Programación Orientada a Objetos

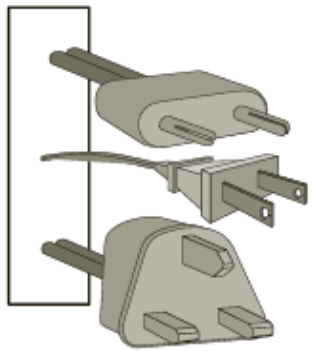
Algorítmica y Programación II

Terminología

- Cada **objeto** creado en un programa es una **instancia** de una **clase**.
- Cada clase presenta al mundo exterior una visión concisa y consistente de los objetos que son instancias de esta clase, sin entrar en demasiados detalles innecesarios ni dar acceso a otros al funcionamiento interno de los objetos.
- La definición de clase especifica típicamente **variables de instancia**, también conocidas como **miembros de datos**, que el objeto contiene, así como los **métodos**, también conocidas como **funciones miembro**, que los objetos pueden ejecutar.

Metas

- **Robustez**
- Queremos que el software sea capaz de manejar entradas inesperadas que no están explícitamente definidas para su aplicación.
- **Adaptabilidad**
- El software debe poder evolucionar con el tiempo en respuesta a las condiciones cambiantes de su entorno. (hardware, SO, etc.)
- **Reusabilidad**
- El mismo código debe ser utilizable como un componente de diferentes sistemas en diversas aplicaciones.

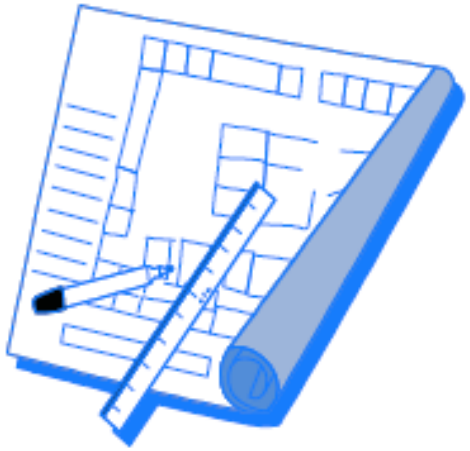


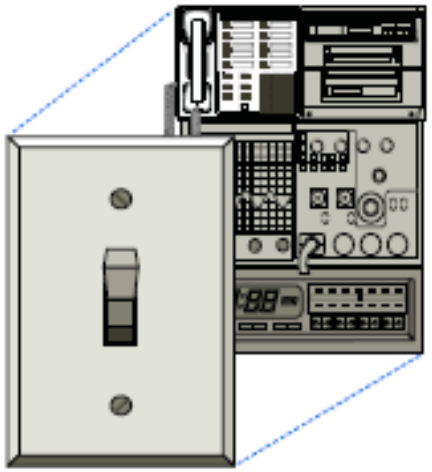
Tipos Abstractos de Datos

- **Abstracción** es representar un sistema por sus partes más fundamentales.
- La aplicación del paradigma de abstracción al diseño de estructuras de datos da lugar a **tipos de datos abstractos** (TAD) **abstract data types** (ADTs).
- Un TAD es un modelo de una estructuras de datos que especifica el **tipo** de dato almacenado, las **operaciones** soportadas por él, y los tipos de parámetros de las operaciones.
- Un TAD especifica qué operaciones hace, pero no como las hace.
- El conjunto colectivo de comportamientos respaldados por un TAD es su **interfaz pública**.

Principios de Diseño Orientado a Objetos

- Abstracción
- Encapsulamiento
- Modularidad





Patrones de Diseño

Patrones algorítmicos:

- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search
- Brute force
- Dynamic programming
- The greedy method

Patrones de diseño de software:

- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

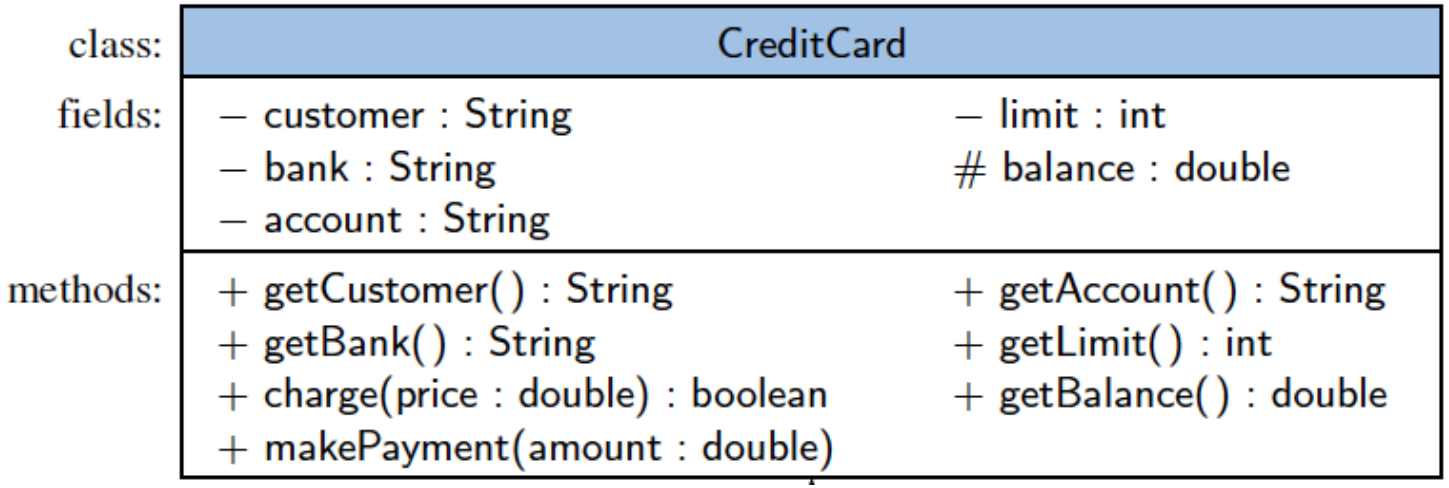
Diseño de Software Orientado a Objeto

- **Responsibilidades**: Divida el trabajo en diferentes actores, cada uno con una responsabilidad diferente.
- **Independencia**: Defina el trabajo para cada clase para ser tan independiente de otras clases como sea posible.
- **Comportamientos**: Defina los comportamientos de cada clase con cuidado y precisión, de modo que las consecuencias de cada acción realizada por una clase sean bien comprendidas por otras clases que interactúen con ella.

Lenguaje de modelado unificado/Unified Modeling Language (UML)

- Un **diagrama de clase** tiene tres porciones.

1. El nombre de la clase
2. Las variables de instancia recomendadas
3. Los métodos de clases recomendados.



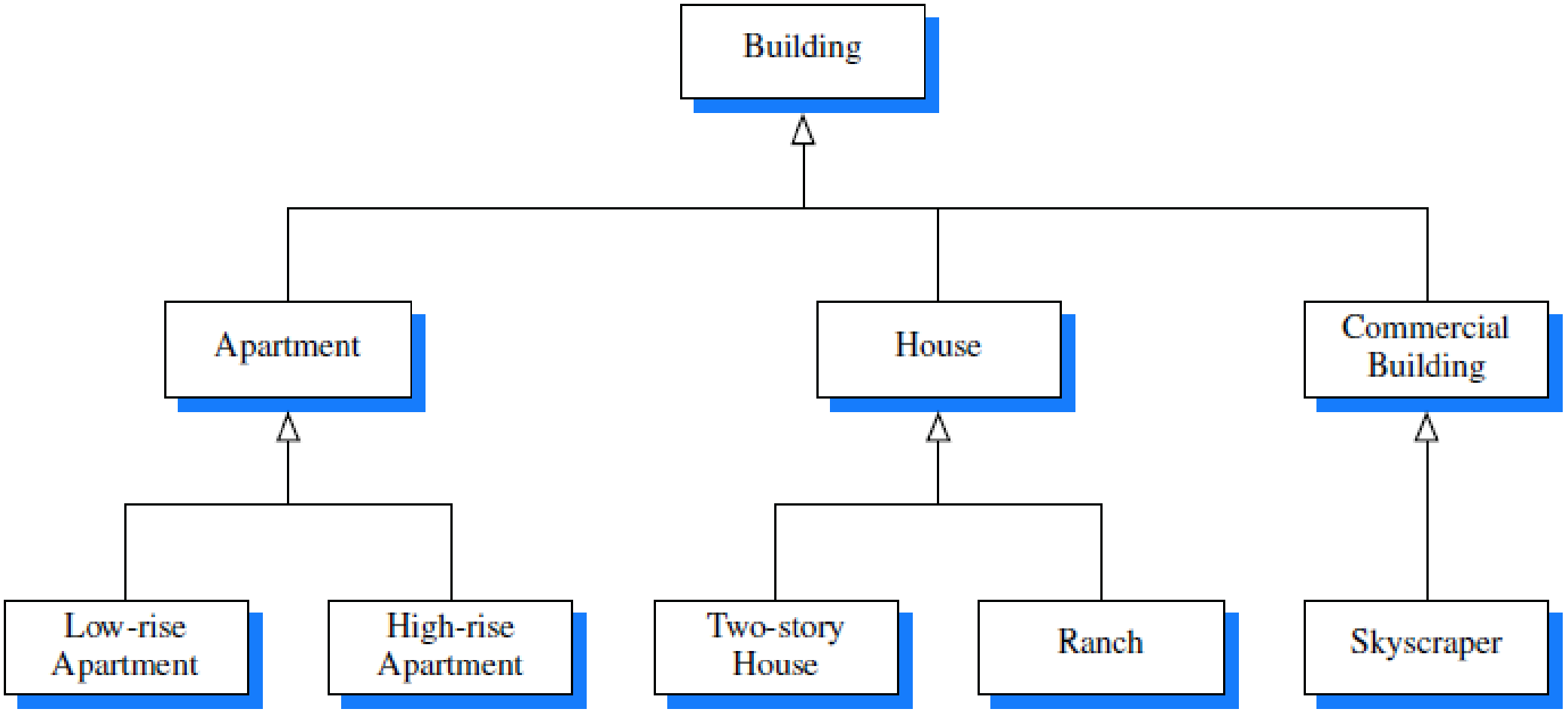
Definiciones de Clases

- Una clase sirve como el medio principal para la abstracción en la programación orientada a objetos.
- En Java, cada variable es un tipo base (primitivo) o es una referencia a una instancia de alguna clase.
- Una clase proporciona un conjunto de comportamientos en forma de funciones miembro (también conocidas como **métodos**), con implementaciones que pertenecen a todas sus instancias.
- Una clase también sirve como modelo para sus instancias, determinando de manera efectiva la forma en que se representa la información de estado para cada instancia en forma de atributos (también conocidos como campos, variables de instancia o miembros de datos).

Constructores

- Un usuario puede crear una instancia de una clase utilizando el operador ***new*** con un método que tiene el mismo nombre que la clase.
- Tal método, conocido como **constructor**, tiene como responsabilidad establecer el estado de un nuevo objeto con valores iniciales apropiados para sus variables de instancia.

Herencia


Building
Apartment
House
Commercial Building
Low-rise Apartment
High-rise Apartment
Two-story House
Ranch
Skyscraper

Herencia y Constructores

- Los constructores nunca se heredan en Java; por lo tanto, cada clase debe definir un constructor por sí mismo.
- Todos sus variables de instancia deben estar inicializadas correctamente, incluidos los campos heredados.
- La primera operación dentro del cuerpo de un constructor debe ser invocar un constructor de la superclase, que inicializa los campos definidos en la superclase.
- Un constructor de la superclase se invoca explícitamente mediante el uso de la palabra clave **super** con los parámetros adecuados.
- Si un constructor para una subclase no realiza una llamada explícita a **super** o **this** como su primer comando, entonces se realizará una llamada implícita a **super (),** la versión de cero parámetros del constructor de la superclase.

Un ejemplo de extensión (herencia)

- Una **progresión numérica** es una secuencia de números, donde cada número depende de uno o más de los números anteriores.
- Una **progresión aritmética** determina el siguiente número *agregando* una constante fija al valor anterior.
- Una **progresión geométrica** determina el próximo número *multiplicando* el valor anterior por una constante fija.
- Una **progresión de Fibonacci** utiliza la *fórmula Ni+1=Ni+Ni-1*

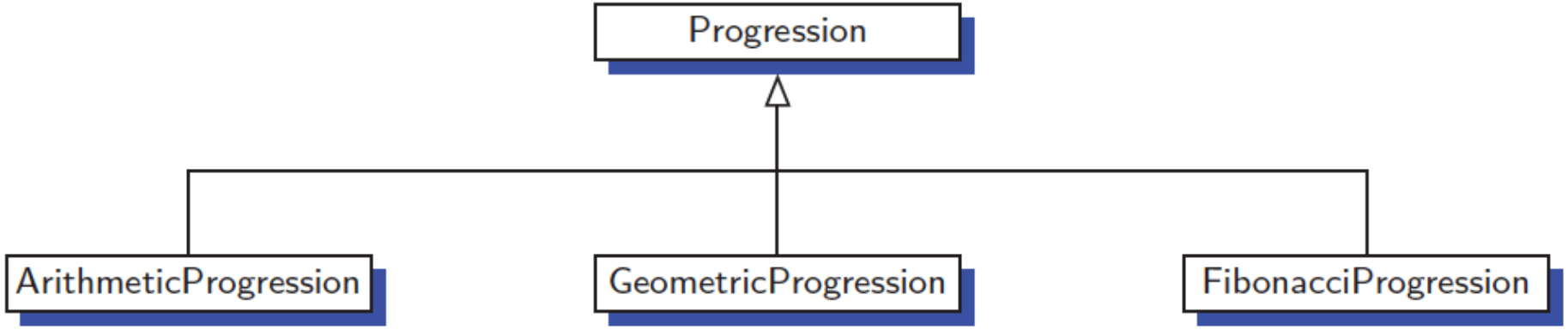
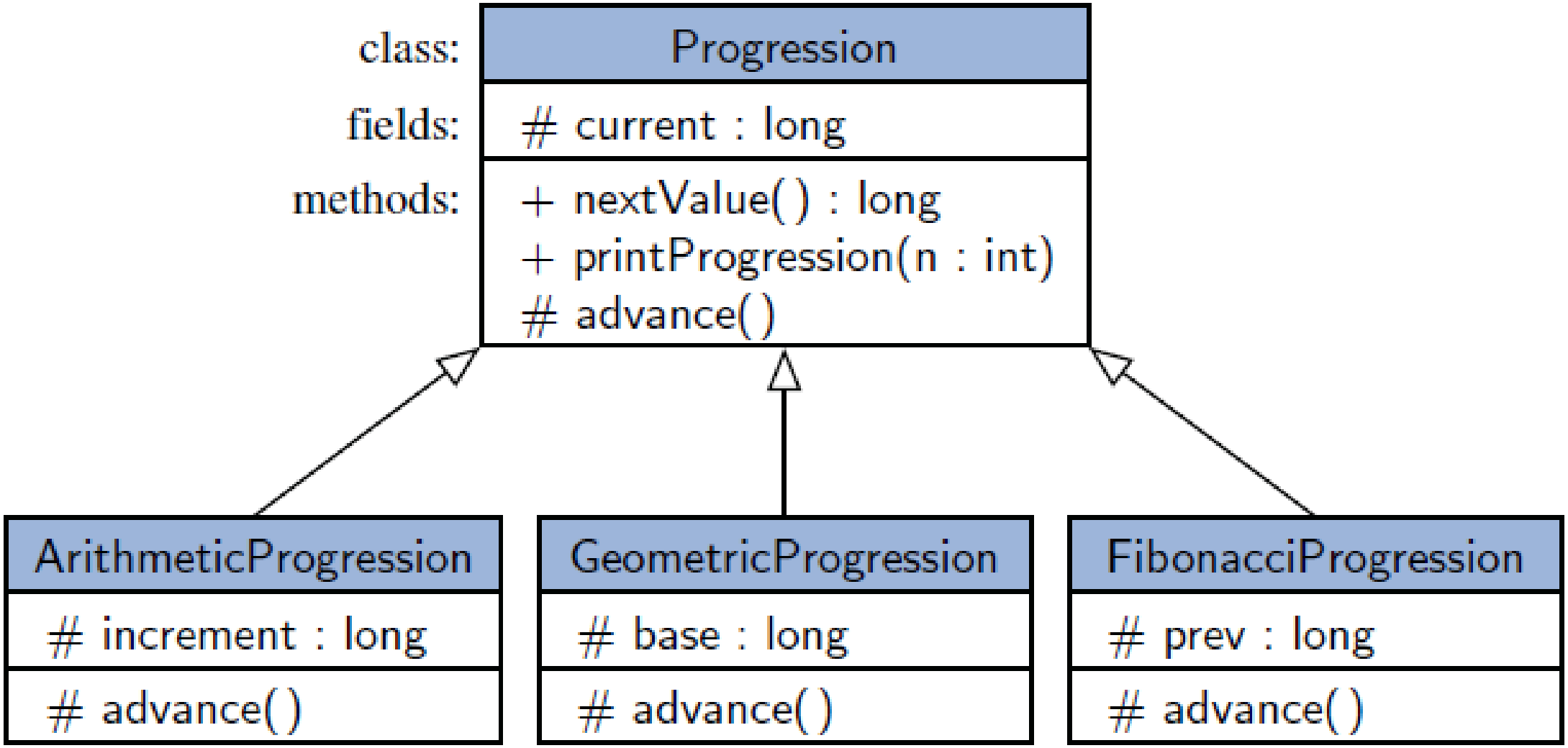


Diagrama de Clases de Progression y sus Subclases



La clase base Progression

```
1  /** Generates a simple progression. By default: 0, 1, 2, ... */
2  public class Progression {
3
4    // instance variable
5    protected long current;
6
7    /** Constructs a progression starting at zero. */
8    public Progression( ) { this(0); }
9
10   /** Constructs a progression with given start value. */
11   public Progression(long start) { current = start; }
12
13   /** Returns the next value of the progression. */
14   public long nextValue( ) {
15     long answer = current;
16     advance( );      // this protected call is responsible for advancing the current value
17     return answer;
18   }
```

La clase base, *continuación*

```
19
20    /** Advances the current value to the next value of the progression. */
21    protected void advance( ) {
22      current++;
23    }
24
25    /** Prints the next n values of the progression, separated by spaces. */
26    public void printProgression(int n) {
27      System.out.print(nextValue( ));              // print first value without leading space
28      for (int j=1; j < n; j++)
29        System.out.print(" " + nextValue( ));  // print leading space before others
30      System.out.println( );                       // end the line
31    }
32  }
```

Subclase ArithmeticProgression

```
1  public class ArithmeticProgression extends Progression {
2
3    protected long increment;
4
5    /** Constructs progression 0, 1, 2, ... */
6    public ArithmeticProgression( ) { this(1, 0); }       // start at 0 with increment of 1
7
8    /** Constructs progression 0, stepsize, 2*stepsize, ... */
9    public ArithmeticProgression(long stepsize) { this(stepsize, 0); }       // start at 0
10
11   /** Constructs arithmetic progression with arbitrary start and increment. */
12   public ArithmeticProgression(long stepsize, long start) {
13     super(start);
14     increment = stepsize;
15   }
16
17   /** Adds the arithmetic increment to the current value. */
18   protected void advance( ) {
19     current += increment;
20   }
21 }
```

Subclase GeometricProgression

```
1  public class GeometricProgression extends Progression {
2
3    protected long base;
4
5    /** Constructs progression 1, 2, 4, 8, 16, ... */
6    public GeometricProgression( ) { this(2, 1); }              // start at 1 with base of 2
7
8    /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
9    public GeometricProgression(long b) { this(b, 1); }                    // start at 1
10
11   /** Constructs geometric progression with arbitrary base and start. */
12   public GeometricProgression(long b, long start) {
13     super(start);
14     base = b;
15   }
16
17   /** Multiplies the current value by the geometric base. */
18   protected void advance( ) {
19     current *= base;                          // multiply current by the geometric base
20   }
21 }
```

Subclase FibonacciProgression

```
1  public class FibonacciProgression extends Progression {
2
3    protected long prev;
4
5    /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
6    public FibonacciProgression( ) { this(0, 1); }
7
8    /** Constructs generalized Fibonacci, with give first and second values. */
9    public FibonacciProgression(long first, long second) {
10     super(first);
11     prev = second − first;          // fictitious value preceding the first
12   }
13
14   /** Replaces (prev,current) with (current, current+prev). */
15   protected void advance( ) {
16     long temp = prev;
17     prev = current;
18     current += temp;
19   }
20 }
```

Interfaces

```
1   /** Interface for objects that can be sold. */
2   public interface Sellable {
3
4     /** Returns a description of the object. */
5     public String description( );
6
7     /** Returns the list price in cents. */
8     public int listPrice( );
9
10    /** Returns the lowest price in cents we will accept. */
11    public int lowestPrice( );
12  }
```

Interfaces (cont.)

```
1  /** Class for photographs that can be sold. */
2  public class Photograph implements Sellable {
3    private String descript;                          // description of this photo
4    private int price;                                // the price we are setting
5    private boolean color;                            // true if photo is in color
6
7    public Photograph(String desc, int p, boolean c) {    // constructor
8      descript = desc;
9      price = p;
10     color = c;
11   }
12
13   public String description( ) { return descript; }
14   public int listPrice( ) { return price; }
15   public int lowestPrice( ) { return price/2; }
16   public boolean isColor( ) { return color; }
17 }
```

Implementación Múltiple

```
1  /** Interface for objects that can be transported. */
2  public interface Transportable {
3    /** Returns the weight in grams. */
4    public int weight( );
5    /** Returns whether the object is hazardous. */
6    public boolean isHazardous( );
7  }
```

Implementación Múltiple

```
1  /** Class for objects that can be sold, packed, and shipped. */
2  public class BoxedItem implements Sellable, Transportable {
3    private String descript;          // description of this item
4    private int price;                // list price in cents
5    private int weight;               // weight in grams
6    private boolean haz;              // true if object is hazardous
7    private int height=0;             // box height in centimeters
8    private int width=0;              // box width in centimeters
9    private int depth=0;              // box depth in centimeters
10   /** Constructor */
11   public BoxedItem(String desc, int p, int w, boolean h) {
12     descript = desc;
13     price = p;
14     weight = w;
15     haz = h;
16   }
17   public String description( ) { return descript; }
18   public int listPrice( ) { return price; }
```

Clases Abstractas

```
1  public abstract class AbstractProgression {
2    protected long current;
3    public AbstractProgression( ) { this(0); }
4    public AbstractProgression(long start) { current = start; }
5
6    public long nextValue( ) {                    // this is a concrete method
7      long answer = current;
8      advance( );     // this protected call is responsible for advancing the current value
9      return answer;
10   }
11
12   public void printProgression(int n) {         // this is a concrete method
13     System.out.print(nextValue( ));             // print first value without leading space
14     for (int j=1; j < n; j++)
15       System.out.print(" " + nextValue( ));     // print leading space before others
16     System.out.println( );                      // end the line
17   }
18
19   protected abstract void advance( );           // notice the lack of a method body
20 }
```

Actividad 1

- Implementar la herencia de progresiones utilizando paquetes
- Reemplazar la clase base por una clase abstracta
- Implementar un arreglo polimórfico
- Testear ambos paquetes utilizando paquete de prueba

Excepciones

- Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa.
- Puede producirse una excepción debido a un recurso no disponible, una entrada inesperada de un usuario o simplemente un error lógico por parte del programador.
- En Java, las excepciones son objetos que puede **lanzar (thrown)** el código que se encuentra con una situación inesperada.
- Una excepción también puede ser **capturada** (**caught)** por un bloque de código que "maneja" el problema.
- Si no se detecta, una excepción hace que la máquina virtual deje de ejecutar el programa y que informe un mensaje apropiado a la consola.

Atrapando Excepciones

- La metodología general para manejar excepciones es una construcción **try-catch** en la que se ejecuta un fragmento de código protegido que podría arrojar una excepción.
- Si **arroja** (**throws)** una excepción, esa excepción se captura haciendo que el flujo de control salte a un bloque de captura predefinido que contiene el código para aplicar una resolución apropiada.
- Si no ocurre una excepción en el código guardado, todos los bloques **catch** se ignoran.

```
try {
    guardedBody
} catch (exceptionType1  variable1) {
    remedyBody1
} catch (exceptionType2  variable2) {
    remedyBody2
} ...
```

Lanzando excepciones

- Las excepciones se originan cuando una parte del código de Java encuentra algún tipo de problema durante la ejecución y arroja un objeto de excepción.
- Esto se hace usando la palabra clave **throw** seguida de una instancia del tipo de excepción que se lanzará.
- A menudo es conveniente crear una instancia de un objeto de excepción en el momento en que se debe lanzar la excepción. Por lo tanto, una instrucción throw se escribe típicamente de la siguiente manera: **throw new** *exceptionType (parameters);* donde *exceptionType* es el tipo de la excepción y los parámetros se envían al constructor de ese tipo.

La cláusula throws

- Cuando se declara un método, es posible declarar explícitamente, como parte de su firma, la posibilidad de que se genere un tipo de excepción particular durante una llamada a ese método.
- La sintaxis para declarar posibles excepciones en una firma de método se basa en palabra clave **throws** (que no debe confundirse con una declaración de lanzamiento real **throw**).
- Por ejemplo, el método *parseInt* de la clase *Integer* tiene la siguiente firma formal:

public static int parseInt(String s) ***throws*** *NumberFormatException;*

Casting

- Casting con Objects permite la conversión entre clases y subclases.
- Se produce una **conversión de ampliación** cuando un tipo T se convierte en un tipo "más ancho" U :
- T y U son tipos de clase y U es una superclase de T.
- T y U son tipos de interfaz y U es una superinterfaz de T.
- T es una clase que implementa la interfaz U.
- Ejemplo:

Progression prog= new AritmeticProgression(...);

Conversiones de reducción/Narrowing Conversions

- Una **conversion de reducción** ocurre cuando un tipo T se convierte a un tipo “reducido” de tipo S.
- T y S son tipos de clase y S es una subclase de T.
- T y S son tipos de interfaz y S es una subinterfaz de T.
- T es una interfaz implementada por la clase S.
- En general, una conversión de reducción de tipos de referencia requiere un lanzamiento explícito.
- Ejemplo:

AritmeticProgression ap = *(AritmenticProgression) prog;*

Actividad 2

Dibuje un diagrama de clases de herencia para el siguiente conjunto de clases:

- Clase **Cabra** extiende Object y agrega una variable de instancia de ***cola*** y métodos: ***leche()*** y ***saltar()***.
- Clase **Pig** extiende Object y agrega una variable de instancia de ***nariz*** y métodos ***comer(comida)*** y ***revolcar()***.
- Clase **Caballo** extiende Object y agrega variables de instancia de ***altura*** y ***color***, y los métodos ***correr()*** y ***saltar()***.
- Clase **Carrera** extiende **Caballo** y agrega un método ***carrera()***.
- La clase **Ecuestre** extiende el **Caballo** y agrega la variable de instancia ***peso*** y ***esEntrenado***, y los métodos ***trotar()*** y ***isEntrenado()***.

Considere la herencia de las clases anterior, y sea ***d*** una variable de objeto de tipo **Caballo**. Si ***d*** se refiere a un objeto real de tipo **Ecuestre**, ¿se puede convertir a la clase **Carrera**? ¿Por qué sí o por qué no?

Genéricos

- Java incluye soporte para escribir clases genéricas y métodos que pueden operar en una variedad de tipos de datos, mientras que a menudo se evita la necesidad de conversiones explícitas.
- El marco (framework) de los genéricos nos permite definir una clase en términos de un conjunto de parámetros de tipo formales, que luego se pueden usar como el tipo declarado para variables, parámetros y valores de retorno dentro de la definición de la clase.
- Esos parámetros de tipo formales se especifican posteriormente cuando se utiliza la clase genérica como un tipo en otro lugar en un programa.

Sintáxis para Genéricos

- Los tipos pueden declararse utilizando nombres genéricos :

```
1  public class Pair<A,B> {
2    A first;
3    B second;
4    public Pair(A a, B b) {                    // constructor
5      first = a;
6      second = b;
7    }
8    public A getFirst( ) { return first; }
9    public B getSecond( ) { return second;}
10 }
```

- A continuación, se crean instancias usando tipos reales :

Pair <String,Double> bid;

Clases Anidadas

- Java permite anidar una definición de clase dentro de la definición de otra clase.
- El uso principal para la anidación de clases es cuando se define una clase que está fuertemente afiliada con otra clase.
- Esto puede ayudar a aumentar el encapsulamiento y reducir los conflictos de nombres no deseados.
- Las clases anidadas son una técnica valiosa cuando se implementan estructuras de datos, ya que una instancia de un uso anidado se puede usar para representar una pequeña porción de una estructura de datos más grande, o una clase auxiliar que ayuda a navegar por una estructura de datos primaria.

Bibliografía

- 2014 Goodrich, Tamassia, Goldwasser Object-Oriented Programming