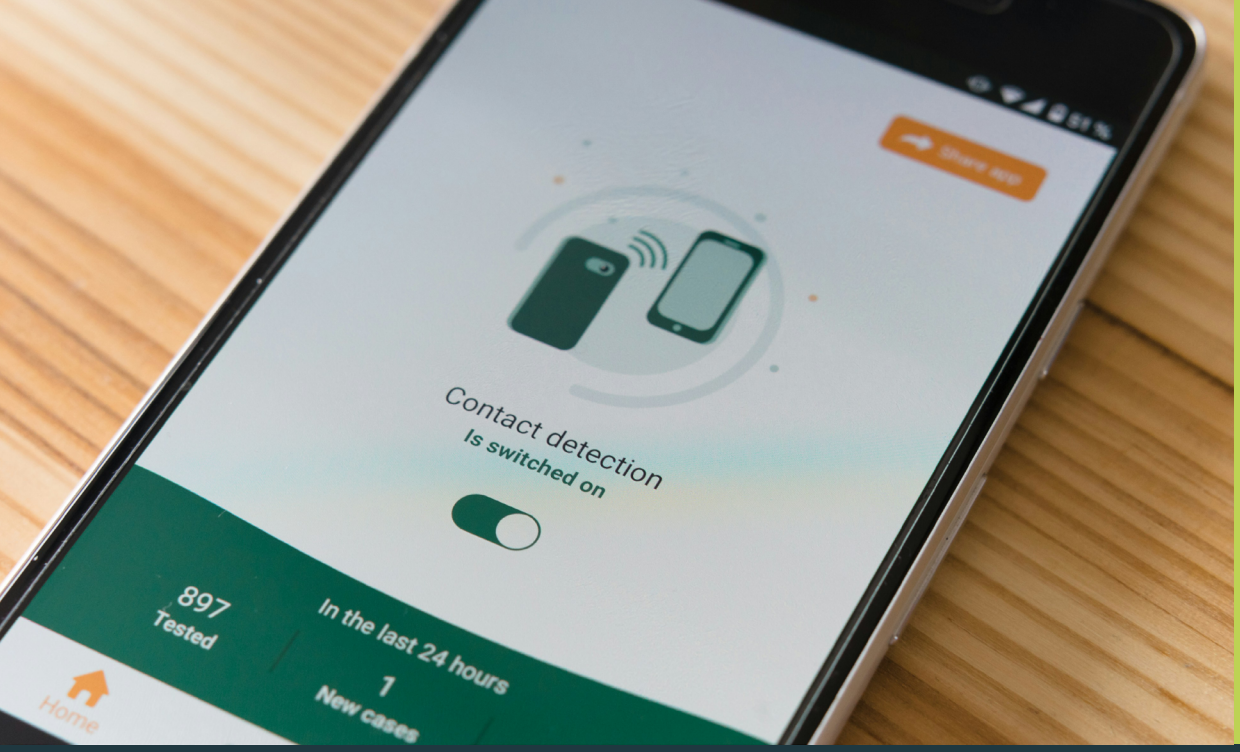

Unidad 11

Tiempo de ejecución

Contenidos

- Parte 1: Tiempo de Ejecución 3
- Parte 2: Comprobación Estática y Dinámica de Tipos 6
- Parte 3: Conversiones de Tipos entre Objetos (Casting) 9
- Parte 4: Clases y Tipos Genéricos o Parametrizados 13
- Conclusión del Bloque: Tiempo de ejecución 18

Parte 1: Tiempo de Ejecución

1. ¿Qué es la Ligadura Dinámica?

La **ligadura dinámica** (o **dynamic binding**) se refiere al proceso mediante el cual el sistema determina, en tiempo de ejecución, qué implementación de un método se debe invocar. Esto ocurre cuando:

1. Un método es sobrescrito en una subclase.

2. El tipo de referencia del objeto es diferente al tipo real del objeto.

1.2. Ligadura Estática vs. Ligadura Dinámica

Aspecto	**Ligadura Estática**	**Ligadura Dinámica**
Momento de Evaluación	Determinada en tiempo de compilación.	Determinada en tiempo de ejecución.
Uso Común	Métodos `static`, atributos y métodos finales.	Métodos sobrescritos.
Flexibilidad	Más rápida, pero menos flexible.	Más lenta, pero permite polimorfismo.

1.3. Ejemplo de Ligadura Dinámica

```
public class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}
public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}
public class Gato extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El gato maúlla.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal miAnimal1 = new Perro(); // Referencia de tipo Animal, objeto tipo Perro
        Animal miAnimal2 = new Gato();  // Referencia de tipo Animal, objeto tipo Gato
        miAnimal1.hacerSonido(); // Llama al método de la clase Perro
        miAnimal2.hacerSonido(); // Llama al método de la clase Gato
    }
}
```

Salida:

```
El perro ladra.
El gato maúlla.
```

1.4. Ventajas de la Ligadura Dinámica

1. Polimorfismo:

Permite que el comportamiento de un método dependa del tipo real del objeto, no del tipo de referencia.

2. Extensibilidad:

Facilita la adición de nuevas subclases sin modificar el código existente.

3. Flexibilidad:

Se pueden tratar los objetos de manera uniforme, incluso si pertenecen a diferentes tipos.

2. Ejercicio Guiado

Problema

1. Crea una clase base `Empleado` con un método `calcularSalario()`.

2. Crea dos subclases:

- `EmpleadoTiempoCompleto`: Implementa `calcularSalario()` con un salario fijo.
- `EmpleadoPorHoras`: Implementa `calcularSalario()` con cálculo basado en horas trabajadas.

3. En la clase principal, crea una lista de empleados y llama al método `calcularSalario()` de cada uno.

3. Preguntas de Evaluación

1. ¿Cuándo se determina la ligadura dinámica?

a) En tiempo de compilación.
b) En tiempo de ejecución.
c) Antes de la ejecución.
d) Durante el análisis estático.

2. ¿Qué mecanismo en Java usa la ligadura dinámica?

a) Sobrecarga de métodos.
b) Polimorfismo en tiempo de ejecución.
c) Métodos estáticos.
d) Clases abstractas únicamente.

3. ¿Qué sucede si un método `final` está en una superclase?

a) Puede ser sobrescrito en la subclase.
b) No puede ser sobrescrito.
c) Se evalúa dinámicamente.
d) Genera un error de compilación.

Parte 2: Comprobación Estática y Dinámica de Tipos

En Java, los **tipos de datos** juegan un papel crucial en la seguridad y consistencia del código. La comprobación de tipos asegura que las operaciones realizadas en los datos sean válidas y coherentes. Hay dos tipos principales de comprobaciones en Java:

1. Comprobación Estática de Tipos

1. Definición:

La **comprobación estática de tipos** ocurre en tiempo de compilación. El compilador verifica que los tipos utilizados en las variables, métodos y operaciones sean válidos según las reglas del lenguaje.

2. Características:

- Garantiza la detección de errores de tipo antes de ejecutar el programa.
- Es más rápida porque ocurre durante la compilación.
- Aumenta la seguridad del código, reduciendo errores en tiempo de ejecución.

3. Ejemplo:

```
  public class Main {
      public static void main(String[] args) {
            int numero = 10; // Correcto
            // String texto = numero; // Error de compilación: Incompatible
types
      }
  }
```

2. Comprobación Dinámica de Tipos

1. Definición:

La **comprobación dinámica de tipos** ocurre en tiempo de ejecución. El sistema verifica el tipo real de los objetos en tiempo de ejecución, especialmente en casos de polimorfismo y casting.

2. Características:

- Ocurre cuando el programa está en ejecución.
- Permite flexibilidad, como el uso de polimorfismo.
- Puede generar excepciones si los tipos no coinciden.

3. Ejemplo:

```java
public class Main {
    public static void main(String[] args) {
        Object obj = "Hola, Mundo";
        if (obj instanceof String) { // Verificación dinámica
            String texto = (String) obj; // Casting seguro
            System.out.println(texto);
        } else {
            System.out.println("El objeto no es una cadena.");
        }
    }
}
```

Salida:

```
Hola, Mundo
```

3. Diferencias entre Comprobación Estática y Dinámica

Aspecto	Comprobación Estática	Comprobación Dinámica
Momento	Ocurre en tiempo de compilación.	Ocurre en tiempo de ejecución.
Flexibilidad	Menos flexible, requiere tipos definidos.	Más flexible, permite polimorfismo y casting.
Errores Detectados	Errores de tipo son detectados antes de ejecutar el programa.	Errores de tipo pueden ocurrir en tiempo de ejecución.
Velocidad	Más rápida porque ocurre antes de ejecutar.	Puede ser más lenta debido a la verificación en tiempo real.

4. Palabra Clave `instanceof`

La palabra clave `instanceof` es fundamental para la comprobación dinámica de tipos. Permite verificar si un objeto es una instancia de una clase o de una subclase específica.

Ejemplo de Uso de `instanceof`

```
public class Animal {}
public class Perro extends Animal {}
public class Gato extends Animal {}
public class Main {
    public static void main(String[] args) {
          Animal miAnimal = new Perro();
          if (miAnimal instanceof Perro) {
                System.out.println(“Es un perro.”);
          } else if (miAnimal instanceof Gato) {
                System.out.println(“Es un gato.”);
          } else {
                System.out.println(“Es otro tipo de animal.”);
          }
    }
}
```

Salida:

```
Es un perro.
```

5. Ventajas y Desventajas de Cada Comprobación

Ventajas de la Comprobación Estática

1. **Detección Temprana de Errores:**

   Los errores se detectan antes de ejecutar el programa, ahorrando tiempo en depuración.

2. **Rendimiento Mejorado:**

   El código es más rápido en ejecución porque no requiere verificaciones adicionales.

3. **Seguridad:**

   Reduce la posibilidad de errores en tiempo de ejecución.

Ventajas de la Comprobación Dinámica

1. **Flexibilidad:**

   Permite trabajar con tipos más genéricos, como `Object`, y determinar su tipo en tiempo de ejecución.

2. **Compatibilidad con Polimorfismo:**

   Permite diseñar sistemas extensibles y dinámicos.

6. Ejercicio Guiado

Problema

1. **Define una clase base `Empleado` y dos subclases:**
   - `Gerente` con un método `dirigirReunion()`.
   - `Desarrollador` con un método `escribirCodigo()`.
2. **Crea una lista de empleados de diferentes tipos.**
3. **Usa `instanceof` para realizar operaciones específicas según el tipo de empleado.**

7. Preguntas de Evaluación

1. ¿Qué tipo de comprobación ocurre en tiempo de compilación?

a) Dinámica.
b) Estática.
c) Ambas.
d) Ninguna.

2. ¿Qué palabra clave se usa para comprobar el tipo de un objeto en tiempo de ejecución?

a) `classof`.
b) `typecheck`.
c) `instanceof`.
d) Ninguna de las anteriores.

3. ¿Cuál es la principal ventaja de la comprobación estática de tipos?

a) Flexibilidad.
b) Seguridad y detección temprana de errores.
c) Uso en polimorfismo.
d) Reducción de tiempo en compilación.

Parte 3: Conversiones de Tipos entre Objetos (Casting)

El **casting** en Java es el proceso de convertir un tipo de datos en otro. En el contexto de objetos, esto implica convertir un objeto de un tipo en otro tipo compatible dentro de la jerarquía de clases. Esto es común en aplicaciones que utilizan polimorfismo y herencia.

1. Tipos de Casting

1. Casting Implícito (Upcasting)

- Ocurre automáticamente cuando un objeto de una subclase se asigna a una referencia de su superclase.
- Siempre es seguro, ya que la subclase hereda todas las propiedades de la superclase.

```
Animal miAnimal = new Perro(); // Upcasting implícito
```

2. Casting Explícito (Downcasting)

- Necesita ser indicado explícitamente para convertir un objeto de la superclase en una referencia de la subclase.
- Requiere una comprobación previa con `instanceof` para garantizar la seguridad.

```
if (miAnimal instanceof Perro) {
    Perro miPerro = (Perro) miAnimal; // Downcasting explícito
}
```

2. Ejemplo de Casting

Upcasting Implícito

```
public class Animal {
    public void hacerSonido() {
        System.out.println("El animal hace un sonido.");
    }
}
public class Perro extends Animal {
    @Override
    public void hacerSonido() {
        System.out.println("El perro ladra.");
    }
}
public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro(); // Upcasting implícito
        miAnimal.hacerSonido(); // Llama al método de la subclase (ligadura dinámica)
    }
}
```

Salida:

```
El perro ladra.
```

Downcasting Explícito

```
public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Perro(); // Upcasting
        if (miAnimal instanceof Perro) { // Verificación de tipo
            Perro miPerro = (Perro) miAnimal; // Downcasting explícito
            miPerro.hacerSonido();
        }
    }
}
```

Salida:

```
El perro ladra.
```

3. Casting Inválido y Excepciones

Si se realiza un **downcasting** a un tipo incorrecto, Java lanzará una **ClassCastException** en tiempo de ejecución.

Ejemplo de Casting Inválido

```
public class Gato extends Animal {}
public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Gato();
        Perro miPerro = (Perro) miAnimal; // Error: ClassCastException
    }
}
```

Salida:

```
Exception in thread "main" java.lang.ClassCastException
```

4. Uso de `instanceof` para Evitar Errores

Es una buena práctica utilizar `instanceof` antes de realizar un **downcasting** para evitar excepciones.

Ejemplo

```
public class Main {
    public static void main(String[] args) {
        Animal miAnimal = new Gato();
        if (miAnimal instanceof Perro) {
            Perro miPerro = (Perro) miAnimal; // Solo se ejecuta si es seguro
            miPerro.hacerSonido();
        } else {
            System.out.println("El objeto no es un Perro.");
        }
    }
}
```

Salida:

```
El objeto no es un Perro.
```

5. Beneficios y Limitaciones del Casting

Beneficios

1. **Flexibilidad:**

   Permite trabajar con referencias más generales y específicas en jerarquías de clases.

2. **Polimorfismo:**

   Aprovecha el polimorfismo para manejar objetos de diferentes tipos de manera uniforme.

3. **Extensibilidad:**

   Facilita el diseño de sistemas que soportan nuevas clases en el futuro.

Limitaciones

1. **Posible ClassCastException:**

   Si el casting es incorrecto, puede fallar en tiempo de ejecución.

2. **Dependencia de instanceof:**

   Aumenta la complejidad cuando se debe verificar el tipo constantemente.

6. Ejercicio Guiado

Problema

1. **Define una clase base `Vehiculo` y dos subclases:**
   - `Coche` con un método `encenderLuces()`.
   - `Barco` con un método `soltarAncla()`.
2. **Crea una lista de vehículos y utiliza `instanceof` y casting para invocar los métodos específicos de cada tipo de vehículo.**

7. Preguntas de Evaluación

1. ¿Qué es el casting implícito?

a) Convertir un tipo base a un tipo derivado.
b) Convertir un tipo derivado a un tipo base.
c) Comprobar el tipo en tiempo de ejecución.
d) Ninguna de las anteriores.

2. ¿Qué ocurre si se realiza un downcasting a un tipo incorrecto?

a) El programa continúa ejecutándose.
b) Se lanza una excepción en tiempo de ejecución.
c) Produce un error de compilación.
d) El objeto se convierte al tipo base.

3. ¿Qué palabra clave se utiliza para verificar el tipo antes de realizar un casting?

a) `super`.
b) `extends`.
c) `instanceof`.
d) Ninguna de las anteriores.

Parte 4: Clases y Tipos Genéricos o Parametrizados

Los **genéricos** en Java son una característica poderosa que permite escribir clases, interfaces y métodos parametrizados con tipos de datos. Esto facilita el diseño de código reutilizable, seguro y flexible, mejorando la verificación de tipos en tiempo de compilación.

1. ¿Qué son los Genéricos?

1. Definición:

- Los genéricos permiten trabajar con tipos de datos que se especifican como parámetros en tiempo de compilación.
- Usan `<>` para declarar el tipo genérico.

2. Ventajas:

- **Seguridad de Tipos**: Detecta errores de tipo en tiempo de compilación.
- **Reutilización**: Las clases y métodos genéricos pueden trabajar con múltiples tipos de datos.
- **Legibilidad**: Reduce la necesidad de castings explícitos.

2. Clases Genéricas

Una **clase genérica** se define utilizando un parámetro de tipo que puede ser reemplazado por cualquier tipo de dato en tiempo de compilación.

Ejemplo de Clase Genérica

```
public class Caja<T> {
    private T contenido;
    public void guardar(T contenido) {
          this.contenido = contenido;
    }
    public T obtener() {
          return contenido;
    }
}
public class Main {
    public static void main(String[] args) {
          Caja<String> miCaja = new Caja<>();
          miCaja.guardar("Hola, Genéricos");
          System.out.println(miCaja.obtener());
          Caja<Integer> otraCaja = new Caja<>();
          otraCaja.guardar(123);
          System.out.println(otraCaja.obtener());
    }
}
```

Salida:

```
123
```

3. Métodos Genéricos

Los **métodos genéricos** permiten definir un método con un tipo genérico dentro de una clase genérica o no genérica.

Ejemplo de Método Genérico

```
public class Utilidades {
    public static <T> void imprimir(T elemento) {
        System.out.println(elemento);
    }
}
public class Main {
    public static void main(String[] args) {
        Utilidades.imprimir("Texto Genérico");
        Utilidades.imprimir(456);
        Utilidades.imprimir(3.14);
    }
}
```

Salida:

```
456
3.14
```

4. Tipos Genéricos en Colecciones

Las **colecciones genéricas** (como `ArrayList`, `HashMap`) son ampliamente utilizadas en Java para almacenar elementos de un tipo específico, mejorando la seguridad y eliminando la necesidad de castings.

Ejemplo con `ArrayList`

```
import java.util.ArrayList;
public class Main {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<>();
        lista.add("Elemento 1");
        lista.add("Elemento 2");
        for (String elemento : lista) {
            System.out.println(elemento);
        }
    }
}
```

Salida:

```
Elemento 1
Elemento 2
```

5. Restricciones de Tipos en Genéricos

Los genéricos pueden tener restricciones para limitar los tipos que se pueden usar.

Uso de Extends

Se puede restringir un tipo genérico a una clase base o interfaz específica usando `extends`.

```
public class CajaNumerica<T extends Number> {
    private T numero;
    public CajaNumerica(T numero) {
        this.numero = numero;
    }
    public double obtenerDoble() {
        return numero.doubleValue() * 2;
    }
}
public class Main {
    public static void main(String[] args) {
        CajaNumerica<Integer> cajaInt = new CajaNumerica<>(5);
        System.out.println("Doble: " + cajaInt.obtenerDoble());
        CajaNumerica<Double> cajaDouble = new CajaNumerica<>(5.5);
        System.out.println("Doble: " + cajaDouble.obtenerDoble());
    }
}
```

Salida:

```
Doble: 10.0
Doble: 11.0
```

6. Corchetes Múltiples y Comodines

Uso de `?` (comodín)

1. Comodín sin restricciones (`?`):

Permite cualquier tipo genérico.

```
public static void imprimirElementos(ArrayList<?> lista) {
    for (Object elemento : lista) {
         System.out.println(elemento);
    }
}
```

2. Comodín con restricciones superiores (`? extends T`):

Limita los elementos a un tipo específico o sus subclases.

```
public static double sumarNumeros(ArrayList<? extends Number> lista) {
    double suma = 0;
    for (Number numero : lista) {
         suma += numero.doubleValue();
    }
    return suma;
}
```

3. Comodín con restricciones inferiores (`? super T`):

Limita los elementos a un tipo específico o sus superclases.

```
public static void añadirNumeros(ArrayList<? super Integer> lista) {
    lista.add(10);
    lista.add(20);
}
```

7. Ejercicio Guiado

Problema

1. Crea una clase genérica `Par` que almacene dos valores de cualquier tipo (`T` y `U`).
2. Define métodos para obtener y modificar los valores.
3. En la clase principal, crea un par de enteros, un par de cadenas y un par mezclado (entero y cadena).
4. Imprime los valores de los pares.

8. Preguntas de Evaluación

1. ¿Qué permite la utilización de genéricos en Java?

a) Verificar tipos en tiempo de ejecución.
b) Reducir el código reutilizable.
c) Verificar tipos en tiempo de compilación.
d) Hacer el código más específico.

2. ¿Qué ocurre si no se especifica el tipo en una colección genérica?

a) Genera un error de compilación.
b) La colección permite cualquier tipo de elemento.
c) No se puede añadir ningún elemento.
d) Ninguna de las anteriores.

3. ¿Qué palabra clave se utiliza para restringir un tipo genérico a una clase base?

a) `implements`.
b) `super`.
c) `extends`.
d) Ninguna de las anteriores.

Conclusión del Bloque: Tiempo de ejecución

En este bloque, hemos explorado conceptos fundamentales relacionados con el comportamiento dinámico de los programas en Java. Estos principios permiten manejar la ejecución de programas de manera más flexible y escalable, aprovechando características avanzadas del lenguaje como el **polimorfismo**, el **casting**, y el uso de **genéricos**.

1. Puntos Clave del Bloque

1. Ligadura Dinámica

- La ligadura dinámica permite que el sistema determine qué método ejecutar en tiempo de ejecución, basándose en el tipo real del objeto y no en su referencia.
- Este mecanismo es crucial para implementar el **polimorfismo**, facilitando sistemas extensibles y modulares.

2. Comprobación Estática y Dinámica de Tipos

- La **comprobación estática de tipos** asegura que los errores de tipo se detecten durante la compilación, mejorando la seguridad del código.
- La **comprobación dinámica de tipos** permite verificar y manejar tipos en tiempo de ejecución, utilizando herramientas como `instanceof` para evitar errores como `ClassCastException`.

3. Conversiones de Tipos entre Objetos (Casting)

- **Casting implícito (upcasting)** permite asignar objetos de subclases a referencias de superclases sin necesidad de una conversión explícita.
- **Casting explícito (downcasting)** requiere verificar el tipo en tiempo de ejecución y realizar la conversión manualmente, siendo una herramienta poderosa pero potencialmente riesgosa.

4. Clases y Tipos Genéricos

- Los **genéricos** permiten crear clases, métodos y colecciones parametrizados con tipos específicos, mejorando la reutilización y la seguridad del código.
- Los **comodines (`?`)** y las restricciones (`extends`, `super`) proporcionan flexibilidad adicional para manejar relaciones entre tipos en estructuras genéricas.

2. Ventajas del Conocimiento Adquirido

1. Flexibilidad en el Diseño de Sistemas:

- La ligadura dinámica y el polimorfismo permiten manejar objetos de diferentes tipos de manera uniforme.

2. Seguridad de Tipos:

- La combinación de comprobación estática y dinámica asegura que los programas sean robustos y menos propensos a errores en tiempo de ejecución.

3. Extensibilidad:

- Genéricos y casting permiten diseñar sistemas que puedan manejar diferentes tipos de datos sin reescribir el código.

4. Eficiencia:

- Las herramientas avanzadas como `instanceof` y el uso de genéricos reducen la necesidad de conversiones manuales y minimizan errores comunes.

3. Ejercicio Final del Bloque

Problema

Diseña un sistema para gestionar una tienda utilizando los conceptos aprendidos:

1. Define una clase base `Producto` con atributos:

- `nombre` (String) y `precio` (double).
- Método `mostrarInformacion()`.

2. Crea dos subclases:

- `ProductoElectronico` con un atributo adicional `marca` y un método `mostrarGarantia()`.
- `ProductoAlimenticio` con un atributo adicional `fechaCaducidad` y un método `verificarFrescura()`.

3. Crea una lista genérica de productos y utiliza `instanceof` para realizar operaciones específicas según el tipo de producto.

Código Propuesto

```
import java.util.ArrayList;
public class Producto {
    protected String nombre;
    protected double precio;
    public Producto(String nombre, double precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    public void mostrarInformacion() {
        System.out.println("Producto: " + nombre + ", Precio: " + precio);
    }
}
public class ProductoElectronico extends Producto {
    private String marca;
    public ProductoElectronico(String nombre, double precio, String marca) {
        super(nombre, precio);
        this.marca = marca;
    }
    public void mostrarGarantia() {
        System.out.println("La garantía de " + nombre + " es de 2 años.");
    }
}
public class ProductoAlimenticio extends Producto {
    private String fechaCaducidad;
    public ProductoAlimenticio(String nombre, double precio, String fechaCa-
ducidad) {
        super(nombre, precio);
        this.fechaCaducidad = fechaCaducidad;
    }
    public void verificarFrescura() {
        System.out.println("Verificando frescura de " + nombre + "...");
    }
}
public class Main {
    public static void main(String[] args) {
        ArrayList<Producto> productos = new ArrayList<>();
        productos.add(new ProductoElectronico("Televisor", 499.99, "Sam-
sung"));
        productos.add(new ProductoAlimenticio("Leche", 1.99, "2024-12-31"));
        for (Producto producto : productos) {
            producto.mostrarInformacion();
```

```
                if (producto instanceof ProductoElectronico) {
                    ((ProductoElectronico) producto).mostrarGarantia();
                } else if (producto instanceof ProductoAlimenticio) {
                    ((ProductoAlimenticio) producto).verificarFrescura();
                }
                System.out.println("---");
            }
        }
    }
```

Salida:

```
Producto: Televisor, Precio: 499.99
La garantía de Televisor es de 2 años.
---
Producto: Leche, Precio: 1.99
Verificando frescura de Leche...
---
```

4. Preguntas de Evaluación

1. ¿Qué permite la ligadura dinámica?

a) Ejecutar métodos en tiempo de compilación.
b) Determinar el método a ejecutar en tiempo de ejecución.
c) Garantizar el tipo de un objeto en tiempo de compilación.
d) Convertir objetos a diferentes tipos.

2. ¿Qué ocurre si se realiza un downcasting sin verificar el tipo con `instanceof`?

a) El programa lanza un error de compilación.
b) El programa lanza una excepción en tiempo de ejecución.
c) El programa continúa, pero el comportamiento es indefinido.
d) No ocurre nada.

3. ¿Cuál es la principal ventaja de usar genéricos en Java?

a) Permitir que las colecciones almacenen diferentes tipos de datos sin restricciones.
b) Mejorar la seguridad de tipos en tiempo de compilación.
c) Aumentar la velocidad del programa.
d) Eliminar la necesidad de conversiones de tipo.