. No se puede derivar una clase no genérica de una genérica.
15. Las instrucciones para el manejo de excepciones nos permiten mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores.

VERDADERO FALSO FALSO FALSO
VERDADERO FALSO VERDADERO FALSO
VERDADERO FALSO FALSO VERDADERO
VERDADERO FALSO

PREGUNTAS:

1. La forma canoníca en C++ incluye el constructor copia, el destructor, el operador asignación, el constructor por defecto y cualquier otro constructor parametrizado.
2. En JAVA la forma canoníca incluye constructor, equals, hashCode, toString y clone.
3. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base.
4. Independientemente del tipo de herencia la clase base siempre podrá acceder a lo público, protegido y default heredado pero no a lo privado.
5. Una clase abstracta se caracteriza por declarar todos sus métodos de instancia como abstractos.
6. En JAVA la clase deberá contener al menos un método abstracto para ser declarada como tal en C++ no.
7. La siguiente sentencia class S {public final void f(){}} constituye una interfaz en JAVA.
8. La siguiente sentencia interfaz S {void f();} constituye una interfaz en C++.
9. Desde un método de una clase derivada nunca puede invocar un método implementado con idéntica signatura de su clase.
10. Desde un método de una clase derivada nunca puede invocarse un método implementado en esta con el mismo nombre que en la clase base.
11. En Java los métodos de instancia con polimorfismo puro pero no abstracto tienen clase dinámico.
12. Una operación de clase solo puede acceder directamente a atributos de clase.
13. Una operación de instancia puede acceder directamente a atributos de clase y de instancia.
14. Una operación de clase no es una función miembro de la clase.
15. En la misma clase podemos definir constructores con distinta visibilidad.
16. El modificador static es correcto para JAVA pero no para C++ cuando lo usamos con constructores.
17. Un buen diseño OO se caracteriza por un alto acoplamiento y una baja cohesión.
18. Mediante la herencia de implementación heredamos la implementación de los métodos de la clase base pero no la interfaz de esta.
19. La genericidad es un tipo de polimorfismo.
20. El polimorfismo es un tipo de genericidad.
21. En JAVA y en C++ todas las clases son polimórficas.
22. El downcasting en JAVA y en C++ es siempre dinámico.
23. Si la conversión, downcasting, es fuera de la jerarquía de herencia JAVA dará error de ejecución.
24. El downcasting implica deshacer el principio de sustitución.
25. En JAVA si no se captura una excepción lanzada por un método da error la compilación.
26. La instrucción throw en JAVA solo permite lanzar objetos que son de la clase throwable o clases derivada de esta.
27. Uno de los objetivos del tratamiento de errores mediante excepciones es el manejo de errores del resto del código.
28. Si no se captura una excepción por un método, el programa no advierte que ha ocurrido error y continúa su ejecución que llamo a este.
29. En JAVA. Siempre es obligatorio especificar que excepciones verificadas (checked exceptions) lanza un método mediante una clausula throws tras la lista de argumentos.
30. Si se produce una excepción el método que la provoca se cancela y se continúa la ejecución en el método que llamo a este.
31. Si se produce una excepción en un constructor el objeto se construirá con los valores por defecto.
32. Todas las excepciones son checked exception salvo las runtime que son uncheked exception.
33. La cláusula throws de un método incluirá todas las excepciones uncheked exception que puedan producirse en este y no estén dentro del bloque try catch que les capture.
34. El orden de las excepciones en los bloques catch no es relevante.
35. Podemos poner un bloque finally sin poner bloques catch.
36. El bloque finally solo se ejecutará si se produce alguna excepción en el bloque try al que este asociado.
37. IllegalArgumentException, ArrayIndexOutOfBoundsException, ClassCastException y IOExceptio son excepciones del tipo runtimeException y por tanto no es necesario capturarlas ni indicar throws en el método en el que se provoquen.
38. La genericidad se considera una característica opcional de los lenguajes.
39. La genericidad se considera una característica opcional de los lenguajes OO.
40. No se puede derivar una clase genérica de una clase no genérica.
41. En JAVA no podemos crear interfaces genéricas.

42. En los métodos genéricos solo podremos usar los métodos definidos en Object.
43. En la genericidad restringida solo podremos usar los métodos de Object al implementar los métodos.
44. La API de reflexión de JAVA incluye métodos para obtener la signatura de todos los métodos.
45. La reflexión permite que un programa obtenga información sobre sí mismo en tiempo de ejecución.
46. Para usar reflexión en JAVA hemos de conocer el nombre de las clases en tiempo de compilación.
47. En JAVA el concepto de meta-clase se presenta con la clase Class.
48. Con el uso de la reflexión solo podemos invocar métodos de instancia.
49. La reflexión solo es útil para trabajar con componentes JavaBeans.
50. La reflexión es demasiado compleja para usarla en aplicaciones de propósito general.
51. La reflexión reduce el rendimiento de las aplicaciones.
52. La reflexión no puede usarse en aplicaciones certificadas con estándar 100% Pure Java.
53. Mediante reflexión podemos saber cuáles son las clases derivadas de una clase dada.
54. Mediante reflexión no podemos saber cuál es el método que se está ejecutando en un determinado momento.
55. Podemos usar reflexión para encontrar un método heredado (solo hacia arriba) y reducir código condicional.
56. La refactorización debe hacerse siempre apoyándonos en un conjunto de tests completo y robusto.
57. Una clase con gran número de métodos y atributos es candidata a ser refactorizada.
58. Los métodos grandes (con muchas instrucciones) son estructuras que sugieren la posibilidad de una refactorización.
59. En la refactorización se permite que cambie la estructura interna de un sistema software aunque varié su comportamiento externo.
60. Un ejemplo de refactorización seria mover un método arriba o abajo en la jerarquía de herencia.
61. Los frameworks no contienen implementación alguna, únicamente un conjunto de interfaces que deben ser implementados por el usuario del framework.
62. Un framework invoca mediante enlace dinámico a nuestra implementación de interfaces propios de framework.
63. Hibernate es un framework para congelar en memoria el estado de nuestra aplicación.
64. JDBC es un framework de JAVA que usan los fabricantes de sistemas de gestión de bases de datos para ofrecer un acceso estandarizado a las bases de datos.
65. El usuario de un framework implementa al componente declarado de los interfaces de framework mediante herencia de implementación.
66. El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de interfaz.
67. Un framework es un conjunto de clases cuyos métodos invocamos para que realicen unas tareas a modo de caja negra.
68. En JAVA el acceso a bases de datos se hace con librerías propietarias de SGBD cuyos interfaces no tienen ningún estándar.
69. Para poder utilizar un framework, es necesario crear clases que implementen todas las interfaces declaradas en el framework.
70. Una librería de clases proporciona una funcionalidad completa, es decir, no requiere que el usuario implemente o herede nada.
71. El polimorfismo es una forma de reflexión.
72. En el proceso de diseño de un sistema de software se debería intentar aumentar el acoplamiento y la cohesión.
73. Cuando diseñamos sistema OO las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación.
74. Todo espacio de nombre define su propio ámbito, distinto de cualquier otro.
75. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea.
76. En el diseño mediante tarjetas CRC utilizamos una tarjeta para cada clase.
77. Una tarjeta CRC contiene el nombre de una clase, su lista de responsabilidades y su lista de colaboradores.
78. La robustez de un sistema software es un parámetro de calidad intrínseco.
79. El principio abierto-cerrado indica que un componente software debe estar abierto a su extensión y cerrado a su modificación.
80. Con el diseño OO es perfectamente posible y deseable hacer uso de variables globales.
81. En el diseño por contrato son dos componentes fundamentes las pre y pos condiciones.

82. Los métodos definidos en una clase derivada nunca pueden acceder a las propiedades privadas de una clase base.
83. Una clase abstracta se caracteriza por no tener definido ningún constructor.
84. La siguiente clase: class S {public: S(); S(const S &s); virtual ~S();}; constituye una interfaz en C++.
85. Desde un método de una clase derivada nunca puede invocarse a un método implementado con la idéntica signatura de una de sus clases base.
86. Los métodos virtuales son métodos abstractos.
87. Los métodos abstractos son métodos con enlace dinámico.
88. Los constructores de las clases abstractas son métodos con enlace dinámico.
89. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
90. Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica.
91. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica.
92. En C++ no podemos hacer sobrecarga de operadores para tipos predefinidos.
93. En C++, si no se define un constructor sin argumentos explícitamente, el compilador proporciona uno por defecto.
94. En la misma clase, podemos definir constructores con distinta visibilidad.
95. Si no se captura una excepción lanzada por un método, el programa no advierte que ha ocurrido algún error y continúa su ejecución normalmente.
96. La instrucción throw permite lanzar como excepción cualquier tipo de dato.
97. Las funciones genéricas no se pueden sobrecargar.
98. Dada una clase genérica, se pueden derivar de ella clases no genéricas.
99. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor.
100. Una clase interfaz no puede tener atributos de instancia. Una clase abstracta sí puede tenerlos.
101. La herencia de interfaz se implementa mediante herencia pública.
102. La herencia pública permite a los métodos definidos en una clase derivada acceder a las propiedades privadas de la clase base.
103. Una clase abstracta se caracteriza por declarar al menos un método abstracto.
104. La siguiente clase: class S {public: Object +o;}; constituye una clase interfaz en C++.
105. Los métodos abstractos siempre tienen enlace dinámico.
106. Los constructores siempre son métodos virtuales.
107. Un método tiene polimorfismo puro cuando tiene como argumentos al menos una variable polimórfica.
108. Un atributo declarado con visibilidad protegida en una clase A es accesible desde clases definidas en el mismo espacio de nombres donde se definió A.
109. Una de las características básicas de una lengua orientada a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
110. Una operación de clase sólo puede ser invocada mediante objetos constantes.
111. En C++, si no se define ningún constructor, el compilador proporciona por defecto uno sin argumentos.
112. Una clase interfaz no puede tener instancias.
113. Una clase abstracta siempre tiene como clase base una clase interfaz.
114. No se puede definir un bloque catch sin su correspondiente bloque try.
115. Tras la ejecución de un bloque catch, termina la ejecución del programa.
116. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo.
117. El downcasting estático siempre es seguro.
118. El principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases.
119. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes.
120. Una operación de clase no puede tener enlace dinámico.
121. La herencia protegida permite a los métodos de la clase derivada acceder a las propiedades privadas de la clase base.
122. La siguiente clase en C++: class S {public: virtual ~S()=0;}; define una interfaz.
123. Los métodos virtuales siempre tienen enlace dinámico.
124. Los constructores siempre tienen enlace dinámico.
125. En C++, un atributo de la clase debe declararse dentro de la clase con el modificador static.
126. Un método tiene polimorfismo puro cuando devuelve una variable polimórfica.

127. Dada la siguiente definición de clase en C++: class TClase {public: TClase (int dim); private: int var1;}; La instrucción TClase c1; no da error de compilación e invoca al constructor por defecto.
128. Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
129. Hablamos de encapsulación cuando agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos.
130. En C++ los constructores se pueden declarar como métodos virtuales.
131. En C++, es obligatorio especificar qué excepciones lanza una función mediante una cláusula throw tras la declaración de la función.
132. Una clase interfaz no debe tener atributos de instancia. Una clase abstracta sí puede tenerlos.
133. Un mensaje tiene cero o más argumentos. Por el contrario, una llamada a procedimiento/función tiene uno o más argumentos.
134. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña.
135. En una agregación, la existencia de un objeto parte depende de la existencia del objeto todo que lo contiene.
136. Una forma de mejorar el diseño de un sistema es reducir su acoplamiento y aumentar su cohesión.
137. Los iniciadores en C++ tienen el formato nombreAtributo(valor) [o nombre_atributo(valor), otra pregunta] y se colocan entre la lista de argumentos y el cuerpo de cualquier método, permitiendo asignar valores a los atributos de instancia de la clase en la que se define dicho método.
138. Un atributo estático ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de cualquier clase derivada de ella.
139. La herencia múltiple se produce cuando de una misma clase base se heredan varias clases derivadas.
140. Un atributo protegido en la clase base es también protegido en cualquier clase que derive de dicha base, independientemente del tipo de herencia utilizado.
141. Cuando se crea un objeto de la clase D de que deriva de una clase B, el orden de ejecución de los constructores es siempre B() D().
142. Una clase abstracta es una clase que no permite instancias de ella.
143. Una interfaz es una clase abstracta con al menos un método de instancia abstracto.
144. Un método de clase (estático) no puede tener enlace dinámico.
145. Un método virtual en C++ siempre tiene enlace estático.
146. La llamada a un método sobrescrito se resuelve en tiempo de compilación.
147. Los métodos definidos en una clase genérica son a su vez genéricos.
148. La instrucción throw (en C++) sólo permite lanzar objetos de la clase exception o de clases derivadas de ella.
149. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro.
150. En una composición de un objeto componente puede formar parte de más de un compuesto.
151. Los destructores en C++ pueden aceptar cualquier número de parámetros.
152. Los TAD's representan una perspectiva orientada a datos, mientas que los objetos reflejan una perspectiva orientada a servicios.
153. Un atributo privado en la clase base no es directamente accesible en la clase derivada, independientemente del tipo de herencia utilizado.
154. Un constructor de copia acepta cualquier tipo de modificador (static, const, public, etc).
155. Una clase es una especificación abstracta de una estructura de datos y de las operaciones que se pueden realizar con ella.
156. Una operación constante sólo puede ser invocada por un objeto constante.
157. El puntero this es un puntero constante al objeto que recibe el mensaje.
158. Implementar la forma canónica ortodoxa de una clase es una condición necesaria (aunque no suficiente para controlar que el valor de un atributo de clase que cuenta el número de instancias de dicha clase esté siempre en un estado consistente.
159. La existencia de una relación todo-parte entre dos clases implica necesariamente que el objeto todo maneja la creación/destrucción de los objetos parte.
160. La forma canónica de la clase está formada por el constructor, el constructor de copia, el destructor y el operador de asignación.
161. Si la clase no proporciona un constructor sin parámetros, el compilador en C++ genera uno de oficio.
162. Tanto composición como herencia son mecanismos de reutilización del software.

163. Un atributo de clase debe declararse dentro de la clase con el modificador const.
164. Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito.
165. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada, pero no modificar los métodos heredados de la clase base.
166. Una interfaz es la definición de un protocolo para cierto comportamiento, sin especificar la implementación de dicho comportamiento.
167. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea.
168. En el diseño mediante tarjetas CRC, utilizamos una tarjeta por cada clase.
169. La sobrescritura es una forma de polimorfismo.
170. El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utiliza.
171. La inversión de control en los frameworks es posible gracias al enlace dinámico de métodos.
172. El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de implementación.
173. La instanciación mediante reflexión de un objeto de la clase Rectángulo del paquete pack se realiza así: Class.forName("pack", "Rectangulo");
174. En Java el enlace por defecto de métodos de instancia es estático.
175. En Java no se pueden derivar clases genéricas de otras clases genéricas.
176. En Java las sentencias de un bloque 'finally' solamente se ejecutan cuando se ha producido una excepción y no la hemos capturado en un bloque 'catch'.
177. En el paradigma orientado a objetos, un objeto siempre es instancia de alguna clase.
178. El método invocado por un objeto en respuesta a un mensaje viene siempre determinado, entre otras cosas por la clase del objeto receptor en tiempo de compilación.
179. En el paradigma orientado a objetos, un programa es un conjunto de objetos que se comunican mediante el paso de mensajes.
180. El siguiente código en Java define una interfaz: interface S {}
181. Sea un método llamado glue(), sin argumentos, implementado en una superclase y sobrescrito en una de sus subclases. Siempre podremos invocar a la implementación del método en la superclase desde la implementación del método en la subclase usando la instrucción super.glue();
182. Los métodos abstractos siempre tienen enlace dinámico.
183. Dado un objeto, mediante reflexión no se puede obtener la lista de todos los métodos declarados en su clase, tanto públicos como privados.
184. Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación.
185. La refactorización nunca produce cambios en las interfaces de las clases.
186. Las sentencias 'switch' son un caso de código sospechoso (código con mal olor).
187. El código duplicado es un caso de código sospechoso en el que se aconseja el uso de técnicas de refactorización para eliminarlo.
188. La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización.
189. "Los métodos que usan referencias a clases base deben ser capaces de usar objetos de clases derivadas sin saberlo" es una posible formulación del principio de inversión de dependencias.
190. El enlace de la invocación a un método sobrescrito se produce en tiempo de ejecución en función del tipo en tiempo de ejecución del receptor del mensaje.
191. this es un ejemplo de variable polimórfica en Java.
192. En Java el downcasting siempre se realiza en tiempo de ejecución.
193. En Java, un atributo de clase debe declararse dentro de la clase con el modificador static.
194. En Java, gracias a la sobrecarga de operadores podemos crear nuevos operadores en el lenguaje.
195. Si en una clase no se declara, implícita o explícitamente, un constructor por defecto, no se pueden crear instancias de esa clase.
196. Una de las características básicas de unos lenguajes orientados a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
197. La instrucción throw en Java sólo permite lanzar objetos que son instancias de la clase java.lang.Throwable o de clases derivadas de ésta.
198. En Java, la instrucción throw no se puede usan dentro de un bloque catch.
199. Los métodos genéricos no se pueden sobrecargar ni sobrescribir.

200. Una clase abstracta siempre tiene como clase base una clase interfaz.
201. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor.
202. C++ sólo permite heredar cuando la clase hija es un subtipo de la clase padre (herencia como implementación de la generalización).
203. El constructor de copia permite argumentos tanto por referencia como por valor.
204. El estado de un objeto es el conjunto de valores de los atributos y métodos que han sido invocados sobre él.
205. En las jerarquías de herencia en C++, si la clase base define un operador de asignación y la clase derivada no lo redefine, al invocar a dicho operador con objetos de la clase derivada se invocará al código de la clase base.
206. La herencia es más flexible en cuanto a posibles cambios en la naturaleza de los objetos que la composición.
207. La herencia privada en C++ es un tipo de herencia insegura porque no preserva el principio de encapsulación.
208. La relación de herencia es una relación de clases no persistente.
209. Un objeto se caracteriza por poseer un estado, un comportamiento y una identidad.
210. Una clase abstracta siempre tiene que tener alguna clase que derive de ella.
211. Tanto la herencia protegida como la privada permiten a una clase derivada acceder a las propiedades privadas de la clase base
212. Una clase abstracta se caracteriza por no tener atributos.
213. La siguiente clase: class S {public: virtual ~S()=0; virtual void f()=0;}; constituye una interfaz en C++.
214. Desde un método de una clase derivada nunca puede invocarse un método implementado con idéntica signatura de una de sus clases base.
215. Los métodos con enlace dinámico son abstractos.
216. Los constructores de las clases abstractas son siempre métodos abstractos.
217. Un atributo de clase debe tener visibilidad pública para poder ser accedido por los objetos de la clase.
218. Un método sobrecargado es aquel que tiene más de una implementación, diferenciando cada una por el ámbito en el que se declara, o por el número, orden y tipo de argumentos que admite.
219. Un método abstracto es un método con polimorfismo puro.
220. Todo espacio de nombres define su propio ámbito, distinto de cualquier otro ámbito.
221. En la sobrecarga de operadores binarios para objetos de una determinada clase, si se sobrecarga como función miembro, el operando de la izquierda es siempre un objeto de la clase.
222. La genericidad se considera una característica opcional de los lenguajes orientados a objetos.
223. Hablamos de encapsulación cuando diferenciamos entre interfaz e implementación.
224. Una operación de clase no es una función miembro de la clase.
225. Los constructores siempre deben tener visibilidad pública.
226. En C++, si no se captura una excepción lanzada por un método, se produce un error de compilación.
227. En C++, la cláusula throw() tras la declaración de una función indica que ésta no lanza ninguna excepción.
228. Dada una clase genérica, no puede ser utilizada como clase base en herencia múltiple.
229. De una clase interfaz no se pueden crear instancias. De una clase abstracta sí.
230. Cuando usamos la varianza estamos haciendo un uso inseguro de la herencia de implementación.
231. En la sobrecarga de operadores como función miembro, el operando de la izquierda puede ser un objeto de la clase o cualquier otro tipo de objeto, mientras que en las funciones amigas siempre es un objeto de la clase.
232. Cuando creamos un objeto en C++ mediante una variable automática el constrictor se autoinvoca. También se autoinvoca el destructor del mismo al salir del ámbito de la función donde se creó.
233. Si para una clase genérica llamada Pila<T> declaramos las siguientes funciones: virtual void apilar(T* pt); virtual void apilar(T t); En caso de sobrescritura ya que el tipo devuelto es el mismo en ambos métodos.
234. De una clase abstracta se pueden crear referencias a objetos.
235. En un atributo de clase se reserva espacio en memoria para una copia de él por cada objeto de su clase creado.
236. Declarar un dato miembro de una clase como private indica que sólo puede acceder a ese atributo desde las funciones miembro de la clase.
237. Una clase abstracta siempre tiene que tener alguna clase que derive de ella.
240. El polimorfismo debido a la sobrecarga de funciones siempre se da en relaciones de herencia.

241. A los atributos de instancia si son constantes se les asigna su valor inicial fuera de la clase.
242. Toda sentencia que aparece después del punto del programa en el que ocurre una excepción, en ningún caso se ejecuta.
243. Si utilizamos los mecanismos de manejo de excepciones disminuye la eficiencia del programa incluso si no se llega a lanzar nunca una excepción.
244. Cuando se captura una excepción y ésta pertenece a una jerarquía de clases, el primer bloque catch debe comenzar con la clase del nivel más alto de la jerarquía.
245. Hablamos de shadowing cuando el método a invocar se decide en tiempo de compilación.
246. A diferencia de otros lenguajes de programación en C++ la sobreesctritura en relaciones de herencia se debe indiciar de forma explícita en la clase padre.
245. La encapsulación es un mecanismo que permite separar de forma estricta interfaz e implementación.
246. La interpretación de un mismo mensaje puede variar en función del receptor del mismo y/o del tipo de información adicional que lo acompaña.
247. Un atributo de clase público puede ser accedido desde fuera de la clase a través de un objeto de la clase, un puntero o referencia al mismo o mediante el nombre de la clase seguido del operador de ámbito.
248. Es posibles definir un constructor de copia invocando en su cuerpo al operador de asignación.
249. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO.
250. Para que se pueda realizar una herencia múltiple en C++, es necesario que no coincida ninguno de los nombre de atributo entre las clases bases involucradas.
251. La signatura de tipo de un método incluye el tipo devuelto por el método.
252. En el principio de sustitución implica una coerción entre tipos de una misma jerarquía de clases.
253. En C++, un destructor no puede ser virtual.
254. El puntero this no es una variable polimórfica porque es constante y no se puede cambiar su valor.
255. Las instrucciones para el manejo de excepciones nos permiten mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores.
1. La herencia de interfaz se implementa mediante herencia pública.
2. Una interfaz no puede tener atributos de instancia. Una clase abstracta si puede tenerlos.
3. No se puede definir un bloque catch sin su correspondiente bloque try.
4. Una variable polimórfica puede hacer referencia a diferentes tipos de objetos en diferentes instantes de tiempo.
5. El downcasting siempre es seguro.
6. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes.
7. En el diseño mediante tarjetas CRC, utilizamos una tarjeta por cada jerarquía de herencia.
8. Un espacio de nombres es un ámbito con nombre.
9. Un sistema de tipos de un lenguaje asocia a cada tipo una expresión.
10.Hacer que el código sea más fácil de entender no es un motivo suficiente para refactorizarlo.
11.En Java, los tipos genéricos sólo se pueden aplicar a clases e interfaces.
12.En Java, una clase genérica puede ser parametrizada empleado más de un tipo.
13.Sean dos clases Base e Hija. La clase Hija hereda de Base. En Java, cuando asignamos un objeto de la clase Hija a una referencia a Base haciendo conversión de tipo explícita estamos haciendo object slicing.
14.Una de las principales fuentes de problemas cuando utilizamos herencia múltiple es que las clases bases hereden de un ancestro común.
15. Los lenguajes de programación soportan el reemplazo y el refinamiento como métodos de sobrescritura, pero no hay ningún lenguaje que proporcione ambas técnicas (JAVA solo soporta reemplazo y C++ sólo soporta refinamiento).
16.Una interfaz puede implementar otra interfaz.
17.En java es obligatorio indicar que un método de una clase derivada sobrescribe un método de la clase base con la misma signatura.
18. En la sobrecarga basada en ámbito los métodos pueden diferir únicamente en el tipo devuelto.
19.En la herencia pública la clase derivada podrá acceder a los atributos privados de la clase base de la que hereda.
20. Una clase abstracta se caracteriza por no tener ningún constructor.
21.El cambio de una condicional por el uso de polimorfismo es un ejemplo de refactorización.
22.Un atributo siempre tiene visibilidad pública.
23.El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utilice.

24.El usuario de un framework implementa el comportamiento declarado en los interfaces del framework mediante herencia de implementación

25. El proceso de diseño de un sistema software se debería intentar aumentar la cohesión y reducir el acoplamiento.
26.Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar cerradas a la extensión y abiertas a la modificación.
27.La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización.
28.Existe un catálogo de refactorizaciones comunes de forma que el programador no se ve obligado a usar su propio criterio y metodología para refactorizar código.
29.Arraylist es una implementación en el Java Collection Framework de la interfaz List.
30.Con el uso de reflexión solo podemos invocar métodos de instancia.
31.La siguiente clase class S{ public Object obj; } constituye una interfaz en Java.
32.Desde un método de una clase derivada solamente puede invocarse un método implementado con idéntica signatura de una de sus superclases si el método en la clase base tiene enlace dinámico.
33.Los métodos con enlace dinámico son métodos abstractos.
34.Un método sobrecargado es aquel que recibe como argumento al menos una variable polimórfica.
35.La genericidad se considera una característica opcional de los lenguajes orientados a objetos.
36.Una de las características básicas de un lenguaje orientado a objetos es que todos los objetos de la misma clase pueden recibir los mismos mensajes.
37.En java, podemos definir constructores con distinta visibilidad en la misma clase.
38.Los métodos genéricos no se puede sobre sobrecargar ni sobrescribir.
39.Una interfaz no tiene instancia. Por ejemplo, dada la interfaz Comprable en Java, no podemos hacer new Comparable().
40.Una interfaz en Java obliga a que las clases no abstractas que la implementan definan todos los métodos que la interfaz declara.

PREGUNTAS 2016

1. Todas las clases que representan excepciones en Java tienen a la clase Object como una de sus superclases.

2. La clase "ERoturaStock" es una clase de tipo excepción y en el UML tiene una relación de dependencia con "Producto" porque usa información proporcionada por ésta última.

3. Viendo el diagrama, podemos decir que diferentes objetos de tipo "Ticket" pueden compartir objetos de tipo "LineaTicket" a través de la relación del UML.

4. El método invocado por un objeto en respuesta a un mensaje viene dado entre otras cosas por la clase del objeto receptor en tiempo de ejecución si el método invocado tiene enlace dinámico.

5. Si en una clase no se declara implícita o explícitamente un constructor por defecto, no se pueden crear instancias de esa clase.

6. Si en una clase no se declara implícita o explícitamente ningún constructor, no se pueden crear instancias de esa clase.

7. En java un objeto de clase Class es creado en tiempo de ejecución cada vez que se crea un nuevo objeto en el programa.

8. La inversión de control implica que al usar un framework de java el método main(), es decir el punto de entrada de la aplicación, formara siempre parte del código del framework.

Tema 1. Introducción al Paradigma Orientado a Objetos

1. La abstracción es una supresión intencionada (u ocultación) de algunos detalles de un proceso o artefacto, con el fin de destacar más claramente otros aspectos, detalles o estructuras. V
2. Destacamos 3 Niveles de Abstracción. V
3. La Perspectiva Funcional tiene el ensamblador, procedimientos y Módulos. V
4. La Perspectiva de Datos los paquetes y Tipos abstractos de datos. V
5. La Perspectiva de Servicios los objetos (TAD, herencia, polimorfismo). V
6. El paradigma se define como la forma de entender y representar la realidad. V
7. Hablamos de enlace estático cuando nos referimos a tiempo ejecución. F
8. Hablamos de enlace dinámico cuando nos referimos a tiempo de compilación. F
9. Hablamos de enlace estático para referirnos a tiempo de compilación y dinámico para referirnos a tiempo de ejecución. V
10. La corrección y la robustez son parámetros de calidad extrínsecos (Fiabilidad). V
11. Extensibilidad, Reutilización y Mantenibilidad son parámetros de calidad Intrínsecos. V
12. La robustez de un sistema software es un parámetro de calidad intrínseco. F

Tema 2. Conceptos Básicos de la Programación Orientada a Objetos

1. Un atributo de clase ocupa una zona de memoria que es compartida por todos los objetos de la clase en la que se define, aunque no por los objetos de la clase derivada. F
2. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación de instancia de los objetos pasados como parámetro. F
3. En el cuerpo de una operación de clase no se puede acceder a ningún atributo/operación del objeto receptor del mensaje. V
4. En el cuerpo de una operación de clase se puede acceder únicamente a atributos/operaciones de clase. V

Constructores y Destructores

5. Si en una clase no se declara, implícita o explícitamente, un constructor por defecto, no se pueden crear instancias de esa clase
6. El objetivo de un constructor es crear e inicializar objetos. V
7. En la misma clase podemos definir un constructor con distinta visibilidad. V
8. Un constructor acepta cualquier tipo de modificación (static, const, public, etc). F
   //Siempre deberá ser public
9. En C++ el Constructor de Copia permite tanto argumentos por referencia como por valor. F //Exclusivamente por valor
10. En C++ el Constructor Copia se invoca de forma implícita cuando devolvemos un valor o pasamos un objeto como argumento por valor. V
    //Esto es así. Tu en C++ cuando en un método devuelves un valor, se llama automáticamente al constructor de copia.

11. En C++, si no se define un constructor sin argumentos explícitamente, el compilador proporciona uno por defecto. F
//Proporcionaría uno por defecto si únicamente no hubiera un constructor.

12. Es conveniente definir siempre un Constructor por Defecto que permita la inicialización sin parámetros de un objeto, donde los atributos se inicalizan con valores por defecto. V

```
//Ejemplo de constructor por defecto en C++:
TvectorCalendario::TvectorCalendario() {
valor=NULL; //valor (atributo) se inicializa a null por defecto.
}
```

13. C++ siempre proporciona un Constructor por Defecto (sin parámetros), Java no, F.
//Java también

14. En Java y en C++, si no se define un Constructor, el compilador genera uno por Defecto con visibilidad pública. V

15. Los Destructores en C++ pueden aceptar cualquier número de parámetros. F
//Creo que deben de recibir los mismos que en el constructor por defecto.

Forma Canónica

16. La Forma Canónica en Java incluye el constructor, constructor de copia, Equals, ToString, Hashcode y Clone. F //(6 métodos).

17. La Forma canónica en Java incluye el constructor, Equals, HashCode, ToString y Clone. V

18. La Forma Canónica en C++ incluye el constructor por defecto, el constructor de copia, el destructor, el operador asignación y cualquier otro constructor parametrizado. F

19. La Forma Canónica en C++ incluye el constructor por defecto, constructor de copia, el destructor y el operador de asignación. V

//Tengo dudas de esta porque en la asignatura de PED (C++) se especifica que también tiene un constructor que recibe argumentos.

20. El método Clone() proporciona un Deep Copy en Java, que copia bit a bit el contenido del objeto pasado por parámetro. F

21. Tanto en Java como C++ proporciona un método Clone() que devuelve una Shallow Copy del objeto pasado por parámetro. F

Relaciones Todo-Parte

22. Las relaciones todo-parte (tiene-un) son la agregación y la composición. V

23. Ni la asociación ni la agregación manejan la creación/destrucción de objetos tipo parte. V

24. La Composición maneja la creación/destrucción de los objetos tipo parte. V

25. Las relaciones todo-parte son relaciones persistentes como la herencia. V
26. En una agregación, la existencia de un objeto parte depende de la existencia del objeto todo que lo contiene. F
27. La agregación es una relación mucho más restrictiva que la herencia. V
28. En una composición un objeto componente puede formar parte de más de un objeto compuesto. F
29. La agregación y la asociación pueden ser bidireccionales. F
30. La única relación no persistente es la relación de uso, ya que no se materializa mediante referencias. V
31. Todo son relaciones de clase excepto la relación de uso que es relación de objeto. F

Tema 3. Introducción al Diseño Orientado a Objetos

Tema 4. Gestión de Errores

1. Las instrucciones para el manejo de excepciones nos permiten mezclar el código que describe el funcionamiento normal de un programa con el código encargado del tratamiento de errores. F
2. Los bloques try-catch no se pueden anidar. F
3. En Java, si no se captura una excepción lanzada por un método da error de compilación. F
4. Si no se captura una excepción lanzada por un método, el programa no advierte de error y continúa su ejecución normalmente. F
5. La instrucción throw en Java sólo permite lanzar objetos que son de la clase throwable o clases derivadas de ésta. V
6. En java, siempre es obligatorio especificar que excepciones verificadas (chequed exception) lanza un método mediante una cláusula throws tras la lista de argumentos. F
7. Si se produce una excepción el método que la provoca se cancela y se continúa la ejecución en el método que llamó a este. V
8. Si se produce una excepción en un constructor el objeto se construirá con los valores por defecto. F
9. Todas las excepciones son chequed exception salvo las Runtime que son unchequed. V
10. En Java, la instrucción throw no se puede usar dentro de un bloque catch.
11. La cláusula throws de un método incluirá todas las excepciones unchequed que puedan producirse en esta y estén dentro del bloque try catch que las capture. F

12. Si en el bloque try de un catch, que captura chequed exception, es imposible que se produzca dicha excepción el código no compilará, indicando que el bloque catch es inalcanzable. V
13. Si en la sentencia thorws de un método especificamos una excepción que es imposible que se lance en el cuerpo de dicho método el compilador no compilará, indicando que el método nunca lanzará la excepción. F
14. El orden de las excepciones en los bloques catch no es relevante. F
15. Podemos poner en bloque finally sin poner bloques catch. F
16. El bloque Finally sólo se ejecutará si se pone alguna excepción en el bloque try al que esté asociado.
17. En java las sentencias de un bloque 'finally' solamente se ejecutan cuando se ha producido una excepción y no la hemos capturado en un bloque catch.
18. IllegalArgumentException, ArrayOutofBounsException, ClassCastException y IOexception son excepciones del tipo runtimeException y por tanto no es necesario capturarlas ni indicar throws en el método en que se propaguen. F

Tema 5. Herencia

Derivación

1. Los métodos definidos en una clase derivada nunca pueden acceder a las propiedades privadas de una clase base. V
2. Desde un método de una clase derivada nunca puede invocarse un método implementando con idéntica signatura de una de sus clases base. F
3. Un atributo privado en la clase base no es directamente accesible en las derivadas sea cual sea el tipo de herencia, tanto en Java como en C++. V

4. La sentencia derivada d=d2.clone(); es correcta si d y d2 son de tipo Derivada. V
5. Una clase derivada puede añadir nuevos métodos/atributos propios de la clase derivada pero no modificar los atributos heredados de la clase base. F
6. No se puede derivar de una clase no genérica a una genérica. F
7. No se puede derivar una clase genérica a otra no genérica. F
8. Un atributo public en base es accesible en main desde un objeto derivado sea cual sea el tipo de herencia. F

Abstracción

9. Una clase abstracta se caracteriza por no tener definido ningún constructor. F
10. Los métodos abstractos son métodos con enlace dinámico. V
11. Los constructores de las clases abstractas son métodos con enlace dinámico. F
12. En Java los métodos abstractos pueden tener implementación pero siempre harán abstracta a la clase que los contenga. F
13. Ser abstracto implica tener enlace dinámico. V
14. Un método con enlace dinámico siempre es abstracto. F
15. Los métodos virtuales son métodos abstractos. F
16. Una clase abstracta siempre tiene que tener alguna clase que derive de ella. F
17. De una clase abstracta no se pueden crear instancias, excepto si se declara explícitamente algún constructor. F
18. Una clase abstracta siempre tiene como clase base una interfaz. F

Otras Preguntas

19. En Java la herencia solo puede ser pública, mientras que en C++ sólo puede ser derivada o protegida. F
20. La herencia pública implica herencia de implementación y de interfaz mientras que la herencia privada o protegida implica herencia de implementación pero no de interfaz. V
21. Para que se pueda realizar herencia múltiple en C++, es necesario que no coincida ninguno de los nombres de atributo entre las clases base involucradas. F
22. La herencia de implementación siempre implica herencia de interfaz. F
23. En C++, los métodos virtuales puros, pueden tener implementación pero siempre harán abstracta a la clase que los contenga. V

Tema 6. Polimorfismo

Modo en el que los lenguajes OO implementan el concepto de polisemia. V

Sobrecarga

1. Definir dos métodos que se llamen igual pero tengan distinto tipo se llama Sobrecarga. F //Creo que también igual el tipo.

2. En Java, gracias a la sobrecarga de operadores podemos crear nuevos operadores de lenguaje.
3. Una clase abstracta se caracteriza por declarar al menos un método abstracto. F
4. En C++ no se pueden definir sobrecargas de operadores fuera de la clase a no ser que sean funciones amigas. F
5. Para sobrecargar dos métodos dentro del mismo ámbito es necesario que difieran en el número, tipo y orden de sus argumentos. F
6. En la sobrecarga nunca se podrá cambiar el tipo devuelto por un método. F
7. Los métodos de sobrecarga son el refinamiento y el remplazo. F
8. Un método sobrecargado es aquel que recibe como argumento una variable polimórfica. F
9. En C++ no podemos hacer sobrecarga de operadores para tipos predefinidos. V
10. La sobrecarga basada en ámbito permite definir el mismo método en dos clases diferentes. V

Genericidad

11. La genericidad es una propiedad que permite definir a una clase o una función sin tener que especificar el tipo de datos o algunos de sus miembros o argumentos. V
12. La genericidad es un tipo de polimorfismo. V
13. Los métodos genéricos no se pueden sobrecargar ni sobreescribir.
14. Su principal utilidad es la de agrupar variables cuyo tipo base no está predeterminado. V
15. 2 tipos de genéricos: Clases Genéricas y Métodos Genéricos. V
16. La genericidad se considera una característica opcional de los lenguajes. V
17. La genericidad se considera una característica opcional de los LOO. V
18. No se puede derivar una clase genérica de otra no genérica. F
19. En Java no se pueden derivar clases genéricas de otras genéricas. F
20. Se pueden derivar clases genéricas y no genéricas de otras genéricas. V
21. En Java no podemos crear interfaces genéricas. F
22. En los métodos genéricos sólo podemos usar los métodos definidos en Object. V
23. En la genericidad restringida sólo podemos usar los métodos definidos en Object. F
24. La genericidad restringida permite indicar que los tipos genéricos pertenezcan a una determinada jerarquía de herencia. V
25. Las funciones genéricas no se pueden sobrecargar. F
26. Dada una clase genérica, se pueden derivar de ella clases no genéricas. V

Variables Polimórficas

27. This es un ejemplo de variable polimórfica en Java.
28. En Java toda clase es potencialmente polimórfica , en C++ sólo si posee algún método virtual. V
29. Un método sobrecargado es aquel que recibe como argumento una variable polimórfica. F
30. Una variable polimórfica puede hacer referencia a distintos tipos de objetos en diferentes instantes de tiempo. V

Otras preguntas de Polimorfismo

1. La sobreescritura es una forma de polimorfismo.
2. En Java, los métodos de instancia con polimorfismo puro pero no abstractos tienen enlace dinámico por defecto

Tema 7. Reflexión.

1. La Reflexión es una infraestructura del lenguaje que permite a un programa conocerse y manipularse a sí mismo en tiempo de ejecución. V
2. Para usar en reflexión en Java hemos de conocer el nombre de las clases en tiempo de compilación. F
3. La Reflexión puede usarse para construir nuevos objetos y arrays, acceder y modificar atributos de instancia, invocar métodos de instancia y estáticos… V.
4. La API de reflexión en Java incluye métodos para obtener la signatura de todos los métodos. V
5. Toda clase que se carga en la JVM tiene asociado un objeto de tipo Class. V
6. En java el concepto de meta-clase se representa con la clase Class. V
7. Con el uso de reflexión sólo podemos invocar métodos de instancia. F
8. Mediante reflexión podemos saber cuáles son las clases derivadas de una clase dada. V
9. Mediante reflexión no podemos saber cuál es el método que se está ejecutando en un momento dado. V
10. Podemos usar reflexión para encontrar un método heredado (sólo hacia arriba) y reducir código condicional. V
11. La reflexión sólo es útil para trabajar con componentes JavaBeans. F
12. La reflexión es demasiado compleja para usarla en aplicaciones de propósito general. F
13. La reflexión reduce el rendimiento de las aplicaciones. F
14. La reflexión no puede usarse en aplicaciones certificadas con el estándar 100% Pure Java. F
15. Con el objeto Method podemos obtener su nombre y lista de parámetros e invocarlo. V
16. Con el objeto Method podemos obtener un método a partir de una signatura, u obtener una lista de todos los métodos con esa signatura. V

Tema 8. Frameworks

1. La inversión de control en los frameworks es posible gracias al enlace dinámico de métodos.

2. Un Framework es un esqueleto para fines específicos que debemos completar y personalizar mediante la implementación de interfaces, herencia de clases abstractas y ficheros de configuración. V
3. Una librería de clases proporciona una funcionalidad completa, es decir, no requiere que el usuario implemente nada. V
4. Los Frameworks no contienen implementación alguna, únicamente un conjunto de interfaces que debes ser implementados por el usuario del framework. F
5. Un framework invoca mediante enlace dinámico a nuestra implementación de interfaces propios de framework. V
6. El usuario de un framework implementa al componente declarado de los interfaces de framework mediante herencia de implementación. F
7. El usuario de un framework implementa al componente declarado de los interfaces de framework mediante herencia de interfaz. V
8. Un framework es un conjunto de clases cuyos métodos invocamos para que realicen varias tareas a modo de caja negra. F
9. Para poder utilizar un framework, es necesario crear clases que implementen todas las interfaces declaradas en el framework. V
10. JCF (Java Collection Framework) tiene un conjunto de clases en el JDK representando tipos de datos abstractos. V
11. Set, List and Map son las interfaces del JCF, y set<List>, hashMap… son implementaciones de esas interfaces. V
12. JDBC (Java Data Base Collection) es un framework que permite conectar Java con Bases de Datos. V
13. Hibernate es un framework para congelar en memoria el estado de nuestra aplicación. F
14. Hibernate está asociado con objetos en Bases de Datos. V
15. GWT (Google Web Toolkit) es una generación de aplicaciones ricas web mediante la generación de JavaScript a partir de Java. V
16. Apache Lucene es un motor de búsqueda para la recuperación de información. V

Tema 9. Refactorización

1. La refactorización nunca produce cambios en las interfaces de las clases
2. Proceso que maneja la estructura interna de un sistema de software de forma que su comportamiento no varía. V
3. Refactorizar es una forma sistemática y segura de realizar cambios en una aplicación con el fin de hacerla más fácil de comprender. V
4. Es una forma sistemática de introducir mejoras en el código que minimiza la posibilidad de introducir errores en él. V
5. En la refactorización se permite que cambie la estructura interna de un sistema de software aunque no varíe su comportamiento externo. F
6. La refactorización debe hacerse siempre apoyándonos en un conjunto de tests completo y robusto. V
7. Una clase con un gran número de métodos y atributos en candidata a ser refactorizada. V
8. Los métodos grandes (con muchas instrucciones) son estructuras que sugieren la posibilidad de una refactorización. V
9. Un ejemplo de refactorización sería mover un método arriba o abajo en la jerarquía de herencia. V
10. La refactorización ha sido identificada como una de las importantes innovaciones en el campo del software. V
11. La existencia de una sólida colección de pruebas unitarias es una precondición fundamental para la refactorización.
12. Refactorizaciones comunes:
    - Mover un atributo
    - Mover un método
    - Extraer una clase
    - Extraer un método
    - Cambiar condicionales por polimorfismo.
    - Cambiar código de error por excepciones.
    - Código Duplicado.
    - Las sentencias switch son código sospechoso.

Tema 10. Principios de Diseño Orientado a Objetos

1. El principio abierto-cerrado indica que un componente software debe estar abierto a la extensión y cerrado a la modificación. V
2. El principio de segregación de interfaz indica que el código cliente no debe ser forzado a depender de interfaces que no utiliza.
3. Cuando diseñamos sistemas orientados a objetos las interfaces de las clases que diseñamos deberían estar abiertas a la extensión y cerradas a la modificación. V

Otros

1. En el diseño de tarjetas CRC utilizamos una tarjeta para cada clase. V
2. Una tarjeta CRC contiene el nombre de una clase, su lista de responsabilidades y su lista de colaboradores. V

3. Una colaboración describe como un grupo de objetos trabaja conjuntamente para realizar una tarea. V
4. El recolector de basura es un mecanismo de liberación de recursos presente en todos los lenguajes OO. F
5. La interpretación de un mismo mensaje puede derivar en función del receptor del mensaje de la información adicional que le acompañe. V
6. Hablamos de encapsulación cuando agrupamos datos junto con las operaciones que pueden realizarse sobre esos datos. V
7. El puntero this es un puntero que no puede cambiar de valor y que contiene la dirección del objeto receptor del mensaje, además existe en cualquier método definido dentro de la clase. F