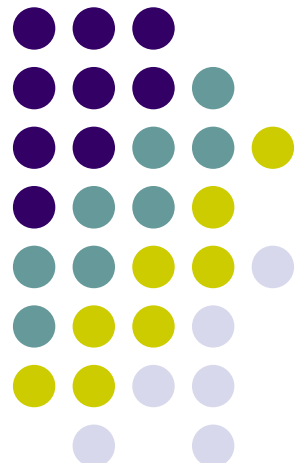
Tema 2

Técnicas básicas de POO



Contenido

- Principios de POO
  - Encapsulación
  - Herencia
  - Polimorfismo
  - Ejemplos
- Técnicas básicas
  - Delegación
  - Uso de interfaces
  - Interfaces de marcado

Principios de POO
Encapsulación

- Abstracción
  - Espacio de la máquina vs. Espacio del problema
    - Difícil escritura de los programas
    - Programas costosos de mantener
  - Descripción del problema en términos del problema
  - Clases y objetos

- Ocultación de la información
  - La interfaz define lo que se le puede pedir a un objeto
  - Diferenciar el qué del cómo
  - Especificadores de visibilidad: `public`, `private`, `protected`

Principios de POO
Herencia

- Reutilización de propiedades y operaciones
  - Clase base y derivada son del mismo tipo
  - Todos los mensajes que pueden enviarse a la clase base también pueden enviarse a la clase derivada
- Redefinición vs. Reutilización del comportamiento
  - *Overriding* (superposición): cambio de comportamiento
  - *Overloading* (sobrecarga): cambio de interfaz
- Herencia pura vs. Extensión

  (aunque java usa `extends` para ambas)
  - Herencia: mantiene la interfaz tal cual (relación *es-un*)
  - Extensión: amplía la interfaz con nuevas funcionalidades (relación *es-como-un*)

Principios de POO
Composición vs. Herencia

- Composición
  - Sirve cuando hacen falta las características de una clase existente dentro de una nueva, pero no su interfaz
  - Los objetos miembro privados pueden cambiarse en tiempo de ejecución
  - Los cambios en el objeto miembro no afectan al código cliente

- Herencia
  - Sirve para hacer una versión especial de una clase existente, reutilizando su interfaz
  - La relación de herencia debe definirse en tiempo de compilación y no puede cambiarse en tiempo de ejecución
  - Permite reinterpretar el tipo de un objeto en tiempo de ejecución

Principios de POO
Polimorfismo

- **Polimorfismo**: fenómeno por el que, cuando se envía un mensaje a un objeto del que no se sabe su tipo concreto, se ejecuta el método adecuado de acuerdo con su tipo

- **Enlace dinámico**: el método a ejecutar se decide en tiempo de ejecución, en función de la clase del objeto.

  Es la implementación del polimorfismo.

- **"Moldes" de objetos**:
  - *Upcasting*: interpretar un objeto de una clase derivada como del mismo tipo que la clase base
  - *Downcasting*: interpretar un objeto de una clase base como del mismo tipo que una clase derivada suya

Principios de POO
Objetivo: reutilización, flexibilidad

- Facilidad para adaptarse a cambios
- ¿Cómo se consigue?
  - Datos encapsulados
  - Métodos polimórficos
  - Clases abstractas
  - Delegación
  - Interfaces
- Marco de trabajo (*framework*) vs. Biblioteca
- Patrones de diseño

Principios de POO

Ej1: clases abstractas - upcasting

Criticar la siguiente implementación:

```
abstract class Intrumento {
   public void tocar () {}
   public static void afinarInstrumento (Instrumento i) {
      // Afinar en función del tipo de i
      if (i instanceOf Viento)
         afinarViento(i);
      else if (i instanceOf Cuerda)
         afinarCuerda(i);
      i.tocar();
   }
   public static void afinarViento (Viento i) { ... }
   public static void afinarCuerda (Cuerda i) { ... }
}
class Viento extends Instrumento {
   public void tocar () { soplar(); }
}
```

Principios de POO

Ej1: clases abstractas - upcasting


```
class Cuerda extends Instrumento {
   public void tocar () { rascar (); }
}
public class Orquesta {
   ArrayList<Instrumento> instrumentos;
   public Orquesta { instrumentos = new ArrayList<Instrumento>(3); }
   public void tocar() {
      for (int i=0; i<instrumentos.size(); i++)
         instrumentos.get(i).tocar();
   }
   public static void main (String[] args) {
      instrumentos.add(new Viento());
      instrumentos.add(new Cuerda());
      for (int i=0; i<instrumentos.size(); i++)
         Instrumento.afinarInstrumento(instrumentos.get(i));
      tocar();
   }
}
```

Principios de POO

Ej1: clases abstractas – polimorfi.

Seguir criticando la implementación:

```
abstract class Intrumento {
   public void tocar () {}
   public void afinar () { }
}
// class Viento ...
// class Cuerda ...
class Percusion extends Instrumento {
   public void tocar() { golpear(); }
   public void afinar() { golpear(); golpear(); // y afinar... }
}
class Orquesta {
   ArrayList<Intrumento> instrumentos;
   public Orquesta { instrumentos = new ArrayList<Intrumento>(3); }
   public void tocar () {
      for (int i=0; i<instrumentos.size(); i++)
         instrumentos.get(i).tocar();
   }
```

Principios de POO

Ej1: clases abstractas – polimorfi.


```
   public void afinar (Instrumento i) {
      i.afinar();  // Método polimórfico
      i.tocar();   // Prueba de que está afinado
   }
}
public PruebaOrquesta {
   public static void main (String[] args) {
      Orquesta orquesta = new Orquesta();
      orquesta.instrumentos.add(new Viento());
      orquesta.instrumentos.add(new Cuerda());
      orquesta.instrumentos.add(new Percusion());
      for (int i=0; i<instrumentos.size(); i++)
         orquesta.afinar(instrumentos.get(i));
      orquesta.tocar();
   }
}
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación

Delegar altas/bajas de `Instrumento` en el agregado `Orquesta`:

```
class Orquesta {
   private ArrayList<Intrumento> instrumentos;
   public Orquesta { instrumentos = new ArrayList<Intrumento>(3);
   public boolean addInstrumento (Instrumento i) {
      return instrumentos.add(i);
   }
   public boolean removeInstrumento (Instrumento i) {
      return instrumentos.remove(i);
   }
   public void tocar() {
      for (int i=0; i<instrumentos.size(); i++)
         instrumentos.get(i).tocar();
   }
   public void afinar (Instrumento i) {
      i.afinar();
      i.tocar();  // Prueba de que está afinado
   }
}
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación


```
class PruebaOrquesta {
   public static void main (String[] args) {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento());
      orquesta.addInstrumento(new Cuerda());
      orquesta.addInstrumento(new Percusion());
      for (int i=0; i<orquesta.instrumentos.size(); i++)
         afinar(instrumentos.get(i));
      orquesta.tocar();
   }
}
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación, interfaces

Para recorrer la colección de instrumentos, aprovechar la interfaz de `Iterator`:

```
class Orquesta {
   private List<Intrumento> instrumentos;
   public Orquesta { instrumentos = new ArrayList<Intrumento>(3);
   public boolean addInstrumento (Instrumento i) {
      return instrumentos.add(i);
   }
   public boolean removeInstrumento (Instrumento i) {
      return instrumentos.remove(i);
   }
   public void tocar() {
      for (Iterator<Instrumento> i=instrumentos.iterator(); i.hasNext(); )
         i.next().tocar();
   }
   public void afinar (Instrumento i) {
      i.afinar();
      i.tocar();  // Prueba de que está afinado
   }
}
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación, interfaces

```
class PruebaOrquesta {
   public static void main (String[] args) {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento());
      orquesta.addInstrumento(new Cuerda());
      orquesta.addInstrumento(new Percusion());
      for (Iterator<Instrumento> i=orquesta.instrumentos.iterator();
           i.hasNext();
           )
         afinar(i.next());
      orquesta.tocar();
   }
}
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación, interfaces

2ª versión con JDK 1.5:

```
class Orquesta implements Iterable<Instrumento> {
   private List<Intrumento> instrumentos;
   public Orquesta { instrumentos = new ArrayList<Intrumento>(3);
   public boolean addInstrumento (Instrumento i) {
      return instrumentos.add(i);
   }
   public boolean removeInstrumento (Instrumento i) {
      return instrumentos.remove(i);
   }
   public Iterator<Instrumento> iterator() {
      return instrumentos.iterator();
   }
   public void tocar() {
      for (Instrumento i: this)
         i.tocar();
   }
```

Principios de POO

Ej1: clases abstractas, polimorfismo, delegación, interfaces

```
 public void afinar (Instrumento i) {
      i.afinar();
      i.tocar();  // Prueba de que está afinado
   }
}
class PruebaOrquesta {
   public static void main (String[] args) {
      Orquesta orquesta = new Orquesta();
      orquesta.addInstrumento(new Viento());
      orquesta.addInstrumento(new Cuerda());
      orquesta.addInstrumento(new Percusion());
      for (Instrumento i: orquesta)
         afinar(i);
      orquesta.tocar();
   }
}
```

Principios de POO

Ej2: downcasting


```
class Util {
   public void f () {}
   public void g () {}
}
class MasUtil extends Util {
   public void f () {}
   public void g () {}
   public void u () {}
   public void v () {}
}
public class RTTI {
   public static void main (String[] args) {
      Util[] x = {new Util(), new MasUtil()};
      x[0].f();
      x[1].g();
      x[1].u();            // Error de compilación, método no encontrado
      ((MasUtil)x[1]).u(); // Downcast
      ((MasUtil)x[0]).u(); // Error de ejecución, ClassCastException
   }
}
```

Principios de POO

Ej3: “herencia múltiple” en java


```
interface SabeLuchar { void luchar (); }
interface SabeNadar  { void nadar  (); }
interface SabeVolar  { void volar  (); }
public class PersonajeDeAccion {
   public void luchar {}
}
public class Heroe extends PersonajeDeAccion implements SabeLuchar, SabeNadar, SabeVolar {
   public void nadar () {}
   public void volar () {}
}
public class Aventura {
   static void t (SabeLuchar x) { x.luchar(); }
   static void u (SabeNadar  x) { x.nadar();  }
   static void v (SabeVolar  x) { x.volar();  }
   static void w (PersonajeDeAccion x) { x.luchar();  }
   public static void main (String[] args) {
      Heroe h = new Heroe();
      t(h); // Lo trata como un objeto SabeLuchar
      u(h); // Lo trata como un objeto SabeNadar
      v(h); // Lo trata como un objeto SabeVolar
      w(h); // Lo trata como un objeto PersonajeDeAccion
   }
}
```

Técnicas básicas

- **Delegación**: cuándo no usar herencia
- **Uso de interfaces**: usar interfaces o referencias a una clase abstracta base en el código cliente
- **Interfaces de marcado**: interfaces sin métodos para señalar atributos semánticos de una clase
- **Otros**: véase M. Grand, *Patterns in Java*, Volume I, capítulo 4

Técnicas básicas

Ej4: delegación


```
class Vuelo {
   CompartimentoEquipaje _luggage;
   void facturarEquipaje (Equipaje bulto) {
      _luggage.facturarEquipaje(bulto);
   }
}

class CompartimentoEquipaje {
   private Vector bultos;
   synchronized void facturarEquipaje (Equipaje bulto) {
      // ...
   }
}
```

Técnicas básicas

Ej5: interfaces


```
interface Figura {
   public void dibujar ();
}

class Circulo implements Figura { public void dibujar () {...} }
class Cuadrado implements Figura { public void dibujar () {...} }
class Triangulo implements Figura { public void dibujar () {...} }

public class PruebaFiguras {
   public static void main (String[] args) {
      ArrayList l = new ArrayList();
      l.add(new Circulo());
      l.add(new Cuadrado());
      l.add(new Triangulo());
      iterator it = l.iterator();
      while (it.hasNext())
         ((Figura)it.next()).dibujar();
   }
}
```

Técnicas básicas

Ej6: interfaces de marcado


```
public class ListaEnlazada implements Cloneable, java.io.Serializable {
   public ListaEnlazada buscar (Object target) {
      if (target==null || target instanceOf IgualdadPorIdentidad)
         return buscarIg(target);
      else
         return buscarIgual(target);
   }
   private synchronized ListaEnlazada buscarIg (Object target) { ... }
   private synchronized ListaEnlazada buscarIgual (Object target) { ... }
   // ...
}

public interface IgualdadPorIdentidad { }
```