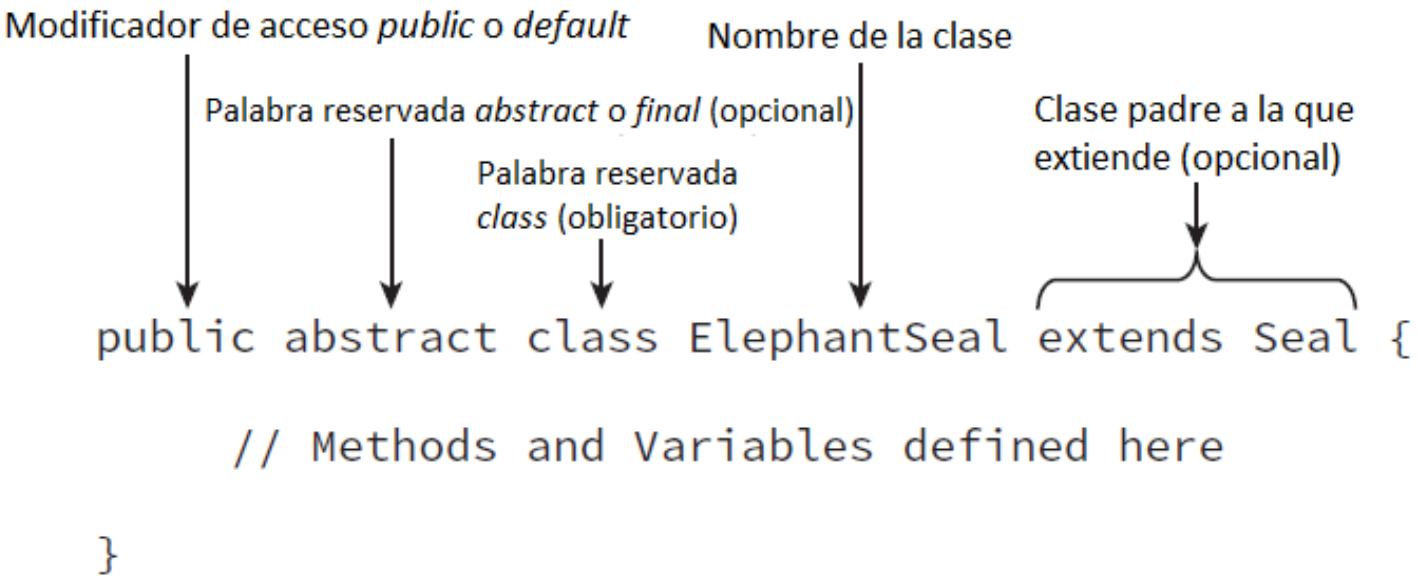
Diseño de clases

Introducción a la herencia de clases



Diseño de clases

Introducción a la herencia de clases

```
public class Animal{
	private int edad;
	public int getEdad(){ return edad; }
	public void setEdad(int Edad){ this.edad = edad; }
}

public class Leon extends Animal{
	private void rugido(){
		System.out.println(“El leon, de ” + getEdad() + “ años de edad dice: Roar!” );
	}
}
```

Diseño de clases

Introducción a la herencia de clases

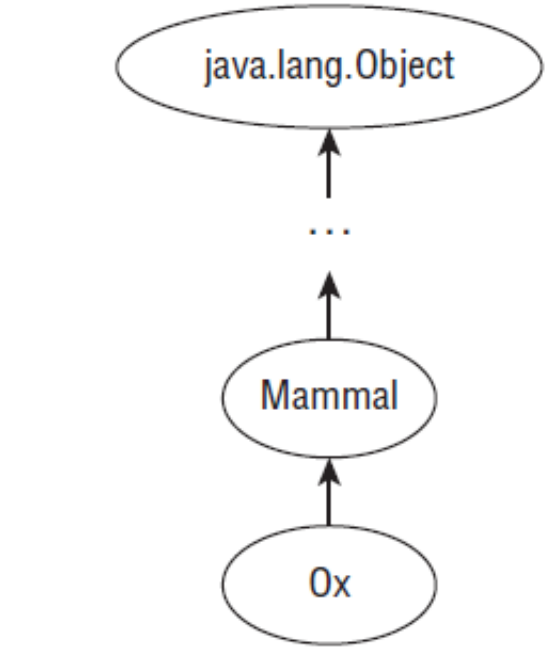
```
class Rodent{}
public class Marmota extends Rodent {}

...

public class Zoo{ }
public class Zoo extends java.lang.Object { }          //Añadido por compilador
```

Diseño de clases

Introducción a la herencia de clases



Todos los objetos heredan de java.lang.Object

Diseño de clases

Definiendo constructores

```
public class Animal{
	private int edad;
	public Animal(int edad){
		super();
		this.edad = edad;
	}
}

...

public class Zebra extends Animal{
	public Zebra(int edad){
		super(edad);
	}
	public Zebra(){ this(4); }
}
```

Diseño de clases

Definiendo constructores

```
public class Zoo {
	public Zoo() {
		System.out.println("Zoo creado");
		super();                                        //NO COMPILA
	}
}

public class Zoo {
	public Zoo() {
		super();
		System.out.println("Zoo creado");
		super();                                        //NO COMPILA
	}
}
```

Diseño de clases

Definiendo constructores

```
public class Burro { }

public class Burro {
	public Burro() {
	}
}

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

Diseño de clases

Definiendo constructores

```
public class Mamifero {
	public Mamifero(int edad) { }
}

public class Elefante extends Mamifero {                    //NO COMPILA
}
```

Diseño de clases

Definiendo constructores

```
public class Mamifero {
	public Mamifero(int edad) { }
}

...

public class Elefante extends Mamifero {                    //NO COMPILA
}

public class Elefante extends Mamifero {
	public Elefante() {
		super(10);
	}
}
```

Diseño de clases

Definiendo constructores - Resumen

1. La primera sentencia de cualquier constructor debe ser una llamada a otro constructor dentro de la clase utilizando this(), o una llamada al constructor de su predecesor directo con el comando super();

2. La llamada super() no puede ser utilizada después de la primera sentencia del constructor;

3. Si no hay una llamada super() declarada en el constructor, Java inserta un super() vacío como primera sentencia del constructor.

4. Si la clase padre no tienen un constructor vacío y el hijo no define ningún constructor, el compilador arroja un error e intenta insertar un constructor vacío por defecto en la clase hijo.

5. Si la clase padre no tiene un constructor sin argumentos, el compilador requiere de una llamada explícita a un constructor de la clase padre en cada constructor de una clase hijo.

Diseño de clases

Definiendo constructores

```
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();
	}
}
```

Resultado

Diseño de clases

Definiendo constructores

```
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();
	}
}
```

Resultado Primate Ape

Diseño de clases

Definiendo constructores

```
class Pez {
	protected int size;
	private int edad;
	public Pez(int edad) {
		this.edad= edad;
	}
	public int getEdad() {
		return edad;
	}
}
public class Tiburon extends Pez {
	private int numeroDeDientes = 8;
	public Tiburon(int edad) {
		super(edad);
		this.size = 4;
	}
	public void displayTiburonDetalles() {
		System.out.print("Tiburon con edad: " + getEdad());
		System.out.print(" y " + size + " metros de largo");
		System.out.print(" con " + numeroDeDientes + " dientes");
	}
}
```

Definiendo constructores

Diseño de clases

```
public class Tiburon extends Pez {

	public void displayTiburonDetalles() {
		System.out.print("Tiburon con edad: " + getEdad());
		System.out.print(" y " + size +" metros de largo");
		System.out.print(" con " + numeroDeDientes +" dientes");
	}
	...
	public void displayTiburonDetalles() {
		System.out.print("Tiburon con edad: " + this.getEdad());
		System.out.print(" y " + this.size +" metros de largo");
		System.out.print(" con " + this.numeroDeDientes + " dientes");
	}
	...
	public void displayTiburonDetalles() {
		System.out.print("Tiburon con edad: " + super.getEdad());
		System.out.print(" y " + super.size + " metros de largo");
		System.out.print(" con " + this.numeroDeDientes + " dientes");
	}
	...
	public void displayTiburonDetalles() {
		System.out.print("Tiburon con edad: " + super.getEdad());
		System.out.print(" y " + super.size + " metros de largo ");
		System.out.print(" con " + super.numeroDeDientes + " dientes");		//NO COMPILA
	}
}
```

Diseño de clases

Definiendo constructores

```
public Conejo(int edad) {
        super();
        super.setEdad(10);
}

public Conejo(int edad) {
        super;                                          //NO COMPILA
        super().setEdad(10);                            //NO COMPILA
}
```

Diseño de clases

Herencia - Sobrescritura

```
public class Can {
	public double getPesoMedio() {
		return 50;
	}
}

public class Lobo extends Can {
	public double getPesoMedio() {
		return super. getPesoMedio()+20;
	}
	public static void main(String[] args) {
		System.out.println(new Can().getPesoMedio ());
		System.out.println(new Lobo().getPesoMedio ());
	}
}
```

Resultado 50.00 70.00

Diseño de clases

Herencia - Sobrescritura

```
public class Lobo extends Can {
	public double getPesoMedio() {
		return super. getPesoMedio()+20;
	}
}

ublic class Lobo extends Can {
	public double getPesoMedio() {
		return getPesoMedio()+20;                                    // Bucle infinito
	}
}
```

Diseño de clases

Herencia - Sobrescritura

Comprobaciones cuando se intenta sobrescribir un método que no es privado:

1. El método en la clase hijo debe tener la misma cabecera que el método en la clase Predecesora;
2. El método en la clase hijo debe ser tan accesible o más que el método en la clase Predecesora;
3. El método en la clase hijo no debe arrojar excepciones obligatorias que sean nuevas o más amplias que las que lance el método de la clase padre;
4. Si el método devuelve un valor, este debe ser del mismo tipo en ambas clases, o una subclase del método en la clase predecesora, conocido como covariant return types.

Diseño de clases

Herencia – Sobrescritura vs. Sobrecarga

```
public class Pajaro {
	public void volar() {
		System.out.println("El pajaro esta volando");
	}
	public void comer(int comida) {
		System.out.println("El pajaro esta comiendo "+comida+" comida ");
	}
}

public class Aguila extends Pajaro {
	public int volar(int altura) {
		System.out.println("El pajaro esta volando a"+ altura +" metros");
		return altura;
	}
	public int comer(int comida) {                                  // DOES NOT COMPILE
		System.out.println("El pajaro esta comiendo "+ comida+ " comida");
		return comida;
	}
}
```

Diseño de clases

Herencia – Sobrescritura vs. Sobrecarga

```
public class Camello {
	protected String getNumeroDeJorobas() {
		return "Indefinido";
	}
}

public class BactrianCamello extends Camello {
	private int getNumeroDeJorobas () {                    //NO COMPILA
		return 2;
	}
}
```

Diseño de clases

Herencia – Sobrescritura vs. Sobrecarga

```
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;
	}
}
```

Diseño de clases

Herencia – Sobrescritura vs. Sobrecarga

```
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;
	}
}
```

Diseño de clases

Herencia – Redeclarando

```
public class Camello {
	private String getNumeroDeJorobas() {
		return "Indefinido";
	}
}

public class BactrianCamel extends Camello {
	private int getNumeroDeJorobas() {
		return 2;
	}
}
```

Diseño de clases

Herencia – Ocultación

5 reglas para ocultar un método:

1. El método en la clase hijo debe tener la misma definición estructural que el método en la clase padre.

2. El método en la clase hijo debe ser al menos tan accesible o más accesible que el método en la clase padre.

3. El método en la clase hijo no puede lanzar una excepción que sea nueva o mejor dicho que la clase de cualquier excepción debe estar lanzada en el método de la clase padre.

4. Si el método devuelve un valor, debe ser el mismo o una subclase del método de la clase padre, conocida como tipos de retorno covariante.

5. El método definido en la clase hijo debe marcarse como estático si está marcado como estático en la clase padre (método oculto). Del mismo modo, el método no debe marcarse como estático en la clase hijo si no está marcado como estático en la clase padre (método primordial).

Diseño de clases

Herencia – Ocultación

```
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();
	}
}
```

Diseño de clases

Herencia – Ocultación

```
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");
	}
}
```

Diseño de clases

Herencia – Sobrescritura vs. Ocultación

```
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();
	}
}
```

Resultado

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

Diseño de clases

Herencia – Sobrescritura vs. Ocultación

```
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();
	}
}
```

Resultado

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

Diseño de clases

Herencia – Métodos final

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

Diseño de clases

Herencia – Variables

```
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();
	}
}
```

Resultado

Diseño de clases

Herencia – Ocultando Variables

```
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();
	}
}
```

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

Diseño de clases

Herencia – No ocultando Variables

```
public class Animal {
	public int length = 2;
}

public class Jellyfish extends Animal {
	public int length = 5;
	public static void main(String[] args) {
		Jellyfish jellyfish = new Jellyfish();
		Animal animal = new Jellyfish();
		System.out.println(jellyfish.length);
		System.out.println(animal.length);
	}
}
```

Resultado

Diseño de clases

Herencia – No ocultando Variables

```
public class Animal {
	public int length = 2;
}

public class Jellyfish extends Animal {
	public int length = 5;
	public static void main(String[] args) {
		Jellyfish jellyfish = new Jellyfish();
		Animal animal = new Jellyfish();
		System.out.println(jellyfish.length);
		System.out.println(animal.length);
	}
}
```

Resultado 5
2

Diseño de clases

Clases abstractas

```
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";
	}
}
```

Diseño de clases

Clases Abstractas – Definición

```
public abstract class Cow { }

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

public abstract class Turtle {
        public abstract void swim() {}                                  // DOES NOT COMPILE
        public abstract int getAge() {                                  // DOES NOT COMPILE
                return 10;
        }
}

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

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

Diseño de clases

Clases Abstractas – Definición

```
public abstract class Whale {
	private abstract void sing();                                   // DOES NOT COMPILE
}
public class HumpbackWhale extends Whale {
	private void sing() {
		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");
	}
}
```

Diseño de clases

Clases Abstractas - Concretando

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

...

public class Walrus extends Animal {                    // DOES NOT COMPILE
}

public class Flamingo extends Bird {
	public String getName() {
		return "Flamingo";
	}
}
```

Diseño de clases

Clases Abstractas – Expandiendo la jerarquía

```
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!");
	}
}
```

Diseño de clases

Clases Abstractas – Expandiendo la jerarquía

```
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!");
	}
}
```

Diseño de clases

Clases Abstractas – Reglas

1. Las clases abstractas no pueden ser instanciadas directamente.
2. Las clases abstractas deben ser definidas con cualquier número, incluido cero, de métodos abstractos o no abstractos.
3. Las clases abstractas no deben ser marcadas como private o final.
4. Una clase abstracta que extiende a otra clase abstracta hereda todos sus métodos abstractos como suyos.
5. La primera clase concreta que extiende una clase abstracta debe proporcionar una implementación para todos los métodos abstractos heredados.

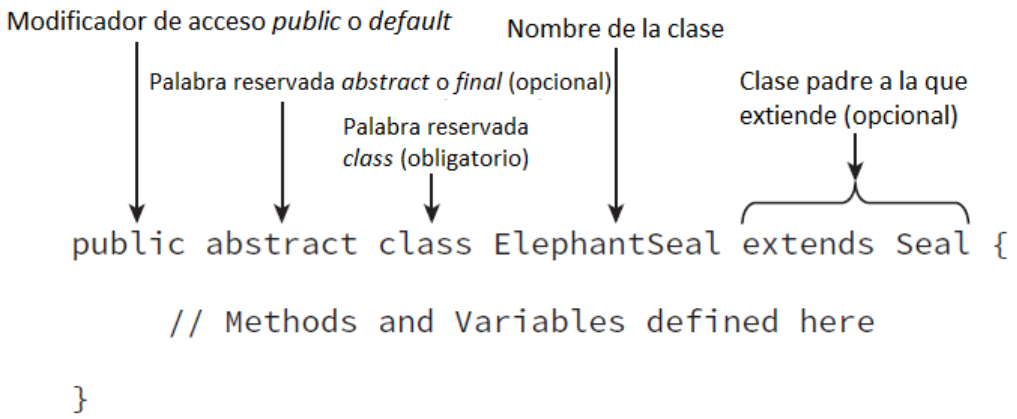
Diseño de clases

Métodos Abstractos – Reglas

1. Los métodos abstractos solo se pueden definir en clases abstractas.
2. Los métodos abstractos no pueden ser declarados como private o final.
3. Los métodos abstractos no deben implementar el cuerpo en la clase abstracta en la que han sido declarados.
4. La definición de un método abstracto en una subclase sigue las mismas reglas para sobrescribir un método. Por ejemplo, el nombre y la cabecera deben ser la misma, y la visibilidad del método en la subclase debe ser al menos tan accesible como en el método de la clase padre.

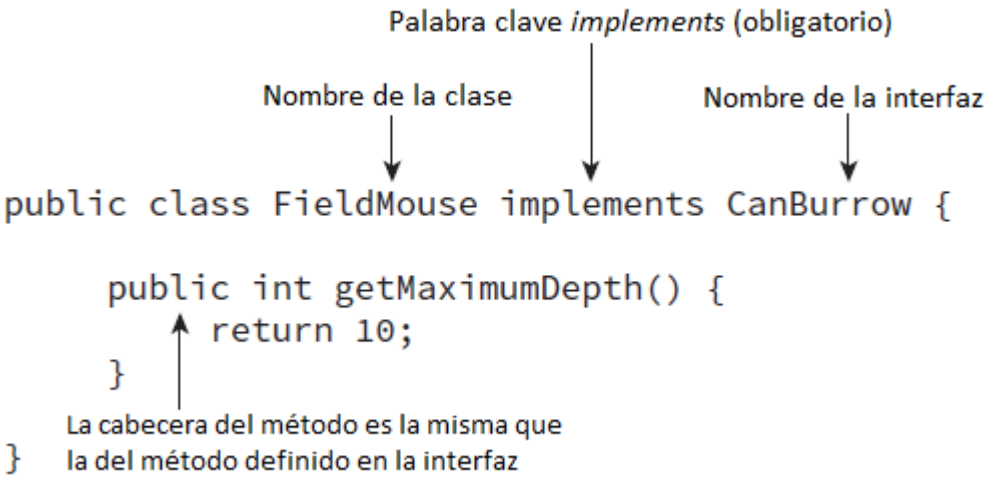
Diseño de clases

Interfaces



Diseño de clases

Interfaces



Diseño de clases

Interfaces

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

Diseño de clases

Interfaces - Reglas

1. Las interfaces no se pueden instanciar directamente.
2. Una interfaz no requiere implementar métodos.
3. Una interfaz no puede estar marcada como final.
4. Se asume que todas las interfaces de alto nivel tienen acceso público o por defecto (public, default), y deben incluir el modificador abstract en su definición. Por lo tanto, marcar una interface como private, protected o final, lanzará un error de compilación, al ser incompatible con esta definición.
5. Se asume que todos los métodos, en una interfaz, que no están marcados como default tienen los modificadores abstract y public en su definición. Por lo tanto, marcar un método como private, protected o final lanzará un error de compilación al ser éstos incompatibles con las palabras reservadas abstract y public.

Diseño de clases

Interfaces

```
public interface WalksOnTwoLegs {}

public class TestClass {
        public static void main(String[] args) {
                WalksOnTwoLegs example = new WalksOnTwoLegs();          // DOES NOT COMPILE
        }
}

...

public final interface WalksOnEightLegs {                              // DOES NOT COMPILE
}
```

Diseño de clases

Interfaces

```
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();
}
...
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
}
```

Diseño de clases

Interfaces - Herencia

```
public interface HasTail {
	public int getTailLength();
}

...

public interface HasWhiskers {
	public int getNumberOfWhiskers();
}

...

public interface Seal extends HasTail, HasWhiskers {
}
```

Diseño de clases

Interfaces - Herencia

```
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
 }
```

Diseño de clases

Interfaces – Clases, Interfaces y palabras clave

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

Diseño de clases

Interfaces – Métodos abstractos y múltiples interfaces

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

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

...

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

Diseño de clases

Interfaces – Métodos abstractos y múltiples interfaces

```
public interface Herbivore {
	public int eatPlants(int quantity);
}

public interface Omnivore {
	public void eatPlants();
}

public class Bear implements Herbivore, Omnivore {
	public int eatPlants(int quantity) {
		System.out.println("Eating plants: "+quantity);
		return quantity;
	}
	public void eatPlants() {
		System.out.println("Eating plants");
	}
}
```

Diseño de clases

Interfaces – Métodos abstractos y múltiples interfaces

```
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");
	}
}
```

Diseño de clases

Interfaces – Métodos abstractos y múltiples interfaces

```
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
```

Diseño de clases

Interfaces – Variables

1. Las variables en una interfaz se asumen que son public, static y final. Por lo tanto, marcar una variable como private o protected lanzara un error de compilación, ya que todas las variables se marcarán como abstract.

2. El valor de una variable de la interfaz debe declararse al estar declarada como final.

Diseño de clases

Interfaces – Variables

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

...

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
}
```

Diseño de clases

Interfaces – Métodos por defecto

4 reglas de los métodos por defecto de las interfaces:

1. Un método por defecto solamente puede estar declarado en una interfaz y no en una clase o clase abstracta.
2. Los métodos por defecto deben estar declarados con la palabra clave default, y debe implementar el cuerpo del método.
3. Los métodos por defecto no se asumen que son estáticos, finales o abstractos, al poder ser utilizados y sobrescritos por la clase que implemente la interfaz.
4. Como todos los métodos de una interfaz, un método por defecto se asume que es público y no compilará si se marca como privado o protegido.

Diseño de clases

Interfaces – Métodos por defecto

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

...

public interface Carnivore {
	public default void eatMeat();                          // DOES NOT COMPILE
	public int getRequiredFoodAmount() {                    // DOES NOT COMPILE
		return 13;
	}
}
```

Diseño de clases

Interfaces – Métodos por defecto

```
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;
	}
}
```

Interfaces – Métodos por defecto

Diseño de clases

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

Diseño de clases

Interfaces – Métodos estáticos

1. Como todos los métodos de una interfaz, el método estático se asume público y no compilará si se define como protected o private.

2. Para referenciar un método estático, se debe utilizar una referencia al nombre de la interfaz.

Diseño de clases

Interfaces – Métodos estáticos

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

Polimorfismo

Diseño de clases

```
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());
	}
}
```

Resultado

Polimorfismo

Diseño de clases

```
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());
	}
}
```

Resultado
```
10
False
True
```

Diseño de clases

Polimorfismo

```
HasTail hasTail = lemur;
System.out.println(hasTail.age);                    // NO COMPILA

Primate primate = lemur;
System.out.println(primate.isTailStriped());        // NO COMPILA
```

Diseño de clases

Objetos vs. Referencia

```
Lemur lemur = new Lemur();
Object lemurAsObject = lemur;

...

Primate primate = lemur;
Lemur lemur2 = primate;                                    // NO COMPILA
Lemur lemur3 = (Lemur)primate;
System.out.println(lemur3.age);
```

Diseño de clases

Casting - Reglas

1. Hacer casting a un objeto desde una subclase a una superclase no requiere un casting Explícito.

2. Hacer casting a un objeto desde una superclase a una subclase requiere un casting Explícito.

3. El compilador no va a permtir casting a tipos no relacionados.

4. Incluso cuando el código compila sin problemas, una excepción puede ser lanzada en tiempo de ejecución si el objeto al que se le está haciendo el casting no es en ese momento una instancia de esa clase.

Diseño de clases

Casting - Reglas

```
public class Bird {}
public class Fish {
	public static void main(String[] args) {
		Fish fish = new Fish();
		Bird bird = (Bird)fish;                                    // NO COMPILA
	}
}
```

Diseño de clases

Casting - Reglas

```
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
	}
}

...

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

Diseño de clases

Métodos Virtuales

```
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();
	}
}
```

Resultado

Diseño de clases

Métodos Virtuales

```
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();
	}
}
```

Resultado The bird name is: Peacock

Parámetros Polimorfos

Diseño de clases

```
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.getName());
	}
	public static void main(String[] args) {
		feed(new Alligator());
		feed(new Crocodile());
		feed(new Reptile());
	}
}
```

Resultado

Parámetros Polimorfos

Diseño de clases

```
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.getName());
	}
	public static void main(String[] args) {
		feed(new Alligator());
		feed(new Crocodile());
		feed(new Reptile());
	}
}
```

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

Diseño de clases

Parámetros Polimorfos

```
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());
	}
}
```