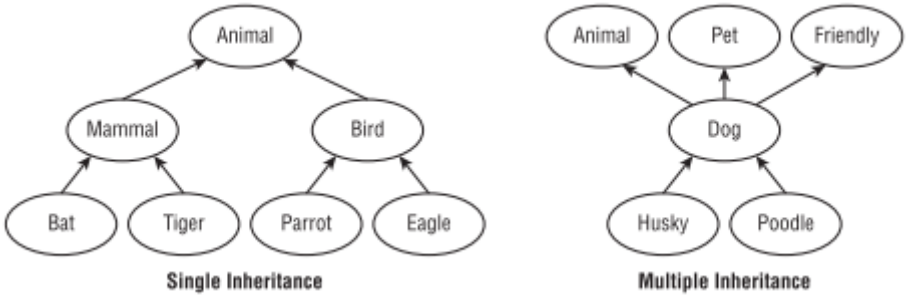
Class Design

1. Introducing Class Inheritance

La herencia es el proceso mediante el cual la nueva subclase hija incluye automáticamente primitivas, objetos o métodos públicos o protegidos definidos en la clase principal.

Java admite la herencia única, por la cual una clase puede heredar de una sola clase principal directa. Java también admite múltiples niveles de herencia, por lo que una clase puede extender otra clase, lo que a su vez amplía otra clase.

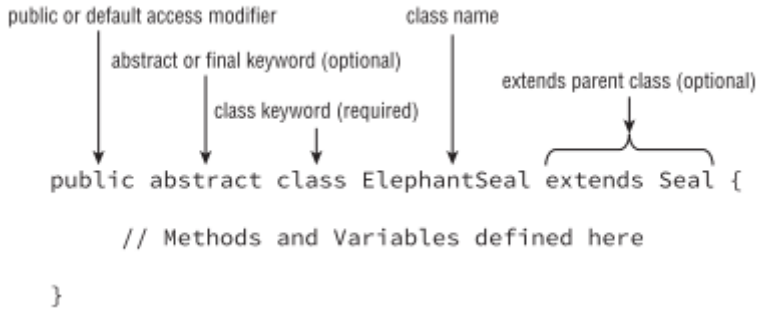
La siguiente figura ilustra los diversos tipos de modelos de herencia. Los elementos de la izquierda se consideran herencia única porque cada hijo tiene exactamente un padre. El lado derecho muestra elementos que tienen herencia múltiple.



Es posible en Java evitar que una clase se extienda marcando la clase con el modificador final.

1.1. Extending a Class

En Java, puede extender una clase agregando el nombre de la clase padre en la definición usando la palabra clave `extends`.



Por ejemplo:

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

Y aquí están los contenidos de Lion.java:

```
public class Lion extends Animal {
```

```
  private void roar() {
    System.out.println("The "+getAge()+" year old lion says: Roar!");
  }
}
```

El siguiente código no compila:

```
public class Lion extends Animal {
  private void roar() {
    System.out.println("The "+age+" year old lion says: Roar!");
    // DOES NOT COMPILE
  }
}
```

A pesar de que `age` de la clase `Animal` es inaccesible, si tenemos una instancia de un objeto `Lion`, todavía existe un valor `age` dentro de la instancia. De esta manera, el objeto `Lion` es en realidad "más grande" que el objeto `Animal` en el sentido de que incluye todas las propiedades del objeto `Animal`.

1.2. Applying Class Access Modifiers

El modificador de acceso `public` aplicado a una clase indica que puede ser referenciado y utilizado en cualquier clase. El modificador privado del paquete predeterminado, que es la falta de cualquier modificador de acceso, indica que a la clase solo se puede acceder mediante una subclase o clase dentro del mismo paquete.

Una característica del uso del modificador privado del paquete predeterminado es que puede definir muchas clases dentro del mismo archivo Java. Por ejemplo:

```
class Rodent {}
public class Groundhog extends Rodent {}
```

Nota:
Para el examen OCA, solo debe estar familiarizado con los modificadores de acceso de clase `public` y `default` a nivel de paquete, porque estos son los únicos que se pueden aplicar a las clases de nivel superior dentro de un archivo Java. Los modificadores `private` y `protected` solo se pueden aplicar a las clases internas, que son clases definidas dentro de otras clases, pero esto está fuera del alcance del examen OCA.

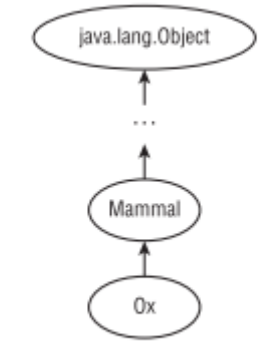
1.3. Creating Java Objects

En Java, todas las clases heredan de una sola clase, `java.lang.Object`. Además, `java.lang.Object` es la única clase que no tiene clases principales.

El compilador ha insertado código automáticamente en cualquier clase que escriba que no extienda una clase específica. Por ejemplo:

```
public class Zoo {
}
public class Zoo extends java.lang.Object {
}
```

La clave es que cuando Java ve que define una clase que no amplía otra clase, inmediatamente agrega la sintaxis extiende `java.lang.Object` a la definición de la clase. Si define una nueva clase que amplíe una clase existente, Java no agrega esta sintaxis, aunque la nueva clase aún hereda de `java.lang.Object`:



All objects inherit java.lang.Object

1.4. Defining Constructors

En Java, la primera declaración de cada constructor es una llamada a otro constructor dentro de la clase, usando `this()`, o una llamada a un constructor en la clase primaria directa, usando `super()`. Por ejemplo:

```
public class Animal {
  private int age;
  public Animal(int age) {
    super();
    this.age = age;
  }
}
public class Zebra extends Animal {
  public Zebra(int age) {
    super(age);
  }
  public Zebra() {
    this(4);
  }
}
```

El comando `super()` solo se puede usar como la primera instrucción del constructor. Por ejemplo, las siguientes dos definiciones de clases no se compilarán:

```
public class Zoo {
  public Zoo() {
    System.out.println("Zoo created");
    super(); // DOES NOT COMPILE
  }
}
public class Zoo {
  public Zoo() {
    super();
    System.out.println("Zoo created");
    super();  // DOES NOT COMPILE
  }
}
```

Si la clase padre tiene más de un constructor, la clase hija puede usar cualquier constructor padre válido en su definición, como se muestra en el siguiente ejemplo:

```
public class Animal {
  private int age;
  private String name;
  public Animal(int age, String name) {
    super();
```

```
    this.age = age;
    this.name = name;
  }
  public Animal(int age) {
    super();
    this.age = age;
    this.name = null;
  }
}
public class Gorilla extends Animal {
  public Gorilla(int age) {
    super(age,"Gorilla");
  }
  public Gorilla() {
    super(5);
  }
}
```

1.4.1. Understanding Compiler Enhancements

El compilador Java inserta automáticamente una llamada al constructor sin argumento `super()` si la primera instrucción no es una llamada al constructor padre. Por ejemplo, el compilador los convertirá automáticamente a la última clase:

```
public class Donkey {
}
public class Donkey {
  public Donkey() {
  }
}
public class Donkey {
  public Donkey() {
    super();
    }
}
```

¿Qué sucede si la clase padre no tiene un constructor sin argumentos? El compilador de Java no ayudará y debe crear al menos un constructor en su clase secundaria que llame explícitamente a un constructor padre a través del comando `super()`. Por ejemplo, el siguiente código no se compilará:

```
public class Mammal {
  public Mammal(int age) {
  }
}
public class Elephant extends Mammal {  // DOES NOT COMPILE
}
```

Debemos definir explícitamente al menos un constructor, como en el siguiente código:

```
public class Mammal {
  public Mammal(int age) {
  }
}
public class Elephant extends Mammal {
  public Elephant() {
    super(10);
  }
}
```

1.4.2. Reviewing Constructor Rules

Reglas de definición de constructor:

1. La primera declaración de cada constructor es una llamada a otro constructor dentro de la clase que usa `this()`, o una llamada a un constructor en la clase primaria directa usando `super()`.
2. La llamada `super()` no se puede usar después de la primera instrucción del constructor.
3. Si no se declara una llamada a `super()` en un constructor, Java insertará un no-argumento `super()` como la primera instrucción del constructor.
4. Si el elemento primario no tiene un constructor sin argumentos y el elemento secundario no define ningún constructor, el compilador generará un error e intentará insertar un constructor predeterminado sin argumentos en la clase secundaria.
5. Si el padre no tiene un constructor sin argumentos, el compilador requiere una llamada explícita a un constructor padre en cada constructor hijo.

1.4.3. Calling Constructors

En Java, el constructor padre siempre se ejecuta antes que el constructor hijo. Por ejemplo, intente determinar qué produce el siguiente código:

```
class Primate {
  public Primate() {
    System.out.println("Primate");
  }
}
class Ape extends Primate {
  public Ape() {
    System.out.println("Ape");
  }
}
public class Chimpanzee extends Ape {
  public static void main(String[] args) {
    new Chimpanzee();
  }
}
```

El compilador primero inserta el comando super () como la primera declaración de los constructores primado y simio. El resultado es:

```
Primate
Ape
```

1.5. Calling Inherited Class Members

Las clases de Java pueden usar cualquier miembro `public` o `protected` de la clase principal, incluidos métodos, primitivas o referencias a objetos. Si la clase principal y la clase hija son parte del mismo paquete, la clase hija también puede usar cualquier miembro predeterminado definido en la clase principal. Finalmente, una clase hija nunca puede acceder a un miembro privado de la clase principal, al menos no a través de ninguna referencia directa. Por ejemplo:

```
class Fish {
  protected int size;
  private int age;

  public Fish(int age) {
    this.age = age;
  }

  public int getAge() {
    return age;
  }
```

```
}
public class Shark extends Fish {
  private int numberOfFins = 8;

  public Shark(int age) {
    super(age);
    this.size = 4;
  }

  public void displaySharkDetails() {
    System.out.print("Shark with age: "+getAge());
    System.out.print(" and "+size+" meters long");
    System.out.print(" with "+numberOfFins+" fins");
  }
}
```

En Java, puede hacer referencia explícitamente a un miembro de la clase padre utilizando la palabra clave super, como en la siguiente definición alternativa de `displayShark Details()`:

```
public void displaySharkDetails() {
  System.out.print("Shark with age: "+super.getAge());
  System.out.print(" and "+super.size+" meters long");
  System.out.print(" with "+super.numberOfFins+" fins"); // DOES NOT COMPILE
                                                        // because numberOfFins is only a
                                                        // member of the current class
}
```

super() vs super
El examen puede intentar engañarte usando tanto super () como super en un constructor. Por ejemplo, considere el siguiente código:

```
public Rabbit(int age) {
  super();
  super.setAge(10);
}
```

La primera instrucción del constructor llama al constructor del padre, mientras que la segunda declara una función definida en la clase padre. Contraste esto con el siguiente código, que no se compila:

```
public Rabbit(int age) {
  super;  // DOES NOT COMPILE
  super().setAge(10);  // DOES NOT COMPILE
}
```

1.6. Inheriting Methods

1.6.1. Overriding a Method

El compilador realiza las siguientes comprobaciones cuando sobrescribe un método no privado:

1. El método en la clase secundaria debe tener la misma firma que el método en la clase principal.
2. El método en la clase secundaria debe ser al menos tan accesible o más accesible que el método en la clase principal.
3. El método en la clase secundaria no puede arrojar una excepción marcada (*checked*) que sea nueva o más amplia que la clase de cualquier excepción lanzada en el método de clase padre.
4. Si el método devuelve un valor, debe ser el mismo o una subclase del método en la clase principal, conocidos como tipos de retorno covariantes.

Repasemos algunos ejemplos:

```
public class Camel {
  protected String getNumberOfHumps() {
```

```
    return "Undefined";
  }
}
public class BactrianCamel extends Camel {
  private int getNumberOfHumps() {  // DOES NOT COMPILE
    return 2;
  }
}
```

En este ejemplo, el método en la clase hija no se compila por dos razones. En primer lugar, viola la segunda regla de sobre escritura de métodos: el método hijo debe ser al menos tan accesible como el padre. También viola la cuarta regla de sobre escritura de métodos: el tipo de devolución del método principal y el método secundario deben ser covariantes.

Echemos un vistazo a algunos ejemplos de métodos que usan excepciones:

```
public class InsufficientDataException extends Exception {}
public class Reptile {
  protected boolean hasLegs() throws InsufficientDataException {
    throw new InsufficientDataException();
  }
  protected double getWeight() throws Exception {
    return 2;
  }
}
public class Snake extends Reptile {
  protected boolean hasLegs() {
    return false;
  }
  protected double getWeight() throws InsufficientDataException{
    return 2;
  }
}
```

Repasemos algunos ejemplos que infringen la tercera regla de sobrescribe de métodos:

```
public class InsufficientDataException extends Exception {}
public class Reptile {
  protected double getHeight() throws InsufficientDataException {
    return 2;
  }
  protected int getLength() {
    return 10;
  }
}public class Snake extends Reptile {
  protected double getHeight() throws Exception {  // DOES NOT COMPILE
    return 2;
  }
  protected int getLength() throws InsufficientDataException { // DOES NOT
COMPILE
    return 10;
  }
}
```

Como `Exception` no es una subclase de `InsufficientDataException`, la tercera regla de sobre escritura de métodos se infringe y el código no se compilará. La clase secundaria define una nueva excepción que no fue la clase principal, que es una violación de la tercera regla de sobre escritura de métodos.

1.6.2. Redeclaring private Methods

En Java, no es posible sobrescribir un método privado en una clase principal, ya que no se puede acceder al método principal desde la clase secundaria. El hecho de que una clase hija no tenga acceso al método padre no significa que la clase hija no pueda definir su propia versión del método. Simplemente significa, estrictamente hablando, que el nuevo método no es una versión sobre escrita del método de la clase padre.

Java le permite volver a declarar un nuevo método en la clase secundaria con la firma igual o modificada que el método en la clase principal. Este método en la clase secundaria es un método separado e independiente, no relacionado con el método de la versión principal, por lo que no se invoca ninguna de las reglas de sobre escritura de métodos. Por ejemplo:

```
public class Camel {
  private String getNumberOfHumps() {
    return "Undefined";
  }
}
public class BactrianCamel extends Camel {
  private int getNumberOfHumps() {
    return 2;
  }
}
```

1.6.3. Hiding Static Methods

Un método oculto se produce cuando una clase secundaria define un método estático con el mismo nombre y firma que un método estático definido en una clase principal. La siguiente lista resume las cinco reglas para ocultar un método:

1. El método en la clase secundaria debe tener la misma firma que el método en la clase principal.
2. El método en la clase secundaria debe ser al menos tan accesible o más accesible que el método en la clase principal.
3. El método en la clase secundaria no puede arrojar una excepción marcada que sea nueva o más amplia que la clase de cualquier excepción lanzada en el método de clase padre.
4. Si el método devuelve un valor, debe ser el mismo o una subclase del método en la clase principal, conocidos como tipos de retorno covariantes.
5. El método definido en la clase secundaria debe marcarse como estático si está marcado como estático en la clase padre (método oculto). Del mismo modo, el método no se debe marcar como estático en la clase secundaria si no se marca como estático en la clase padre (sobre escritura del método).

Tenga en cuenta que los primeros cuatro son los mismos que las reglas para sobrescribir un método. Por ejemplo:

```
public class Bear {
  public static void eat() {
    System.out.println("Bear is eating");
  }
}
public class Panda extends Bear {
  public static void eat() {
    System.out.println("Panda bear is chewing");
  }
  public static void main(String[] args) {
    Panda.eat();
  }
}
```

El método `eat()` en la clase secundaria oculta el método `eat()` en la clase principal. Debido a que ambos están marcados como estáticos, esto no se considera un método sobre escrito. Comparemos esto con ejemplos que violan la quinta regla:

```
public class Bear {
  public static void sneeze() {
    System.out.println("Bear is sneezing");
  }
  public void hibernate() {
    System.out.println("Bear is hibernating");
  }
}
public class Panda extends Bear {
  public void sneeze() {  // DOES NOT COMPILE
    System.out.println("Panda bear sneezes quietly");
  }
  public static void hibernate() {  // DOES NOT COMPILE
    System.out.println("Panda bear is going to sleep");
  }
}
```

En este ejemplo, `sneeze()` se marca como estático en la clase principal pero no en la clase secundaria. El compilador detecta que está intentando sobrescribir un método que debe ocultarse y genera un error de compilación.

1.6.4. Overriding vs. Hiding Methods

A diferencia de reemplazar un método, en el que un método secundario reemplaza el método padre en llamadas definidas tanto en el padre como en el hijo, los métodos ocultos solo reemplazan los métodos principales en las llamadas definidas en la clase hija.

En tiempo de ejecución, la versión secundaria de un método reemplazado siempre se ejecuta para una instancia independientemente de si la llamada al método está definida en un método de clase principal o secundaria. De esta manera, el método padre nunca se usa a menos que se haga referencia a una llamada explícita al método padre, usando la sintaxis `ParentClassName.method()`. Alternativamente, en tiempo de ejecución, la versión principal de un método oculto siempre se ejecuta si la llamada al método se define en la clase padre. Echemos un vistazo a un ejemplo:

```
public class Marsupial {
  public static boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}
public class Kangaroo extends Marsupial {
  public static boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

... y el resultado:

```
Marsupial walks on two legs: false
Kangaroo hops on two legs: true
```

Contraste este primer ejemplo con el siguiente ejemplo, que utiliza una versión sobrescrita de `isBiped()` en lugar de una versión oculta:

```
class Marsupial {
  public boolean isBiped() {
    return false;
  }
  public void getMarsupialDescription() {
    System.out.println("Marsupial walks on two legs: "+isBiped());
  }
}
public class Kangaroo extends Marsupial {
  public boolean isBiped() {
    return true;
  }
  public void getKangarooDescription() {
    System.out.println("Kangaroo hops on two legs: "+isBiped());
  }
  public static void main(String[] args) {
    Kangaroo joey = new Kangaroo();
    joey.getMarsupialDescription();
    joey.getKangarooDescription();
  }
}
```

... pero produce un texto ligeramente diferente:

```
Marsupial walks on two legs: true
Kangaroo hops on two legs: true
```

1.6.5. Creating final methods

Esta regla se aplica tanto cuando se sobrescribe como cuando se oculta un método. En otras palabras, no puede ocultar un método estático en una clase principal si está marcado como final. Echemos un vistazo a un ejemplo:

```
public class Bird {
  public final boolean hasFeathers() {
    return true;
  }
}
public class Penguin extends Bird {
  public final boolean hasFeathers() { // DOES NOT COMPILE
    return false;
  }
}
```

1.7. Inheriting Variables

Java no permite que las variables se anulen sino que se oculten.

1.7.1. Hiding Variables

Cuando oculta una variable, define una variable con el mismo nombre que una variable en una clase principal. Esto crea dos copias de la variable dentro de una instancia de la clase hija: una instancia definida para la referencia principal y otra definida para la referencia secundaria.

Si hace referencia a la variable desde la clase principal, se utiliza la variable definida en la clase principal. Alternativamente, si hace referencia a la variable desde una clase secundaria, se utiliza la variable definida en la clase secundaria. Considere el siguiente ejemplo:

```
public class Rodent {
  protected int tailLength = 4;
  public void getRodentDetails() {
    System.out.println("[parentTail="+tailLength+"]");
  }
}
public class Mouse extends Rodent {
  protected int tailLength = 8;
  public void getMouseDetails() {
    System.out.println("[tail="+tailLength +",parentTail="+super.tailLength+"]");
  }
  public static void main(String[] args) {
    Mouse mouse = new Mouse();
    mouse.getRodentDetails();
    mouse.getMouseDetails();
  }
}
```

...y la salida:

```
[parentTail=4]
[tail=8,parentTail=4]
```

Lo importante para recordar es que no existe la noción de sobre escritura de una variable miembro. Estas reglas son las mismas independientemente de si la variable es una variable de instancia o una variable estática.

2. Creating Abstract Classes

Una clase abstracta es una clase que está marcada con la palabra clave abstracta y no se puede crear una instancia. Un método abstracto es un método marcado con la palabra clave abstracta definida en una clase abstracta, para la cual no se proporciona implementación en la clase en la que se declara. Por ejemplo:

```
public abstract class Animal {
  protected int age;
  public void eat() {
    System.out.println("Animal is eating");
  }
  public abstract String getName();
}
public class Swan extends Animal {
  public String getName() {
    return "Swan";
  }
}
```

Una clase abstracta puede incluir métodos y variables no abstractas, como se vio con la variable edad y el método `eat()`:

1. De hecho, una clase abstracta no está obligada a incluir ningún método abstracto.
2. La clase abstracta no puede ser final.
3. Un método abstracto no se puede marcar como final por la misma razón que una clase abstracta no se puede marcar como final.
4. Finalmente, un método no puede marcarse como abstracto y privado.
5. El compilador reconoce esto en la clase padre y lanza una excepción tan pronto como se aplican el método privado y el abstracto.

Por ejemplo:

```
public abstract class Cow {
}

public class Chicken {
  public abstract void peck();  // DOES NOT COMPILE
}

public final abstract class Tortoise {  // DOES NOT COMPILE
}

public abstract class Goat {
  public abstract final void chew();  // DOES NOT COMPILE
}

public abstract class Whale {
  private abstract void sing();  // DOES NOT COMPILE
}
public class HumpbackWhale extends Whale {
  private void sing() {          // DOES NOT COMPILE
    System.out.println("Humpback whale is singing");
  }
}

public abstract class Whale {
  protected abstract void sing();
}
public class HumpbackWhale extends Whale {
  private void sing() {  // DOES NOT COMPILE
    System.out.println("Humpback whale is singing");
  }
}
```

2.1. Creating a Concrete Class

Al trabajar con clases abstractas, es importante recordar que, por sí solas, no se pueden crear instancias y, por lo tanto, no hacen mucho más que definir variables y métodos estáticos. Por ejemplo:

```
public abstract class Eel {
  public static void main(String[] args) {
    final Eel eel = new Eel();  // DOES NOT COMPILE
  }
}
```

Una clase concreta es la primera subclase no abstracta que extiende una clase abstracta y se requiere para implementar todos los métodos abstractos heredados. Por ejemplo:

```
public abstract class Animal {
  public abstract String getName();
}
public class Walrus extends Animal { // DOES NOT COMPILE
}
```

El punto clave es que la primera clase para extender la clase no abstracta debe implementar todos los métodos abstractos heredados. Por ejemplo:

```
public abstract class Animal {
  public abstract String getName();
```

```
}
public class Bird extends Animal { // DOES NOT COMPILE
}
public class Flamingo extends Bird {
  public String getName() {
    return "Flamingo";
  }
}
```

Aunque una segunda subclase `Flamingo` implementa el método abstracto `getName()`, la primera subclase concreta `Bird` no; por lo tanto, la clase `Bird` no compilará.

2.2. Extending an Abstract Class

Las clases abstractas pueden extender otras clases abstractas y no están obligadas a proporcionar implementaciones para ninguno de los métodos abstractos. Se sigue, entonces, que una clase concreta que extiende una clase abstracta debe implementar todos los métodos abstractos heredados. Por ejemplo:

```
public abstract class Animal {
  public abstract String getName();
}
public abstract class BigCat extends Animal {
  public abstract void roar();
}
public class Lion extends BigCat {
  public String getName() {
    return "Lion";
  }
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

Existe una excepción a la regla para los métodos abstractos y las clases concretas: no se requiere una subclase concreta para proporcionar una implementación de un método abstracto si una clase abstracta intermedia proporciona la implementación.

```
public abstract class Animal {
  public abstract String getName();
}
public abstract class BigCat extends Animal {
  public String getName() {    return "BigCat";
  }
  public abstract void roar();
}
public class Lion extends BigCat {
  public void roar() {
    System.out.println("The Lion lets out a loud ROAR!");
  }
}
```

Las siguientes son listas de reglas para clases abstractas y métodos abstractos que hemos cubierto en esta sección.

Reglas abstractas de definición de clase:

1. Las clases abstractas no se pueden crear instancias directamente.
2. Las clases abstractas se pueden definir con cualquier número, incluyendo cero, de métodos abstractos y no abstractos.

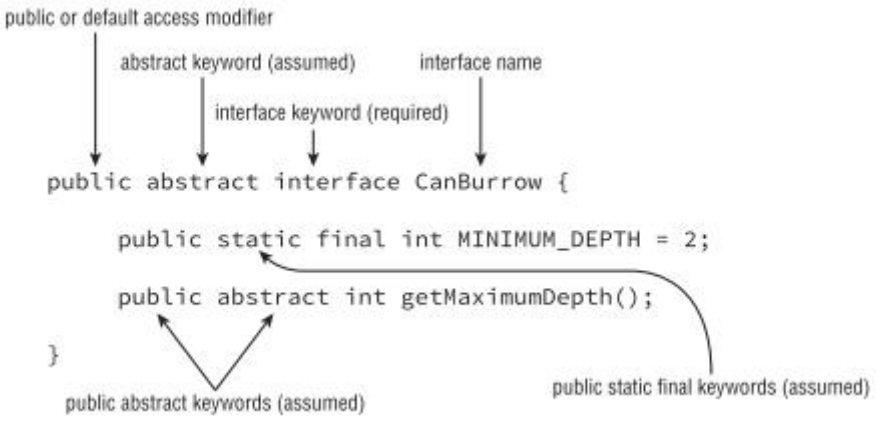
3. Las clases abstractas no pueden marcarse como privadas o finales.
4. Una clase abstracta que amplía otra clase abstracta hereda todos sus métodos abstractos como sus propios métodos abstractos.
5. La primera clase concreta que amplía una clase abstracta debe proporcionar una implementación para todos los métodos abstractos heredados.

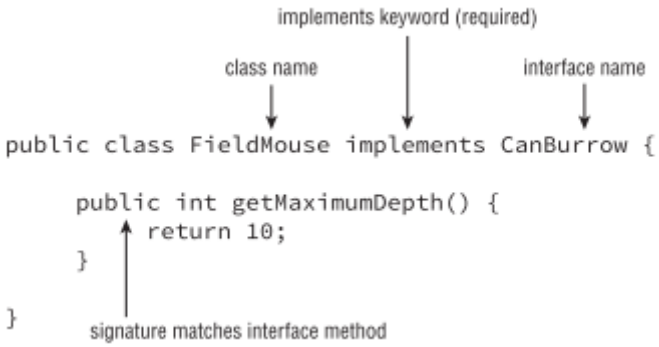
Resumen de reglas de definición de método:

1. Los métodos abstractos solo se pueden definir en clases abstractas.
2. Los métodos abstractos no pueden declararse private o final.
3. Los métodos abstractos no deben proporcionar un cuerpo / implementación de método en la clase abstracta para la cual se declara.
4. La implementación de un método abstracto en una subclase sigue las mismas reglas para sobrescribir un método. Por ejemplo, el nombre y la firma deben ser iguales, y la visibilidad del método en la subclase debe ser al menos tan accesible como el método en la clase principal.

3. Implementing Interfaces

Si bien Java no permite la herencia múltiple, sí permite que las clases implementen cualquier cantidad de interfaces. Una interfaz es un tipo de datos abstractos que define una lista de métodos públicos abstractos que debe proporcionar cualquier clase que implemente la interfaz. Una interfaz también puede incluir una lista de variables constantes y métodos predeterminados.





Una clase puede implementar múltiples interfaces, cada una separada por una coma, como en el siguiente ejemplo:

```
public class Elephant implements WalksOnFourLegs, HasTrunk, Herbivore {
}
```

3.1. Defining an Interface

La siguiente es una lista de reglas para crear una interfaz, muchas de las cuales debe reconocer como adaptaciones de las reglas para definir clases abstractas:

1. Las interfaces no se pueden instanciar directamente.
2. No se requiere una interfaz tener algún método.
3. Una interfaz no puede ser marcada como final.
4. Se supone que todas las interfaces de nivel superior tienen acceso público o predeterminado, y deben incluir el modificador abstracto en su definición. Por lo tanto, marcar una interfaz como privada, protegida o final desencadenará un error de compilación, ya que esto es incompatible con estas suposiciones.
5. Se supone que todos los métodos no predeterminados en una interfaz tienen los modificadores abstractos y públicos en su definición. Por lo tanto, marcar un método como privado, protegido o final desencadenará errores de compilación ya que son incompatibles con las palabras clave abstractas y públicas.

La cuarta regla no se aplica a las interfaces internas, aunque las clases internas y las interfaces no están dentro del alcance del examen OCA.

Por ejemplo, las siguientes dos definiciones de interfaz son equivalentes, ya que el compilador las convertirá a las dos en el segundo ejemplo:

```
public interface CanFly {
  void fly(int speed);
  abstract void takeoff();
  public abstract double dive();
}
public abstract interface CanFly {
  public abstract void fly(int speed);
  public abstract void takeoff();
  public abstract double dive();
}
```

Echemos un vistazo a un ejemplo que infringe las palabras clave supuestas:

```
private final interface CanCrawl {  // DOES NOT COMPILE
  private void dig(int depth);  // DOES NOT COMPILE
  protected abstract double depth();  // DOES NOT COMPILE
  public final void surface();  // DOES NOT COMPILE
}
```

3.2. Inheriting an Interface

Hay dos reglas de herencia que debe tener en cuenta al extender una interfaz:

1. Una interfaz que amplía otra interfaz, así como una clase abstracta que implementa una interfaz, hereda todos los métodos abstractos como sus propios métodos abstractos.
2. La primera clase concreta que implementa una interfaz, o extiende una clase abstracta que implementa una interfaz, debe proporcionar una implementación para todos los métodos abstractos heredados.

Por ejemplo:

```
public interface HasTail {
  public int getTailLength();
}
public interface HasWhiskers {
  public int getNumberOfWhiskers();
}
public abstract class HarborSeal implements HasTail, HasWhiskers {
```

```
}
public class LeopardSeal implements HasTail, HasWhiskers {  // DOES NOT COMPILE
}
```

Se requerirá que cualquier clase que extienda `HarborSeal` implemente todos los métodos en la interfaz `HasTail` y `HasWhiskers`.

3.2.1. Classes, Interfaces, and Keywords

A los creadores de los exámenes les gustan las preguntas que mezclan terminología de clase e interfaz. Aunque una clase puede implementar una interfaz, una clase no puede extender una interfaz. Del mismo modo, mientras que una interfaz puede extender otra interfaz, una interfaz no puede implementar otra. Los siguientes ejemplos ilustran estos principios:

```
public interface CanRun {}
public class Cheetah extends CanRun {}  // DOES NOT COMPILE
public class Hyena {}
public interface HasFur extends Hyena {} // DOES NOT COMPILE
```

3.2.2. Abstract Methods and Multiple Inheritance

Como Java permite la herencia múltiple a través de interfaces, es posible que se pregunte qué sucederá si define una clase que hereda de dos interfaces que contienen el mismo método abstracto:

```
public interface Herbivore {
  public void eatPlants();
}
public interface Omnivore {
  public void eatPlants();
  public void eatMeat();
}
```

Las firmas para los dos métodos de interfaz `eatPlants()` son compatibles, por lo que puede definir una clase que satisfaga ambas interfaces simultáneamente:

```
public class Bear implements Herbivore, Omnivore {
  public void eatMeat() {
    System.out.println("Eating meat");
  }
  public void eatPlants() {
    System.out.println("Eating plants");
  }
}
```

Desafortunadamente, si el nombre del método y los parámetros de entrada son los mismos, pero los tipos de devolución son diferentes entre los dos métodos, la clase o interfaz que intenta heredar ambas interfaces no se compilará. No es posible en Java definir dos métodos en una clase con el mismo nombre y los mismos parámetros de entrada, pero diferentes tipos de devolución. Por ejemplo:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
public class Bear implements Herbivore, Omnivore {
  public int eatPlants() {  // DOES NOT COMPILE
```

```
    System.out.println("Eating plants: 10");
    return 10;
  }
  public void eatPlants() {  // DOES NOT COMPILE
    System.out.println("Eating plants");
  }
}
```

El compilador también lanzaría una excepción si define una interfaz o clase abstracta que hereda de dos interfaces en conflicto, como se muestra aquí:

```
public interface Herbivore {
  public int eatPlants();
}
public interface Omnivore {
  public void eatPlants();
}
public interface Supervore extends Herbivore, Omnivore {} // DOES NOT COMPILE
public abstract class AbstractBear implements Herbivore, Omnivore {}
                                                  // DOES NOT COMPILE
```

3.3. Interface Variables

Aquí hay dos reglas de variables de interfaz:

1. Se supone que las variables de interfaz son públicas, estáticas y finales. Por lo tanto, marcar una variable como privada o protegida desencadenará un error de compilación, al igual que marcar cualquier variable como abstracta.
2. El valor de una variable de interfaz debe establecerse cuando se declara, ya que está marcado como final.

De esta manera, las variables de interfaz son esencialmente variables constantes definidas en el nivel de interfaz. Por ejemplo:

```
public interface CanSwim {
  int MAXIMUM_DEPTH = 100;
  final static boolean UNDERWATER = true;
  public static final String TYPE = "Submersible";
}
public interface CanSwim {
  public static final int MAXIMUM_DEPTH = 100;
  public static final boolean UNDERWATER = true;
  public static final String TYPE = "Submersible";
}
```

Según estas reglas, no debería sorprender que las siguientes entradas no se compilarán:

```
public interface CanDig {
  private int MAXIMUM_DEPTH = 100;  // DOES NOT COMPILE
  protected abstract boolean UNDERWATER = false;  // DOES NOT COMPILE
  public static String TYPE;  // DOES NOT COMPILE
}
```

3.4. Default Interface Methods

Un método predeterminado es un método definido dentro de una interfaz con la palabra clave `default` en la que se proporciona un cuerpo de método. Un método predeterminado dentro de una interfaz define un método abstracto con una implementación predeterminada. De esta manera, las clases tienen la opción de

sobrescribir el método predeterminado si es necesario, pero no están obligados a hacerlo. Si la clase no sobrescribe el método, se usará la implementación predeterminada. De esta manera, la definición del método es concreta, no abstracta.

```
public interface IsWarmBlooded {
  boolean hasScales();
  public default double getTemperature() {
    return 10.0;
  }
}
```

Las siguientes son las reglas del método de interfaz predeterminado con las que necesita familiarizarse:

1. Un método predeterminado solo se puede declarar dentro de una interfaz y no dentro de una clase o clase abstracta.
2. Un método predeterminado debe marcarse con la palabra clave default. Si un método está marcado como predeterminado, debe proporcionar un cuerpo de método.
3. No se supone que un método predeterminado sea estático, final o abstracto, ya que puede ser implementado o sobrescrito por una clase que implementa la interfaz.
4. Como todos los métodos en una interfaz, se supone que un método predeterminado es público y no se compilará si se marca como privado o protegido.

Cuando una interfaz amplía otra interfaz que contiene un método predeterminado:

1. Puede optar por ignorar el método predeterminado, en cuyo caso se utilizará la implementación predeterminada para el método.
2. Alternativamente, la interfaz puede sobre escribir la definición del método predeterminado utilizando las reglas estándar de sobre escritura de métodos, como no limitar el acceso al método y usar retornos covariantes.
3. Finalmente, la interfaz puede re declarar el método como abstracto, requiriendo clases que implementen la nueva interfaz para proporcionar explícitamente un cuerpo de método.
4. Se aplican opciones análogas para una clase abstracta que implementa una interfaz.

Por ejemplo:

```
public interface HasFins {
  public default int getNumberOfFins() {
    return 4;
  }
  public default double getLongestFinLength() {
    return 20.0;
  }
  public default boolean doFinsHaveScales() {
    return true;
  }
}public interface SharkFamily extends HasFins {
  public default int getNumberOfFins() {
    return 8;
  }
  public double getLongestFinLength();
  public boolean doFinsHaveScales() {  // DOES NOT COMPILE
    return false;
  }
}
```

3.4.1. Default Methods and Multiple Inheritance

Demos una revisada al siguiente ejemplo:

```
public interface Walk {
```

```
  public default int getSpeed() {
    return 5;
  }
}
public interface Run {
  public default int getSpeed() {
    return 10;
  }
}
public class Cat implements Walk, Run {  // DOES NOT COMPILE
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

Si una clase implementa dos interfaces que tienen métodos predeterminados con el mismo nombre y firma, el compilador emitirá un error. Sin embargo, existe una excepción a esta regla: si la subclase sobrescribe los métodos por defecto duplicados, el código compilará sin problemas la ambigüedad sobre qué versión del método para llamar ha sido eliminada. Por ejemplo:

```
public class Cat implements Walk, Run {
  public int getSpeed() {
    return 1;
  }
  public static void main(String[] args) {
    System.out.println(new Cat().getSpeed());
  }
}
```

3.5. Static Interface Methods

Estos métodos se definen explícitamente con la palabra clave estática y su función es casi idéntica a los métodos estáticos definidos en las clases. De hecho, en realidad solo hay una distinción entre un método estático en una clase y una interfaz. Un método estático definido en una interfaz no se hereda en ninguna clase que implemente la interfaz. Estas son las reglas del método de interfaz estático con las que debe familiarizarse:

1. Como todos los métodos en una interfaz, se supone que un método estático es público y no se compilará si se marca como privado o protegido.
2. Para hacer referencia al método estático, se debe usar una referencia al nombre de la interfaz.

El siguiente es un ejemplo de un método estático definido en una interfaz:

```
public interface Hop {
  static int getJumpHeight() {
    return 8;
  }
}
```

Una clase `Bunny` que implementa Hop:

```
public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(getJumpHeight()); // DOES NOT COMPILE
  }
}
```

La siguiente versión modificada del código resuelve el problema:

```
public class Bunny implements Hop {
  public void printDetails() {
    System.out.println(Hop.getJumpHeight());
```

```
  }
}
```

De ello se desprende que una clase que implemente dos interfaces que contengan métodos estáticos con la misma firma se compilará en tiempo de ejecución, ya que los métodos estáticos no son heredados por la subclase y deben accederse con una referencia al nombre de la interfaz.

4. Understanding Polymorphism

Java admite el polimorfismo, la propiedad de un objeto para tomar muchas formas diferentes. Para poner esto más precisamente, se puede acceder a un objeto Java usando una referencia del mismo tipo que el objeto, una referencia que es una superclase del objeto, o una referencia que define una interfaz que el objeto implementa, ya sea directamente o a través de una superclase. Por ejemplo:

```
public class Primate {
  public boolean hasHair() {
    return true;
  }
}
public interface HasTail {
  public boolean isTailStriped();
}
public class Lemur extends Primate implements HasTail {
  public boolean isTailStriped() {
    return false;
  }
  public int age = 10;
  public static void main(String[] args) {
    Lemur lemur = new Lemur();
    System.out.println(lemur.age);
    HasTail hasTail = lemur;
    System.out.println(hasTail.isTailStriped());
    Primate primate = lemur;
    System.out.println(primate.hasHair());
    }
}
```

Este código se compila y ejecuta sin problemas y produce el siguiente resultado:

```
10
false
true
```

Los siguientes fragmentos de código no se compilarán:

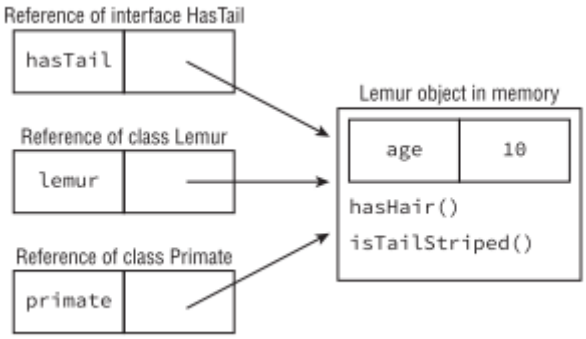
```
HasTail hasTail = lemur;
System.out.println(hasTail.age);  // DOES NOT COMPILE
Primate primate = lemur;
System.out.println(primate.isTailStriped());  // DOES NOT COMPILE
```

4.1. Object vs. Reference

En Java, se accede a todos los objetos por referencia, de modo que como desarrollador nunca se tiene acceso directo al objeto en sí. Tomemos en cuenta las siguientes reglas:

1. El tipo del objeto determina qué propiedades existen dentro del objeto en la memoria.
2. El tipo de referencia al objeto determina qué métodos y variables son accesibles para el programa Java.

Vamos a ilustrar esta propiedad usando el ejemplo anterior en la siguiente figura:



4.2. Casting Objects

Podemos reclamar esas referencias devolviendo el objeto a la subclase específica de la que proviene:

```
Primate primate = lemur;
Lemur lemur2 = primate; // DOES NOT COMPILE
Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

Aquí hay algunas reglas básicas a tener en cuenta cuando se lanzan variables:

1. Lanzar un objeto de una subclase a una superclase no requiere un casteo explícito.
2. Lanzar un objeto de una superclase a una subclase requiere un casteo explícito.
3. El compilador no permitirá conversiones a tipos no relacionados.
4. Incluso cuando el código se compila sin problemas, se puede lanzar una excepción en el tiempo de ejecución si el objeto que se está emitiendo no es realmente una instancia de esa clase.

Considera este ejemplo:

```
public class Bird {}
public class Fish {
  public static void main(String[] args) {
    Fish fish = new Fish();
    Bird bird = (Bird)fish;  // DOES NOT COMPILE
  }
}
```

Casting no está sin sus limitaciones. A pesar de que dos clases comparten una jerarquía relacionada, eso no significa que una instancia de una se puede convertir automáticamente a otra. Aquí hay un ejemplo:

```
public class Rodent {
}
public class Capybara extends Rodent {
  public static void main(String[] args) {
    Rodent rodent = new Rodent();
    Capybara capybara = (Capybara)rodent; // Throws ClassCastException at runtime
  }
}
```

Lanzará una `ClassCastException` en tiempo de ejecución ya que el objeto al que se hace referencia no es una instancia de la clase `Capybara`.

Nota:
Aunque este tema está fuera del alcance del examen OCA, tenga en cuenta que el operador `instanceof` se puede usar para comprobar si un objeto pertenece a una clase en particular y para evitar `ClassCastExceptions` en tiempo de ejecución:

```
if(rodent instanceof Capybara) {
  Capybara capybara = (Capybara)rodent;
}
```

4.3. Virtual Methods

Un método virtual es un método en el que la implementación específica no se determina hasta el tiempo de ejecución. De hecho, todos los métodos Java no finales, no estáticos y no privados se consideran métodos virtuales, ya que cualquiera de ellos puede sobrescribirse en tiempo de ejecución. Por ejemplo:

```
public class Bird {
  public String getName() {
    return "Unknown";
  }
  public void displayInformation() {
    System.out.println("The bird name is: "+getName());
  }
}
public class Peacock extends Bird {
  public String getName() {
    return "Peacock";
  }
  public static void main(String[] args) {
    Bird bird = new Peacock();
    bird.displayInformation();
  }
}
```

En otras palabras, aunque la clase padre `Bird` defina su propia versión de `getName()` y no sepa nada sobre la clase `Peacock` durante el tiempo de compilación, en el tiempo de ejecución la instancia utiliza la versión sobrescrita del método, como se define en la instancia del objeto.

La naturaleza del polimorfismo es que un objeto puede tomar muchas formas diferentes. Al combinar su comprensión del polimorfismo con sobre escritura de métodos, verá que los objetos pueden interpretarse de maneras muy diferentes en el tiempo de ejecución.

4.4. Polymorphic Parameters

Una de las aplicaciones más útiles del polimorfismo es la capacidad de pasar instancias de una subclase o interfaz a un método. Por ejemplo:

```
public class Reptile {
  public String getName() {
    return "Reptile";
  }
}
public class Alligator extends Reptile {
  public String getName() {
    return "Alligator";
  }
}
public class Crocodile extends Reptile {
  public String getName() {
    return "Crocodile";
```

```
  }
}
public class ZooWorker {
  public static void feed(Reptile reptile) {
    System.out.println("Feeding reptile "+reptile.getName());
  }
  public static void main(String[] args) {
    feed(new Alligator());
    feed(new Crocodile());
    feed(new Reptile());
  }
}
```

Este código se compila y ejecuta sin problemas, produciendo el siguiente resultado:

```
Feeding: Alligator
Feeding: Crocodile
Feeding: Reptile
```

4.5. Polymorphism and Method Overriding

Concluyamos este capítulo volviendo a las últimas tres reglas de sobre escritura de métodos para demostrar cómo el polimorfismo requiere que se incluyan como parte de la especificación de Java. Por ejemplo:

```
public class Animal {
  public String getName() {
    return "Animal";
  }
}
public class Gorilla extends Animal {
  protected String getName() {  // DOES NOT COMPILE
    return "Gorilla";
  }
}
public class ZooKeeper {
  public static void main(String[] args) {
    Animal animal = new Gorilla();
    System.out.println(animal.getName());
  }
}
```

La referencia `animal.getName()` está permitida porque el método es público en la clase Animal, pero debido al polimorfismo, el objeto `Gorilla` se ha sobrescrito con una versión menos accesible, no disponible para la clase `ZooKeeper`. Esto crea una contradicción en el sentido de que el compilador no debe permitir el acceso a este método, pero como está referenciado como una instancia de Animal, está permitido. Por lo tanto, Java elimina esta contradicción, impidiendo que un método sea sobrescrito por una versión menos accesible del método.

Del mismo modo, una subclase no puede declarar un método reemplazado con una excepción nueva o más amplia que en la superclase, ya que se puede acceder al método utilizando una referencia a la superclase. Por lo tanto, el compilador de Java no permite los métodos sobrescritos con excepciones nuevas o más amplias.

Finalmente, los métodos reemplazados deben usar tipos de retorno covariantes por las mismas razones que acabamos de mencionar. Si un objeto se convierte en una referencia de superclase y se invoca el método reemplazado, el tipo de retorno debe ser compatible con el tipo de retorno del método principal. Si el tipo de devolución en el niño es demasiado amplio, dará como resultado una excepción de lanzamiento inherente cuando se acceda a través de la referencia de la superclase.

5. Review Questions

5.1.

What is the output of the following code?

```
1: class Mammal {
2:   public Mammal(int age) {
3:     System.out.print("Mammal");
4:   }
5: }
6: public class Platypus extends Mammal {
7:   public Platypus() {
8:     System.out.print("Platypus");
9:   }
10:  public static void main(String[] args) {
11:    new Mammal(5);
12:  }
13: }
```

A. `Platypus`
B. `Mammal`
C. `PlatypusMammal`
D. `MammalPlatypus`
E. The code will not compile because of line 8.
F. The code will not compile because of line 11.

5.2.

Choose the correct statement about the following code:

```
1: interface HasExoskeleton {
2:   abstract int getNumberOfSections();
3: }
4: abstract class Insect implements HasExoskeleton {
5:   abstract int getNumberOfLegs();
6: }
7: public class Beetle extends Insect {
8:   int getNumberOfLegs() { return 6; }
9: }
```

A. It compiles and runs without issue.
B. The code will not compile because of line 2.
C. The code will not compile because of line 4.
D. The code will not compile because of line 7.
E. It compiles but throws an exception at runtime.

5.3.

Choose the correct statement about the following code:

```
1: public interface CanFly {
2:   void fly();
3: }
4: interface HasWings {
5:   public abstract Object getWindSpan();
6: }
7: abstract class Falcon implements CanFly, HasWings {
8: }
```

A. It compiles without issue.

B. The code will not compile because of line 2.
C. The code will not compile because of line 4.
D. The code will not compile because of line 5.
E. The code will not compile because of lines 2 and 5.
F. The code will not compile because the class Falcon doesn't implement the interface methods.

5.4.

What is the output of the following code?

```
1: interface Nocturnal {
2:   default boolean isBlind() { return true; }
3: }
4: public class Owl implements Nocturnal {
5:   public boolean isBlind() { return false; }
6:     public static void main(String[] args) {
7:     Nocturnal nocturnal = (Nocturnal)new Owl();
8:     System.out.println(nocturnal.isBlind());
9:     }
10: }
```

A. `true`
B. `false`
C. The code will not compile because of line 2.
D. The code will not compile because of line 5.
E. The code will not compile because of line 7.
F. The code will not compile because of line 8.

5.5.

What is the output of the following code?

```
1: class Arthropod
2:   public void printName(double input) { System.out.print("Arthropod"); }
3: }
4: public class Spider extends Arthropod {
5:   public void printName(int input) { System.out.print("Spider"); }
6:   public static void main(String[] args) {
7:     Spider spider = new Spider();
8:     spider.printName(4);
9:     spider.printName(9.0);
10:   }
11: }
```

A. `SpiderArthropod`
B. `ArthropodSpider`
C. `SpiderSpider`
D. `ArthropodArthropod`
E. The code will not compile because of line 5.
F. The code will not compile because of line 9.

5.6.

Which statements are true about the following code? (Choose all that apply)

```
1: interface HasVocalCords {
2:   public abstract void makeSound();
3: }
4: public interface CanBark extends HasVocalCords {
```

```
5:   public void bark();
6: }
```

A. The `CanBark` interface doesn't compile.
B. A class that implements `HasVocalCords` must override the `makeSound()` method.
C. A class that implements `CanBark` inherits both the `makeSound()` and `bark()` methods.
D. A class that implements `CanBark` only inherits the `bark()` method.
E. An interface cannot extend another interface.

5.7.

What is the output of the following code? (Choose all that apply)

```
1: interface Aquatic {
2:   public default int getNumberOfGills(int input) { return 2; }
3: }
4: public class ClownFish implements Aquatic {
5:   public String getNumberOfGills() { return "4"; }
6:   public String getNumberOfGills(int input) { return "6"; }
7:   public static void main(String[] args) {
8:     System.out.println(new ClownFish().getNumberOfGills(-1));
9:   }
10: }
```

A. `2`
B. `4`
C. `6`
D. The code will not compile because of line 5.
E. The code will not compile because of line 6.
F. The code will not compile because of line 8.