```
abstract class Figura {
	abstract public void pintar();
}

class Circulo extends Figura {
	@Override
	public void pintar() {
	}
}

class Rectangulo extends Figura {
	@Override
	public void pintar() {
	}
}

class Lienzo {
	private List figuras;

	public void Lienzo() {
		figuras = new LinkedList();
	}

	public void add(List< ? extends Figura > listaFiguras) {
		for (Figura figura : listaFiguras) {
			this.figuras.add(figura);
		}
	}
}



public class Genericidad {
	void F() {
		Lienzo lienzo = new Lienzo();

		List< Figura > fs = new ArrayList<>();
		fs.add(new Circulo());
		fs.add(new Rectangulo());

		lienzo.add(fs);

		List<Circulo> circulos = new ArrayList<>();
		circulos.add(new Circulo());
		lienzo.add(circulos);
	}
}
```

¿Por qué hemos usado genericidad restringida en el método:

```
public void add(List<? extends Figura> listaFiguras)
```

Si el prototipo fuera:

```
public void add(List<Figura> listaFiguras)
```

la llamada lienzo.add(fs); sería correcta porque envía un Array_ist<Figura> para List<Figura>, que son compatibles.

Sin embargo, la llamada **lienzo.add(circulos)** envía ArrayList<Circulo> que no es compatible con List<Figura>. Esto se resuelve indicando que Lienzo.add puede recibir tanto una lista de Figura como de cualquier tipo que herede de Figura con genericidad restringida
<? extends Figura>, con lo que queda:

```
public void add(List<? extends Figura> listaFiguras)
```

Rellena con las palabras clave correctas de forma que compile, escribe la cadena VACIO si no es necesario que aparezca ninguna palabra clave

Suponemos que las clases a instanciar están todas en el paquete "figuras"

```
public static Figura crearFigura(String figura) throws InstantiationException, IllegalAccessException, ClassNotFoundException {
        Class<?> c = Class.  forName  ✔ (  VACIO  ✘ figura);
        Figura objeto =  (Figura)  ✔  c.  newInstance  ✔ ();
        return objeto;
}
```

En Java, los tipos genéricos sólo se pueden usar para parametrizar clases, como por ejemplo

```
class ClaseParametrizada<T> {
    T unCampo;
    void calcula(T a) {

       ....
    }
}
```

Seleccione una:

- Verdadero
- Falso ✔

No es cierto, también pueden valer para parametrizar métodos. P.ej. este método de clase:

```
static <T> boolean comparar(T a, T b) {
.....
}
```

En Java, podemos incluir varios tipos parametrizados dentro del operador diamante, e incluso anidar varios operadores diamante.

Seleccione una:

- Verdadero ✔
- Falso

Sí, dentro del operador diamante, es decir, <>, se pueden incluir un número arbitrario de tipos parametrizados. Un tipo parametrizado puede ser a su vez también parametrizado. Por ejemplo:

```
clase A<Tipo1, Tipo2<Tipo1>, Tipo3> {
    Tipo1 a;
    Tipo2<Tipo1> b;
    List<Tipo3> c;
}
```