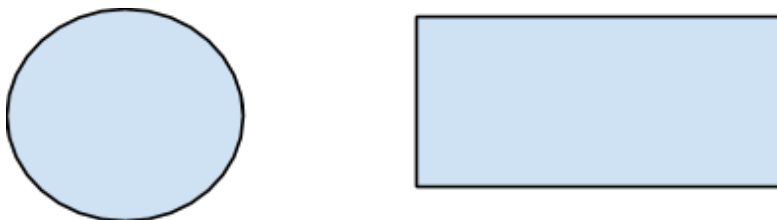


6. Herencia y polimorfismo¹

A manera de introducción, vamos a plantear una situación que nos permita estudiar los conceptos de herencia, polimorfismo, sobrecarga, overriding e interfaces:

Desarrollaremos un *software* que permita calcular el área y perímetro de figuras geométricas, vamos a comenzar con un rectángulo y un círculo, pero dejaremos el código para después poder extenderlo y reutilizar partes del mismo.



El dominio del problema es bastante sencillo, sabemos que necesitaremos calcular el área y perímetro de figuras geométricas, en este caso un *círculo* y un *rectángulo*. Aunque parezca que no tienen mucho en común, al menos ambos objetos tienen: un nombre, un área y un perímetro, aunque no necesariamente se calculan igual.

De nuestras clases de matemáticas sabemos que:

Área de un círculo = πr^2

Perímetro de un círculo = $2\pi \times r$

Área de un rectángulo = base x altura

Perímetro de un círculo = $2 \times (\text{base} + \text{altura})$

Herencia

De forma teórica, tenemos las siguientes definiciones sobre herencia de profesores de la UNAM:

La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la OOP. Esto es, el código de cualquiera de las clases puede ser utilizado sin más que crear una clase derivada de ella, o bien una *subclase*.

¹ Documento elaborado por el Ing. Arturo Nereu Núñez.

Hay dos tipos de herencia: *Herencia Simple* y *Herencia Múltiple*. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java sólo permite herencia simple. [1]

De nuestro ejemplo sobre figuras geométricas, sabemos que podemos definir una clase que se llame **Figura** y que contenga las propiedades: *nombre*, *area* y *perimetro*.

```
public class Figura
{
    string nombre;
    float area;
    float perimetro;

    //Constructor
    public Figura()
    {
        //Asignamos valores por default a una figura
        nombre = "Sin definir";
        area = 0;
        perimetro = 0;
    }

    public void imprimeInformacion()
    {
        System.out.println("Nombre: " + nombre);
        System.out.println("Area: " + area);
        System.out.println("Perimetro: " + perimetro);
    }
}
```

Tenemos una abstracción de nuestra figura geométrica y sabemos que cualquier objeto que sea una figura geométrica podrá tener un nombre, área y perímetro. Sin embargo, es tan abstracto este concepto que no nos sirve de mucho, ya que como tal una figura geométrica no tiene fórmula para calcular su área ni perímetro, necesitamos ser más específicos.

```
public class EjemploFiguras
{
    public static void main(String[] args)
    {
        //Instanciamos un nuevo objeto de tipo figura
        Figura figura = new Figura();

        //Imprimos su informacion
        figura.imprimeInformacion();
    }
}
```

Si ahora creamos una clase de ejemplo e instanciamos una clase figura, tendríamos el siguiente resultado:

Nombre: Sin definir
Area: 0.0
Perímetro: 0.0

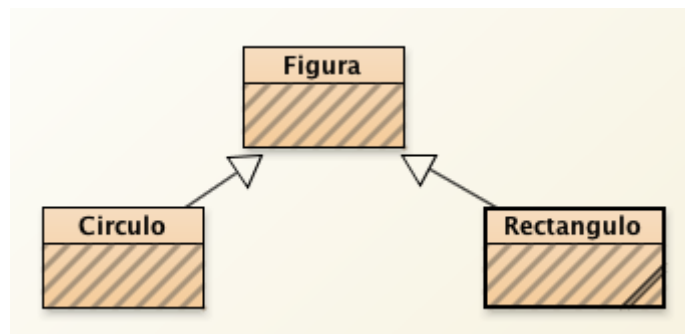
Vamos entonces a declarar nuestras clases *Rectangulo* y *Circulo*:

```
public class Rectangulo extends Figura  
{  
    //Implementacion de la clase  
}
```

```
public class Circulo extends Figura  
{  
    //Implementacion del circulo  
}
```

Antes de implementar nuestras clases *Rectangulo* y *Circulo*, vamos a ver que al definir la clase, le agregamos la palabra reservada **extends** seguida del nombre de la clase de la cual vamos a heredar, es decir **Figura** en nuestro ejemplo.

El diagrama de relación con nuestras clases definidas queda así:



Tanto la clase *Circulo* como la clase *Rectangulo* **heredan** de la clase *Figura*, a esto se le llama que son **Subclases**. En el caso de *Figura*, se dice que es una **Superclase** con referencia a las otras dos.

Lo que está pasando es que *Circulo* y *Figura* ahora tienen los métodos y propiedades `protected` y `public` que están declarados en la clase *Figura*.

Ahora vamos a implementar la clase *Rectangulo*:

```
public class Rectangulo extends Figura
{
    float base;
    float altura;

    //Constructor del rectangulo
    public Rectangulo(float base, float altura)
    {
        nombre = "Rectangulo";
        this.base = base;
        this.altura = altura;
    }

    public void calculaArea()
    {
        area = base * altura;
    }

    public void calculaPerimetro()
    {
        perimetro = 2 * (base + altura);
    }
}
```

Nuestra clase rectangulo tiene como propiedades las variables *base* y *altura* definidas como float.

Ahora, en el constructor, recibimos valores para ponerlos a las variables. Dentro del mismo constructor, vemos que asignamos a la variable nombre el valor del string "Rectngulo" aunque en esta clase no hay ninguna variable llamada nombre. ¿Marca error de compilación? No, dado que al ser *Rectangulo* una subclase de *Figura*, hereda sus propiedades y métodos protected y public. Es así como podemos obtener sus valores como si fueran declarados en la misma clase.

La clase circulo sería así:

```
public class Circulo extends Figura
{
    float radio;

    public Circulo(float radio)
    {
        nombre = "Circulo";
        this.radio = radio;
    }

    public void calculaArea()
    {
        area = (float)(Math.PI * radio * radio);
    }

    public void calculaPerimetro()
    {
        perimetro = (float)(2 * Math.PI * radio);
    }
}
```

La definición de la clase tiene los mismos métodos, pero diferentes propiedades específicas de la clase, aunque hereda los mismos de la superclase figura.

Cabe notar que en el cálculo del área y perímetro del círculo usamos **Math.PI**, ésta es una variable (en realidad constante) estática de la clase Math de Java. Por ahora, basta con saber que en esa referencia está el valor del número PI. Para conocerlas a fondo, visita el enlace:

<http://monillo007.blogspot.mx/2008/05/variables-y-mtodos-esticos-en-java.html>.

También vemos que tanto en área y perímetro ponemos **(float)** en la asignación, esto es porque el cálculo del perímetro y área regresa un tipo **double**, es decir más precisión que lo que se guarda en un **float** (que es como está definido área y perímetro). A esto se le llama *casting* y es una manera de convertir un tipo de dato en otro.

En nuestro programa de ejemplo vamos a crear un objeto círculo:

```
public class EjemploFiguras
{
    public static void main(String[] args)
    {
        //Instanciamos un nuevo objeto de tipo figura
        Figura figura = new Figura();
        //Imprimos su informacion
        figura.imprimeInformacion();

        //Vamos a instanciar un circulo
        Circulo circulo = new Circulo(5.0f);
        //Calculamos su area y perimetro
        circulo.calculaArea();
        circulo.calculaPerimetro();
        //Imprimos su informacion
        circulo.imprimeInformacion();
    }
}
```

Creamos un nuevo círculo, luego calculamos su área y perímetro e imprimimos su valor. Recuerda que aunque la clase Circulo no tiene el método *imprimeInformacion()* definido, sí lo hereda de Figura.

El resultado en consola es:

```
Nombre: Sin definir
Area: 0.0
Perímetro: 0.0
Nombre: Circulo
Area: 78.53982
Perímetro: 31.415926
```

Obtenemos el resultado que buscábamos, como ejercicio, en el programa agrega un rectángulo, calcula su área y perímetro e imprime su información.

Visita el siguiente enlace para conocer cómo funcionan los constructores y la herencia en JAVA:
<https://sites.google.com/site/pro012iessanandres/java/constructores-y-herencia>

Polimorfismo

Recordemos que el polimorfismo se refiere a la capacidad de un objeto para comportarse como otro, siempre y cuando pertenezca a la misma jerarquía de herencia.

En nuestro caso, ya vimos el polimorfismo en acción, cuando mandamos a llamar el método

```
circulo.imprimeInformacion();
```

Realmente nuestro objeto `circulo` lo declaramos como un `Circulo`, no como una `Figura`. El comportamiento polimórfico se da, dado que un círculo **es una** figura, por lo tanto puede comportarse como tal.

Hay un operador en Java (y en otros lenguajes) que nos ayuda a saber si un objeto responde a cierta jerarquía y así podemos ejecutar código sabiendo que dicho objeto pertenece a alguna jerarquía.

El operador `instanceof` regresa un valor boolean; `true` si la variable es del tipo enviado como parámetro y `false` en caso de no serlo:

```
if(circulo instanceof Figura)
{
    System.out.println("Circulo es una figura");
}

if(circulo instanceof Circulo)
{
    System.out.println("Circulo es una Circulo");
}

if(circulo instanceof Rectangulo)
{
    System.out.println("Circulo es una Rectangulo");
}
```

En este ejemplo solamente las dos primeras condiciones serían **true** e imprimirán su resultado dado que el círculo es tanto un *Circulo* como una *Figura*.

Sobrecarga de métodos

La sobrecarga de métodos se refiere a la posibilidad de nombrar de la misma manera a diferentes métodos con la condición de que sean diferentes tanto los parámetros como el valor de retorno.

Por ejemplo, en nuestra clase *Circulo*, podríamos definir otro parámetro con el nombre *calculaArea*, pero en lugar de no recibir parámetros y regresar **void**, podríamos tener:

```
public void calculaArea()
{
    area = (float)(Math.PI * radio * radio);
}

public float calculaArea(float nRadio)
{
    return (float)(Math.PI * nRadio * nRadio);
}
```

En la clase *circulo* estamos sobrecargando el método *calcula área*. Ambos tienen el mismo nombre; sin embargo, cambia el valor de retorno y los parámetros que se reciben. La segunda versión del método podría ser usada si se quiere calcular el área de un círculo sin necesidad de crear otro, sino usando una instancia ya creada.

Sobreescritura (overriding) de métodos.

La sobreescritura de métodos va ligada más a la herencia y se refiere a “redefinir” métodos en una subclase que previamente fueron definidos en una superclase.

En nuestro caso, tenemos el método “*imprimeInformacion*” definido en la clase *Figura*. Si lo llamamos desde una subclase, sabemos que tendrá el formato que se define ahí.

Ahora, en nuestra clase *rectangulo* vamos a sobrecribir dicho método:

```
public void imprimeInformacion()
{
    System.out.println("Soy un rectangulo");
    System.out.println("Mi base es: " + base);
    System.out.println("Mi altura es: " + altura);
}
```

Si en nuestro ejemplo ejecutamos *imprimeInformacion* sobre un objeto *rectangulo*, ya no vamos a obtener la salida definida en *Figura*, sino la nueva que sobreescribimos:

```
public class EjemploFiguras
{
    public static void main(String[] args)
    {
        Rectangulo rect = new Rectangulo(2.0f, 5.0f);
        rect.imprimeInformacion();
    }
}
```

Interfaces

Para terminar la unidad, revisa el siguiente vínculo donde se habla sobre interfaces en Java y su relación con el tema de la herencia: <http://www.arrakis.es/~abelp/ApuntesJava/Interfaces.htm>

Referencias

[1] Herencia: http://profesores.fi-b.unam.mx/carlos/java/java_basico3_4.html

Ejemplo adaptado de la unidad:

<http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/lecture3c.html>

Variables estáticas JAVA: <http://monillo007.blogspot.mx/2008/05/variables-y-mtodos-estticos-en-java.html>

Constructores y herencia: <https://sites.google.com/site/pro012iessanandres/java/constructores-y-herencia>

Interfaces en JAVA: <http://www.arrakis.es/~abelp/ApuntesJava/Interfaces.htm>