[HERENCIA JERÁRQUICA 1](#_Toc415768572)

[CLASES ABSTRACTAS 1](#_Toc415768573)

[Clase Abstracta ObjetoGeometrico 1](#_Toc415768574)

[Métodos Abstractos 2](#_Toc415768575)

[Subclases de la superclase ObjetoGeometrico 3](#_Toc415768576)

[¿Por qué métodos abstractos? 4](#_Toc415768577)

[Puntos Interesantes Sobre las Clases Abstractas 5](#_Toc415768578)

[CLASE ABSTRACTA NUMBER 6](#_Toc415768579)

[CLASE ABSTRACTA CALENDAR 8](#_Toc415768580)

# HERENCIA JERÁRQUICA

* Las clases se vuelven más específicas y concretas con cada nueva subclase (hacia abajo).
* Si se mueve de una subclase hacia una superclase (hacia arriba), las clases se vuelven más generales y menos específicas.
* El diseño de las clases debe asegurar que una superclase contiene características comunes para sus subclases.
* En ocasiones, una superclase es tan abstracta que no puede ser usada para crear alguna instancia (algún objeto) en específico. Dicha clase es conocida como clase abstracta.

# CLASES ABSTRACTAS

* Una clase abstracta no puede ser utilizada para crear objetos (instancias).
* Una clase abstracta puede tener métodos abstractos, los cuales se implementan en una subclase concreta.
* Considerar el siguiente código como ejemplo:

## Clase Abstracta ObjetoGeometrico

|  |
| --- |
| import java.util.Date;  public abstract class **ObjetoGeometrico** {  private String color = "blanco";  private boolean lleno;  private Date fechaCreacion;    protected **ObjetoGeometrico**() {  fechaCreacion = new Date();  }  ­  protected **ObjetoGeometrico**(String color, boolean lleno) {  fechaCreacion = new Date();  this.color = color;  this.lleno = lleno;  }    public String **getColor**() {  return this.color;  }    public void **setcolor**(String color) {  this.color = color;  }    public boolean **isLleno**(){  return lleno;  }    public void **setLleno**(boolean lleno){  this.lleno = lleno;  }    public abstract double **getArea**();    public abstract double **getPerimetro**();    } |

* La clase **ObjetoGeometrico** es una superclase que representa/modela características comunes de objetos geométricos.
* El constructor de la clase **ObjetoGeometrico** es definido como protected, porque solo es usado por subclases.

## Métodos Abstractos

* Es un hecho que se puede calcular el área y el perímetro de todos los objetos geométricos. Entonces, se crean los métodos **getArea()** y **getPerimetro()** en la clase **ObjetoGeometrico**.
* Sin embargo, estos métodos no pueden ser implementados en la clase **ObjetoGeometrico,** porque su implementación depende de un objeto geométrico en específico (de una subclase).
* Es decir, el área o perímetro de un objeto no siempre se calcula igual que la de otro objeto geométrico. Por ejemplo: el área de un círculo no se calcula de la misma manera en que se calcula el área de un rectángulo. Lo mismo ocurre con el cálculo del perímetro.
* Debido a esto, los métodos **getArea()** y **getPerimetro()** deben ser abstractos. Para que un método sea abstracto se debe anotar el modificador **abstract** en el encabezado del método. Además, se deben definir **sin implementación**, es decir, no deben tener un cuerpo, a continuación un ejemplo:

|  |
| --- |
| public abstract double **getArea**();    public abstract double **getPerimetro**(); |

* Una clase que tiene métodos abstractos debe definirse como **clase abstracta**, se debe anotar el modificador **abstract** en el encabezado de la clase. La clase **ObjetoGeometrico** contiene métodos abstractos, por lo tanto debe definirse como clase abstracta:

|  |
| --- |
| public abstract class **ObjetoGeometrico** |

**Entonces:**

* Un método abstracto se define sin implementación, su implementación es proporcionada por las subclases de la superclase.
* Los métodos abstractos de una superclase deben ser sobre escritos e implementados en sus subclases.

## Subclases de la superclase ObjetoGeometrico

|  |  |
| --- | --- |
| class **Circulo** extends *ObjetoGeometrico* {  private double radio;    public **Circulo**() {  }    public **Circulo**(double radio) {  this.radio = radio;  }    public double **getRadio**() {  return radio;  }    public void **setRadio**(double radio) {  this.radio = radio;  }    @Override  public double **getArea**() {  return radio \* radio \* Math.*PI*;  }    public double **getDiametro**() {  return radio \* 2;  }    @Override  public double **getPerimetro**() {  return 2 \* radio \* Math.*PI*;  }  } | class **Rectangulo** extends *ObjetoGeometrico* {  private double ancho;  private double alto;    public **Rectangulo**() {  }    public **Rectangulo**(double ancho, double alto) {  this.ancho = ancho;  this.alto = alto;  }    public double **getAncho**(){  return ancho;  }    public void **setAncho**(double ancho) {  this.ancho = ancho;  }    public double **getAlto**(){  return alto;  }    public void **setAlto**(double alto) {  this.alto = alto;  }    @Override  public double **getArea**() {  return ancho \* alto;  }    @Override  public double **getPerimetro**() {  return 2 \* (ancho + alto);  }  } |

* **ObjetoGeometrico** representa/modela características comunes de objetos geométricos.
* **ObjetoGeometrico** es la superclase de las subclases **Circulo** y **Rectangulo**.
* Un método abstracto no puede ser implementado (no se coloca cuerpo) en la superclase, el método abstracto debe ser implementado en las subclases.
* Los métodos getArea() y getPerimetro() de la superclase ObjetoGeometrico se implementan (se coloca cuerpo) en las subclases Circulo y Rectangulo, como se muestra en las imágenes de arriba.
* Los métodos abstractos de la superclase se sobre escriben (@Override) en las subclases.

## ¿Por qué métodos abstractos?

¿Cuál es la ventaja de definir los método **getArea()** y **getPerimetro()** como abstractos en la clase **ObjetoGeometrico**? El código siguiente muestra los beneficios:

|  |
| --- |
| class **CirculosYRectangulos** {  public static void ***main***(String[] args) {  *Circulo* miCirculo = new *Circulo*(5);  *Rectangulo* miRectangulo = new *Rectangulo*(5, 3);    *verInformacion*(miCirculo);  *verInformacion*(miRectangulo);    boolean comparacionAreas = *compararAreas*(miCirculo, miRectangulo);  System.*out*.println("**\n**Los dos objetos tienen areas iguales: "  + comparacionAreas);  }    public static void ***verInformacion***(*ObjetoGeometrico* objeto) {  System.*out*.println();  System.*out*.println("Area: " + objeto.*getArea*());  System.*out*.println("Perimetro: " + objeto.*getPerimetro*());  }    public static boolean ***compararAreas***(*ObjetoGeometrico* primero, *ObjetoGeometrico* segundo) {  return primero.*getArea*() == segundo.*getArea*();  }  } |

* El programa crea dos objetos geométricos, **miCirculo** y **miRectangulo**.
* Se invoca al método **comparacionAreas()** para verificar si los dos objetos geométricos tienen áreas iguales.
* Se invoca al método **verInformación()** para mostrar la información de dichos objetos.
* Cuando se invoca al método **compararAreas()**, el método **getArea()** definido en la clase **Circulo** es usado para **primero.getArea()** ya que el *argumento* **primero** es una instancia de tipo **ObjetoGeometrico** y el parametro **miCirculo** también lo es**.**
* Lo mismo sucede con el **Rectangulo**, el método **getArea()** definido en la clase **Rectangulo** es usado para **segundo.getArea()** ya que el *argumento* **segundo** es una instancia de **ObjetoGeometrico** y el *parametro* **miRectangulo** también lo es.
* Algo parecido ocurre cuando se invoca al método **verInformacion(miCirculo)**, los métodos **getArea()** y **getPerimetro()** definidos en **Circulo** son usados para **objeto.getArea()** y **objeto.getPerimetro().**
* Lo mismo pasa cuando se invoca de nuevo al método **verInformacion(miRectangulo)**. Los métodos **getArea()** y **getPerimetro()** definidos en **Rectangulo** son usados para **objeto.getArea()** y **objeto.getPerimetro().**
* No podríamos definir el método **compararAreas()** si el método **getArea()** no fuera definido en la clase **ObjetoGeometrico**.

# Puntos Interesantes Sobre las Clases Abstractas

* Una clase abstracta es como una clase normal, pero no se pueden crear instancias de una clase abstracta usando el operador new.
* Un método abstracto es definido sin implementación, su implementación es proporcionada por las subclases.
* Una clase que contiene métodos abstractos debe definirse como abstracta.
* Un método abstracto no puede ser contenido por una clase no abstracta.
* Si una subclase de una superclase abstracta no implementa todos los métodos, la subclase debe definirse como abstracta.
* Los métodos abstractos son no estáticos.
* Una clase abstracta no puede ser instancia utilizando el operador new, pero puedes definir su constructor o constructores. Dicho constructor será invocado en el constructor de la subclase.
* Una clase que contiene métodos abstractos debe ser definida como abstracta. Sin embargo es posible definir una clase abstracta sin métodos abstractos. En este caso, tampoco se puede instanciar utilizando el operador new, esta clase es utilizada como una clase base para definir subclases.
* Una subclase puede sobre escribir un método de su superclase para definirla como abstracta. Esto es muy inusual, pero es muy útil cuando la implementación del método en la superclase se vuelve inválido en la subclase. En este caso, la subclase debe definirse como abstracta.
* Una subclase puede ser abstracta incluso si su superclase no lo es (clase concreta, clase normal). Por ejemplo: la clase **Object** es concreta pero sus subclases, como **ObjetoGeometrico**, pueden ser abstractas.
* No se pueden crear instancias de una clase abstracta pero si se puede utilizar como tipo de dato.

|  |
| --- |
| class **AbstractTipoDato** {  public static void ***main***(String[] args) {    *ObjetoGeometrico*[] objetos = new *ObjetoGeometrico*[10];    objetos[0] = new *Circulo*();  objetos[1] = new *Circulo*(7);  objetos[2] = new *Rectangulo*(10, 20);    }  } |

# CLASE ABSTRACTA NUMBER

* **Number**  es una súper clase abstracta para las clases numéricas **BigInteger** y **BigDecimal**. Estas clases tienen métodos en común: ***byteValue()****,* ***shortValue****,* ***intValue****,* ***longValue****,* ***floatValue*** y ***doubleValue()*** para retornar un valor ***byte***,***short***, ***int***, ***long***, ***float y double*** respectivamente a un objeto/instancia de dichas clases. Estos métodos están definidos en la clase **Number**, la cual es una súper clase para las clases numéricas **BigInteger** y **BigDecimal** como se muestra en la siguiente figura.

|  |
| --- |
|  |

|  |
| --- |
| package java.lang;  public abstract class **Number** implements java.io.Serializable {  public abstract int **intValue**();  public abstract long **longValue**();  public abstract float **floatValue**();  public abstract double **doubleValue**();  public byte **byteValue**() {  return (byte)intValue();  }  public short **shortValue**() {  return (short)intValue();  }  private static final long *serialVersionUID* = -8742448824652078965L;  } |

* Desde que los métodos **intValue()**, **longValue()**, **floatValue()**, y **doubleValue()** no son implementados en la clase **Number**, dichos métodos son definidos como métodos abstractos. Por lo tanto, la clase **Number** también es definida como abstracta.
* Los métodos **byteValue()** y **shortValue()** son implementados del método **intValue()** como se muestra a continuación:

|  |
| --- |
| public byte **byteValue**() {  return (byte)intValue();  }  public short **shortValue**() {  return (short)intValue();  } |

* Con **Number** definida como la súper clase para las clases numéricas podemos definir métodos para realizar operaciones comunes con números. El siguiente programa encuentra el número mayor en una lista de objetos **Number**.

|  |
| --- |
| import java.util.ArrayList;  import java.math.BigDecimal;  import java.math.BigInteger;  public class **NumeroMayor** {    public static Number ***getNumeroMayor***(ArrayList<Number> lista){    if ( (lista == null) || (lista.size() == 0)) {  return null;  }    Number numero = lista.get(0);  for (int i=1; i<lista.size(); i++) {  if( numero.doubleValue() < lista.get(i).doubleValue() ){  numero = lista.get(i);  }  }  return numero;  }    public static void ***main*** (String[] args) {  ArrayList<Number> lista = new ArrayList<>();  lista.add(45);  lista.add(3445.53);  lista.add(new BigInteger("3432323234344343101"));  lista.add(new BigDecimal("2.0909090989091343433344343"));    Number mayor = *getNumeroMayor*(lista);  System.*out*.println("EL mayor es: " + mayor);  }  } |

* El programa crea un **ArrayList** de objetos **Number**, agrega un objeto **Integer**, un objeto **Double**, un objeto **BigInteger** y un objeto **BigDecimal**.
* Notar que el número **45** es automáticamente convertido a **Integer** y se agrega a la lista, ocurre lo mismo con el número **3445.53** que es automáticamente convertido en un objeto **Double** y se agrega a la lista usando autoboxing.
* Invocando el método **getNumeroMayor** retorna el número mayor en la lista, el método retornará null si la lista está vacía/nula o si el tamaño de la lista es cero.
* Para encontrar el número mayor en la lista, los números son comparados invocando a su método **doubleValue()**, este método es definido en la clase **Number** el implementado en una sub clase concreta de **Number**. Si un número es un objeto **Integer**, el método **doubleValue()** de **Integer** es invocado. Si un número es un objeto **BigDecimal**, el método **doubleValue**() de **BigDecimal** es invocado.
* Si el método **valueDouble()** no estuviera definido en la clase **Number** no podríamos encontrar el número más largo entre diferentes tipos de números usando la clase **Number**.

# CLASE ABSTRACTA CALENDAR

* **GregorianCalendar** es una sub clase concreta de la clase abstracta **Calendar**.
* Una instancia de import.java.util.**Date** representa un instante de tiempo específico con milisegundos.
* java.util.**Calendar** es una clase abstracta base para extraer información detallada del calendario, como el año, mes, día, hora, minuto y segundo. Las subclases de **Calendar** pueden implementar un sistema de calendario específico, como el calendario gregoriano o el calendario lunar. El calendario gregoriano es soportado en java por la clase java.util.**GregorianCalendar**.
* El método ***add()*** de la clase **Calendar** es abstracto porque su implementación depende de un sistema de calendario en concreto.

|  |
| --- |
|  |

* Se puede usar **new GregorianCalendar** para construir un **GregorianCalendar** por defecto con la fecha actual y **new GregorianCalendar(year, month, date)** para construir un **GregorianCalendar** con el año, mes y día específico. El parámetro mes es basado en 0, quiere decir que para enero corresponde 0, febrero 1 y así consecutivamente.
* El método **get(int field)** definido en la clase **Calendar** es muy útil para extraer la fecha e información de tiempo de un objeto **Calendar**. Los campos son definidos como constantes como se muestra en la tabla:

|  |
| --- |
|  |

* El siguiente ejemplo la clase abstracta **Calendar** y la clase **GregorianCalendar**.

|  |
| --- |
| import java.util.Calendar;  import java.util.Date;  import java.util.GregorianCalendar;  public class **CalendarPrueba** {  public static void ***main*** (String[] args) {    Calendar calendario = new GregorianCalendar();    System.*out*.println("Fecha actual: " + new Date());  System.*out*.println("Año: " + calendario.get(Calendar.*YEAR*));  System.*out*.println("Mes: " + calendario.get(Calendar.*MONTH*));  System.*out*.println("Día: " + calendario.get(Calendar.*DATE*));    Calendar miCalendario = new GregorianCalendar(2015, 03, 02);  String[] dias = {"Sábado", "Domingo", "Lunes", "Martes", "Miércoles", "Jueves", "Viernes"};  System.*out*.println("El 1 de Abril fue: ");  System.*out*.print(dias[miCalendario.get(Calendar.*DAY\_OF\_WEEK*) -1 ]);  }  } |

* El método **set (int campo, valor)** definido en la clase **Calendar** puede ser usado para establecer un campo. Por ejemplo: podemos usar **objCalendario.set(Calendar.DAY\_OF\_MONTH, 1)** para establecer el primer día del mes en el calendario **objCalendario**.
* El método **add (campo, valor)** que es *abstracto*, agrega la cantidad específica a un campo dado. Por ejemplo: **objCalendario.add(Calendar\_DAY\_OF\_MONTH, 5)** agrega 5 días al calendario actual, y, **objCalendario.add(Calendar\_DAY\_OF\_MONTH, -5)** substrae 5 días del objeto calendario.
* Para obtener el número máximo de días en un mes se utiliza: **objCalendario.getActualMaximun(Calendar.DAY\_OF\_MONTH)**. Si el calendario fuera de marzo, este método retornaría 31.

Se puede establecer un tiempo representado en un objeto Date para un calendario invocando calendar.setTime (date) y recuperar el tiempo invocando calendar.getTime ().