Programación Orientada a Objetos

Francisco Hidrobo / Kay Tucci / Mayerlin Uzcátegui

Universidad de Los Andes
Facultad de Ciencias. SUMA
Mérida, 5101. Venezuela

hidrobo, kay, maye@ula.ve

Programa del Curso

- Programación Orientada a Objetos (OO)
- Implementación de la Programación OO
- Instalación y Compilación del Software
- Componentes Básicos de la Tecnología OO
- Diseño de Clases OO
- Manipulación y Optimización de Código OO
- Diseño de Interfaces Gráficas de Usuario
- Manejo de Entrada/Salida y Redes

Material del Curso

Bibliografía

- Cualquier libro de OO
- http://www.javasoft.com
- http://www.python.org
- http://gcc.gnu.org

Repositorio de Clases

- http://sai.ula.ve
- http://webdelprofesor.ula.ve

Recomendaciones Generales

1. Programación orientada a objetos

Al finalizar esta clase el estudiante deberá

- Comprender la abstracción de clases
- Identificar objetos
- Comprender las ventajas del encapsulamiento
- Comprender la relación entre objetos y clases
- Comprender la herencia
- Comprender las subclases, su especialización y el polimorfismo
- Comprender las clases abstractas

1.1. Paradigma OO

El análisis y diseño orientado a objetos, u OO, es un paradigma de programación relativamente nuevo que permite:

- Tratar la complejidad mediante la división del programa en módulos manejables
- Reducir la complejidad mediante la abstracción
- Soportar los nuevos desarrollos con los desarrollos previos
- Reducir el tiempo y esfuerzo de pruebas.

1.2. Orientación a Objetos

Es una técnica de modelado de sistemas. Los sistemas pueden ser sistemas de software o sistemas en un contexto más general. Con esta técnica se describe, o modela, un sistema mediante un conjunto de objetos que interactúan entre sí.

Ejemplo:

- Un computador
- Una Facultad

1.3. Abstración

Es el proceso de capturar detalles fundamentales de las cosas mientras que se suprimen o ignoran los detalles inecesarios.

Ejemplo:

Si vemos un halcón y una paloma, ambos tienen alas, plumas, dos patas, un pico, etc.; y podemos hacer con ambos la abstracción de un ave. Sin embargo el halcón y la paloma tienen características diferentes como la forma del pico, las garras, el peso, etc.

1.3.1. Tipos de abstracción

Abstracción funcional

- Oculta detalles del prosesamiento
- Usa un lenguaje de alto nivel

Abstracción de datos

- Oculta detalles de los datos
- Usa un formato general de almacenamiento

1.4. Objeto

Representación abstracta de una cosa

Los objetos pueden representar cosas tangibles y cosas intangibles y tienen las siguientes características:

- Son cosas
- Pueden ser simples o complejos
- Pueden ser reales o imaginarios

1.5. Identificación de objetos

Al describir un sistema por lo general los objetos se pueden relacionar con los sustantivos usados en la descripción.
Pero hay que tener en cuenta que hay sinónomos, que hay sustantivos que expresan acciones, eventos y atributos que no siempre pueden ser interpretados como objetos.
A los sustantivos los pueden acompañar los adjetivos que cambian o complementan su significado.

1.6. Componentes de un objeto

Los objetos están compuestos por

Atributos: Características del objetos

Métodos: Acciones que el objeto puede hacer o que se pueden hacer con él

Los atributos y los métodos que componen un objeto depende de la abstracción que se haga del mismo y por lo tanto dependen de problema que se esté atacando.

1.7. Reconocer objetos

No todos los posibles objetos identificados son relevantes en un problema. Para identificar cuáles son los que pertenecen al problema hay que responder si el objeto identificado

- ¿Está dentro del dominio del problema?
- ¿Es necesario para el sistema?
- ¿Se requiere como parte de la interacción del sistema con los usuarios?
- ¿Es independiente de otros objetos?
- ¿Tiene atributos y métodos?

1.8. Encapsulamiento

Es el proceso de ocultar los detalles de funcionamiento de un objeto que no son esenciales para su uso.

- Interface pública
- Implementación
- Información interna

La ventaja del encapsulamiento es que facilita la evolución de los objetos sin afectar la forma de cómo usarlos.

1.9. Miembros de un objeto

Son todos los atributos y métodos de él.
Los miembros pueden ser:

Publicos: que componen la interfaz

Privados: que son parte de la implementación

En términos estrictos de OO, todos los atributos deben ser privados y de ser necesario, deben manipularse mediante métodos `set` y `get`

1.10. Clase

Es una generalización de un conjunto de objetos del mismo tipo.
Clase es una plantilla que permite describir el estado y el comportamiento de los objetos que instancian a la clase.
Clase es un bloque de construcción de la programación OO.

Ejemplo:

automóvil, mamífero, persona

Objetos son instancias de clases.

1.11. Generalización

Es el proceso mediante el cual se identifican y definen los atributos y métodos comunes de un conjunto de objetos.
La generalización permite obtener las clases principales de un sistema, evita que haya redundancia en el desarrollo de la solución y de esta forma promueve la reutilización.

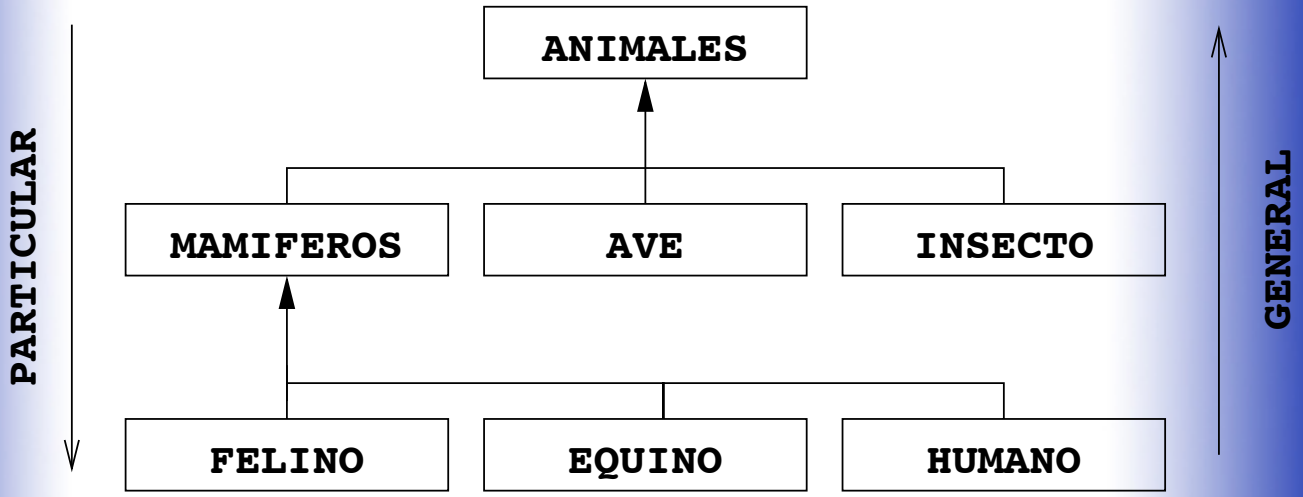
1.12. Herencia

Es la forma de definir una nueva clase en términos de otra clase ya existente.

- Permite agrupar clases relacionadas
- Promueve la reutilización

De una superclase, por ejemplo *animal*, pueden heredar varias subclases como: *mamífero*, *insecto*, *ave*

1.13. Jerarquía de Clases



1.14. Anulación / Sustitución de Métodos

Es el mecanismo mediante el cual una subclase modifica la implementación de un método que ha heredado de una superclase.

Ejemplo:

Para calcular el perímetro de una figura geométrica se implementa el método `perimetro()` que suma las longitudes de los lados. Pero para el caso de la subclase `circulo` esta implementación no funciona, por lo tanto hay que anularla y sustituirla

1.15. Especialización

Crea clases cada vez más específicas. Es cuando se agregan atributos o métodos, o se sustituyen métodos en el mecanismo de herencia para crear nuevas clases que puedan resolver un problema específico.

- Tener más atributos y métodos
- Particularizar los métodos

1.16. Polimorfismo

Del griego *polymorphos* que significa muchas formas. Es la habilidad que tienen los lenguajes o los sistemas de soportar que objetos de diferente tipo puedan recibir una misma instrucción para realizar una acción semánticamente similar.
El polimorfismo sólo se aplica a aquellos métodos que se heredan de una superclase común.

"Una interfaz, múltiples métodos" [a]

[a] Naugthon y Schildt, *Java 2, The Complete Reference*, McGraw-Hill, (1999).

1.17. Abstracción

Clase Abstracta: es aquella clase en la que al menos uno de sus métodos es abstracto.

Método Abstracto: es un método que se ha declarado pero no se ha implementado (escrito su código) debido a que la clase a la que pertenece es tan general que no es posible su implementación.

Las subclases de una superclase abstracta deben implementar todos los métodos abstractos o serán también clases abstractas.

1.18. Clases Abstractas

- Una clase abstracta tiene al menos un método abstracto, es decir un método declarado pero sin implementación.
- Las clases abstractas se utilizan como formatos o plantillas para la creación de las subclases, pero no pueden ser instanciadas por objetos.
- Las subclases que heredan de una clase abstracta deben implementar todos los métodos abstractos para ser instanciadas.

2. Implementación de POO

Al finalizar esta clase el estudiante deberá

- Comprender el código de declaración de clases
- Declarar variables de tipos primitivos y de clases
- Usar las clases abstractas y la herencia
- Usar la declaración `import` para incluir clases
- Usar la declaración `package` para agrupar clases
- Definir el comportamiento de una clase
- Explicar la sobrecarga de métodos
- Usar un constructor para instanciar a un objeto

2.1. Variables

Variable es el término que se usa en Java para definir un atributo de una clase, es decir, una variable es de una clase.

Una variable está compuesta por

- Un identificador
- Un tipo de dato
  - Primitivo
  - Referencia
- Un valor

2.2. Identificadores

Identificadores son los nombres que se le asignan a las clases, variables, métodos, etc.

Los caracteres válidos son (A-Z), (a-z), _, $, (0-9). No pueden comenzar con dígitos.
Por convención,

- Clase: primer caracter mayúscula
- Variable/Método: primer caracter minúscula
- Las siguientes palabras comienzan en mayúscula `notaAlumno`, `segundoNombre`

2.3. Tipo de datos primitivos

Tipo	Dominio	Tamaño	Rango
`byte`	$\mathcal{Z}$	1	$\pm$
`short`	$\mathcal{Z}$	2	$\pm$
`int`	$\mathcal{Z}$	4	$\pm$
`long`	$\mathcal{Z}$	8	$\pm$
`float`	$\Re$	4	$\pm$
`double`	$\Re$	8	$\pm$
`boolean`	$\mathcal{L}$	1	`true/false`
`char`	$\mathcal{C}$	2	Unicode

2.4. Variables Primitivas

Sintaxis:

```
tipo identificador;
tipo identificador = valor;
```

Ejemplo:

```
int miEntero;
int miEntero = 24;
double miDoble = 3.6;
double miDoble = 3.6E-3;
float miFlotante = 3.6f;
char miCaracter = 'k';
boolean miLogico = false;
```

2.5. Objetos

Objetos son instancias de una clase y ocupan porción de memoria en la que se almacenan los atributos y los métodos de una entidad del sistema.

Sintaxis:

```
Clase IdObjeto;
Clase IdObjeto = new Clase();
```

2.6. Asignación de Variables

Sintaxis:

```
int x = 1;
int y = x;
Persona alumno = new Persona();
Persona nuevoAlumno = alumno;
```

- ¿Qué le pasa a `y` si modificamos a `x`?
- ¿Qué le pasa a `nuevoAlumno` si modificamos a `alumno`?

Los objetos no se copian, se referencian.

2.7. Cadenas de Caracteres

Cadena de caracteres `String` es una clase y no un tipo primitivo de datos.

Sintaxis:

```
String nombre;
String saludo = "Hola";
String nombre = new String("Hola");
```

2.8. Declaración de las Clases

Sintaxis:

```
[⟨modificador⟩] class identificador { cuerpo }
```

Ejemplo:

```
class Alumno {}
abstract class Persona {}
public class Materia {}
final class Profesor {}
```

2.9. Al Fin...Hola Mundo!

Ejemplo:

```
//
// Ejemplo HolaMundo
//
public class HolaMundo {
    public static void main(String args[]){
       System.out.println ("Hola Mundo!");
   }
}
```

2.10. Declaración de las Clases

Las variables de una clase se definen en el cuerpo de la clase.

Ejemplo:

```
public class Materia {
    String nombre;
    String codigo;
    int    unidadesCredito;
    int    horasTeoria;
    int    horasPractica;
}
```

2.11. Encapsulamiento

Ejemplo:

```
public class Circulo {
   private double radio;
   private Punto centro;

   public void setRadio(double r){}
   public void setCentro(double x,y){}
   public double getArea(){}
   public double getPerimetro(){}
}
```

2.12. Herencia

La aplicamos cuando tenemos dos clases que comparten varios atributos o métodos.

```
class Profesor{
   String nombre;
   String cedula;
   List   materias;
   long   sueldo;
   ...
}
```

```
class Alumno{
   String nombre;
   String cedula;
   List materias;
   int  promedio;
   ...
}
```

Herencia

Creamos una superclase con lo común

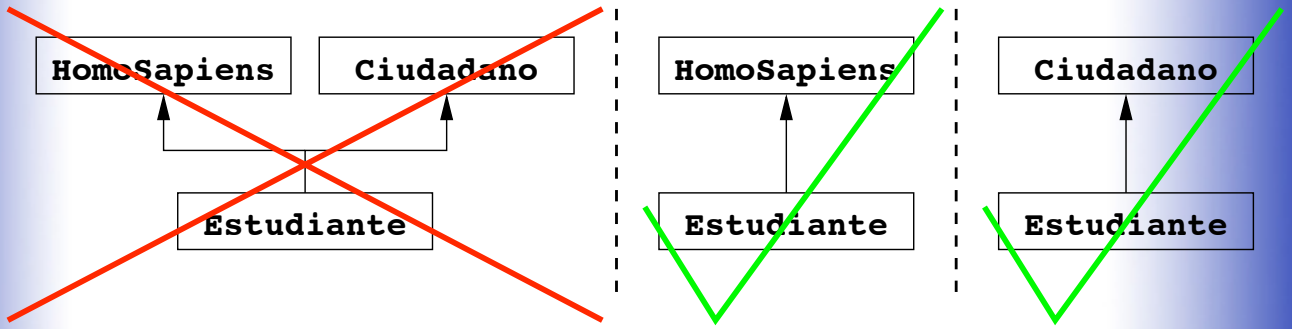
```
class Persona{
    String nombre;
    String cedula;
    List   materias;
}
```

```
class Profesor extends Persona{
    long   sueldo;
}
```

```
class Alumno extends Persona{
    int  promedio;
}
```

Herencia

Desde el punto de vista del paradigma de OO una clase solamente puede hereder de una única superclase[a].



[a]Existen lenguajes como el C++ que permite herencia de múltiples padres

Herencia: *¿Es un?*

Dos clases que comparten métodos y atributos no siempre están relacionadas por la herencia.

```
class HomoSapien {
    int numOjos=2;
    int numPatas=2;
    ...
}
```

```
class Gallina{
    int numOjos=2;
    int numPatas=2;
    int numAlas=2;
    ...
}
```

`Gallina` NO es subclase de `HonoSapiens`

2.13. Clases Contenedoras: *¿Tiene un?*

Son clases que contienen o agrupan referencias a objetos relacionados para así manipularlos.

```
class Punto{ ... }     // clase contenida

class Circulo{
   private Punto  centro;
   private double radio;
   public  void   setRadio(double r);
   public  double getArea(){...}
   ...
}
```

2.14. Clases Abstractas

Son clases generales donde se declaran uno o más métodos sin que se especifique su implementación

```
public abstract class Figura{
    public abstract double getArea();
    public abstract double getPerimetro();
}
```

Clases Abstractas

```
public class Circulo extends Figura{
    ...
    public  double getArea(){
        return Math.PI*radio*radio;
    }
    public  double getPerimetro(){
        return 2*Math.PI*radio;
    }
    ...
}
```

2.15. Agrupando Clases

Un grupo de clases relacionadas se pueden agrupar en un paquete, `package`.

- Es posible importar, `import`, una o varias clases de un paquete, para usarlas en nuestros programas.
- Es posible empaquetar varias clases dentro de un paquete,

Los paquetes en OO son el equivalente a las bibliotecas (*"librerias"*) en la programación estructurada

2.16. Paquetes

Estandar del Java 2

`java.util` Números aleatorios, fecha, propiedades del sistema.

`java.io` Manejo de E/S de archivos

`java.net` redes

`java.applet`

`java.awt` Abstract Windows Toolkit

`java.lang` El núcleo del lenguaje donde se declaran las clases `String`, `Math`, `Integer`, etc.

Empaquetando

Sintaxis:

```
package idPaquete;
```

- Facilita la distribución de aplicaciones
- Oculta clases y métodos sin que sean privados
- Permite el acceso a todas las clases y métodos no privados

Debe ir al principio del archivo donde esté la clase a empaquetar

Usando los Paquetes

Al principio, luego de la palabra `package`

- `import nombrePaquete.*;`
  Todas las clases en el paquete
- `import nombrePaquete.nombreClase;`
  Sólo la clase mencionada
- `import nombrePaquete;`
  Todas las clases del paquete usando nombre del paquete y de la clase
- Sin usar el `import` Para tener acceso a las clases hay que dar el camino completo

Código Fuente Java

1. Una declaración opcional `package`
2. Tantos `import` como sean necesarios
3. Definición de la clase y la interfaz

Los archivos fuente

- deben tener la extensión `.java`
- deben tener una única clase pública
- la clase pública debe llamarse como el archivo
- pueden tener varias clases no públicas

2.17. Métodos

- Los métodos en Java son las instrucciones.
- Los métodos se agrupan en bloques de acuerdo a qué o sobre qué actúan
- Un método puede invocar a otro método.
- Se llama recursión cuando un método se invoca a sí mismo.
- Todas las aplicaciónes Java comienzan con el método `main` que se encarga de invocar a otros métodos y de coordinar la ejecución del programa.

Métodos

- hacen que los programas sean más fáciles de leer
- facilitan el desarrollo y el mantenimiento
- son fundamentales para la reutilización de software
- evitan la duplicación de código
- se requieren para modificar y consultar a los atributos

Métodos

- Todos los métodos en Java se escriben dentro del cuerpo de alguna clase.
- Las acciones que un método lleva a cabo deben estar relacionadas con la clase a la que pertenece.

Sintaxis:

```
[⟨modif⟩] tipo idMetodo ( [argumentos] ) {
    cuerpoDelMetodo
}
```

Invocando Métodos

Por lo general, los métodos se invocan con la siguiente sintaxis

objeto`.`*idMetodo*`(`*[argumentos]*`)`

Ejemplo:

```
miCirculo.setRadio(5);
miCirculo.setCentro(4,1,-8);
otroCirculo.setRadio(1);
```

Métodos Estáticos `static`

- Son métodos muy generales de una clase
- No requieren que la clase esté instanciada para ser invocados

```
Clase . idMetodo ( [argumentos] )
```

- El método `main` debe ser estático.
- Su declaración tiene la sintaxis

```
[⟨modif⟩] static tipo idMetodo ( [args] )
```

2.18. Autoreferencias

Cuando un método invoca a otro y ambos son del misma clase no hay que hacer la referencia a este último.

```
public double semiArea(){
    return getArea()/2;
}
```

También se puede usar la referencia `this` para referenciar al objeto actual

```
public double semiArea(){
    return this.getArea()/2;
}
```

2.19. Sobrecarga de Métodos

Sobrecarga es cuando dentro de una clase existen dos o más métodos con el mismo nombre pero con parámetros diferentes

Al invocar a un método sobrecargado se determina cual de ellos usar gracias al número y tipo de los parámetros. Se ejecutará aquel método cuyos parámetros coincidan con la invocación.
La combinación del método con los argumentos se conoce como firma o *"signature"*

2.20. Métodos constructores

- Métodos especiales que se pueden usar al momento de instanciar una clase.
- La palabra `new` los invoca
- inicializan los atributos del objeto
- Si no se invoca el constructor, a los atributos se les asignará su valor por omisión
- No tienen tipo
- Tienen el mismo nombre que la Clase

Métodos constructores

Se declaran así

```
public class Racional{
    Racional(int n, int d){
        numerador = n;
        denominador = d;
    }
}
```

y se se invocan así

```
Racional f = new Racional(1,2);
```

Métodos constructores

Los constructores pueden ser sobrecargados

```
public class Racional{
    Racional(){
        Racional(0,1);
    }
    Racional(int n, int d){
        numerador = n;
        denominador = d;
    }
}
```

Métodos constructores

Los constructores con sobrecarga se usan así

```
Racional f = new Racional(1,2);
Racional g = new Racional();
```

- Toda clase debe tener al menos un constructor.
- Si no se implementa el constructor de una clase, el compilador Java creará automáticamente un constructor sin ningún argumento.

2.21. Conversión de Tipo "*Casting*"

- Permite cambiar el tipo de dato de una expresión
- Se usa generalmente cuando un método solamente acepta valores de un tipo determinado.
- De un tipo de menor rango a uno de mayor rango no hay pérdida de información
- De un tipo de mayor rango a uno de menor rango puede haber pérdida de información
- Debe hacerse en forma explicita.

Conversión de Tipo "*Casting*"

Sintaxis:

(*tipoNuevo*) *expresión*
constanteNumericaLetra

Ejemplo:

```
i = (long)2*edad;
long segundos = 1000L;
float x = 3.6;       // Error
float x = 3.6F;      // Correcto
double y = 3.6F;     // Correcto
```

Conversión de Tipo "*Casting*"

Los tipos primitivos pueden mezclarse libremente en las expresiones numéricas.

```
short a = 2;
double x = 3.6;

x = x + a;
a = a + a           // Error
a = (short)(a + a) // Correcto
```

`byte`, `short`, `char` → `int`
`int` → `long` → `float` → `double`

2.22. Operadores

Operadores Relacionales

Operación	Operador	Ejemplo
es igual que	==	`if(i == 1)`
es diferente de	!=	`if(i != 1)`
es menor que	<	`if(i <1)`
es mayor que	>	`if(i >1)`
es menor o igual que	<=	`if(i <= 1)`
es mayor o igual que	>=	`if(i >= 1)`

Operadores Lógicos

Operación	Operador	Ejemplo
Y	&&	`if((i==1)&&(j==2))`
O	||	`if((i == 1)||(j==2))`
O exclusivo	^	`if((i == 1)^(j==2))`
no	!	`if(!(i == 1))`
Sin cortocircuito		
Y	&	`if((i==1)&(j==2))`
O	|	`if((i==1)|(j==2))`

Operadores Aritméticos

Operación	Operador	Ejemplo
suma	`+`	`suma = a + b`
resta	`-`	`resta = a - b`
multiplicación	`*`	`mult = a * b`
división	`/`	`div = a / b`
módulo	`%`	`mod = a % b`

Operadores Incrementales

Operador	Ejemplo	Equivalente
`++`	`i++`	`i = i + 1`
`--`	`i--`	`i = i - 1`
`+=`	`a += b`	`a = a + b`
`-=`	`a -= b`	`a = a - b`
`*=`	`a *= b`	`a = a * b`
`/=`	`a /= b`	`a = a / b`
`%=`	`a %= b`	`a = a % b`

Precedencia y Asociatividad

$$2 - 6/2 * 3 - 1$$

Precedencia	Operadores	Asociatividad
2	+ - (unarios)	I $\rightarrow$ D
3	* / %	I $\rightarrow$ D
4	+ -	I $\rightarrow$ D

2.23. Estructuras de Selección

Usos

- No siempre se quiere ejecutar todas las instrucciones
- Se puede escojer entre dos o más conjuntos de instrucciones
- Se detecta un error y se trata de corregir
- El computador es más que una calculadora costosa

Selección simple

Ejecuta o no una instrucción o grupo de instrucciones

```
if( ExpresionLogica){
  Instrucción1;
}
```

Selección doble

Ejecuta una u otra instrucción

```
if( ExpresionLogica){
  Instrucción1;
else{
  Instrucción2;
}
```

```
ExpresionLog ? Instruccion1 : Instruccion2;
```

Selección anidada

Una decisión dentro de otra

Sintaxis:

```
if( ExpresionLogica1){
  Instrucción1;
else{
  Instrucción2;
  if( ExpresionLogica2){
    Instrucción3;
  }
}
```

Selección múltiple

Selecciona una entre múltiples instrucciones

Sintaxis:

```
switch (selector){
  case etiqueta1:
    Instrucción1 ; break;
  case etiqueta2:
    Instrucción2 ; break;
  default:
    InstrucciónN ;
}
```

2.24. Estructuras de Repetición

Los tres tipos de papás:

1. Al llegar primero pregunta y si se portó mal le da una nalgada

```
while ( expLog ) { instrucción ; }
```

2. Al llegar le da una nalgada y luego le pregunta cómo se portó

```
do { instrucción ; } while( expLog ) ;
```

3. Al llegar le da n nalgadas

```
for ( i = 0 ; i < n ; i++ ) { instrucción ; }
```

Repetición Anómala

- `break;`
  Finaliza la ejecución de un bucle de forma anómala, es decir, el bucle termina antes de lo previsto. También se utiliza en las instrucciones de selección múltiple.

- `continue;`
  Hace que el bucle prosiga en la próxima iteración omitiendo la ejecución de las instrucciones que le siguen.

3. Diseño de Clases OO

Al finalizar esta clase el estudiante deberá

- Crear clases utilizando el concepto de encapsulamiento
- Crear clases que hereden características de su antecesor
- Explicar algunos conceptos avanzados del diseño de clases como polimorfismo y clases intrínsecas

3.1. Encapsulamiento

- El acceso se provee mediante una interfaz
- La parte interna de la clase se mantiene segura
- Si la interfaz no se modifica, los cambios de la parte interna de la clase no afectan su uso
- Facilita la reutilización de código

Java provee dos maneras de encapsular:

- Usando los paquetes `package`
- Usando los constructores de las clases

Encapsulamiento usando paquetes

- Las clases que pertenecen a un paquete serán visibles para otras clases fuera del paquete si las primeras son declarar clases públicas `public`
- Todas las clases de un paquete que no sean declaradas públicas `public` estarán ocultas para todas las clases que no pertenecen al mismo paquete

Encapsulamiento usando la Clase

- Crear una clase que agrupe atributos y métodos
- Restringir el acceso a todos los atributos y a los métodos que lo requieran

Modificador	Interno	Paquete	Subclases	Otros
`private`	Si	No	No	No
	Si	Si	No	No
`protected`	Si	Si	Si	No
`public`	Si	Si	Si	Si

3.2. Herencia y Constructores

Las subclases heredan los atributos y los métodos de la superclase, pero no heredan el constructor de la superclase.
Hay dos formas de tener constructores en una clase:

- Escribiendo el o los constructores de la clase
- Utilizando el constructor por omisión

Herencia y Constructores

Al instanciar una clase ocurre:

1. Se pide la memoria en la pila para el objeto, incluyendo atributos declarados y heredados.
2. Si se invoca, se ejecuta otra forma del constructor
3. Se invoca el constructor de la superclase
4. Se inicializan los atributos declarados
5. Se ejecuta el resto del código del constructor

Note que los pasos 3 y 4 se ejecutan recursivamente

Herencia y Constructores

```
public class Alumno{
   String materia;
   public Alumno(){materia = "";}
   public Alumno(String m){materia = m;}
}

public class AlumnoPost extends Alumno{
   String postgrado;
   public AlumnoPost(String m, String p) {
      postgrado = p;
   }
}
```

Como no esta explicita la llamada al constructor utiliza el constructor por omisión `Alumno()` el cual asigna a la variable `materia` una cadena vacía

Herencia y Constructores

```
public class Alumno{
   String materia;
   public Alumno(){materia = "";}
   public Alumno(String m){materia = m;}
}

public class AlumnoPost extends Alumno{
   String postgrado;
   public AlumnoPost(String m, String p){
      super(m);
      postgrado = p;
   }
}
```

En el constructor de `AlumnoPost` debe invocarse el constructor de la clase ascendente `super` como primera línea

3.3. Herencia y Anulación/Sustitución

- El tipo de dato del método sustituto debe ser idéntico al tipo de dato del método sustituido
- El método sustituto no puede ser menos visible que el método sustituido
- El método sustituto no puede arrojar excepciones distintas a las arrojadas por el método sustituido

Dentro del método sustituto se puede ejecutar al método sustituido

Sintaxis:

```
super.idmetodo([args])
```

3.4. Herencia y Referencia

Al declarar una clase como extensión de una superclase, los objetos de la clase derivada son también objetos de la superclase.

```
Barco b;
Velero v;
Lancha l;
b = v;       //Un Velero es un Barco
b = l;       //Una Lancha es un Barco
v = b;       //Error! un Barco no es un Velero
Barco b1 = new Velero();      //Correcto
```

En la asignación `v=b` se genera una incompatibilidad de tipos

Herencia y Referencia

Se pueden evitar los errores en tiempo de compilación con el *casting*

```
class Puerto{
   public void main(String[] args){
       Barco b;
       Velero v;
       v = (Velero)b;       //Error al ejecutar
       b = v;
       v = (Velero)b;       //No hay error!!!
   }
}
```

Herencia y Referencia

SUMA

El objeto referenciado sabe de que tipo es

```
class Barco{
   public void tipo(){
     System.out.println("Barco");
}
class Velero extends Barco{
   public void tipo(){
     System.out.println("Velero");
   }
}
class Puerto{
   public static void main(String[] args){
       Barco b;
       Velero v;
       b = v;
       b.tipo();
   }
}
```

Velero

3.5. Clases y Métodos Abstractos

- Las clases abstractas llevan la palabra `abstract` en su declaración
- Una clase abstracta puede tener métodos y constructores no abstractos
- Cualquier clase con al menos un método abtracto debe ser abstracta
- No es posible instanciar clases abstractas

Clases y Métodos Abstractos

- En los métodos abstractos el cuerpo del método se sustituye por un `;`
- Los métodos abtractos no pueden usar los modificadores `private`, `static` y `final`
- Se puede declarar un objeto como una clase abstracta y almacenarlo como una subclase

```
abstract class Persona{ ... }
class Alumno extends Persona{ ... }
Persona a;
a = new Alumno();
```

3.6. El modificador `final`

- Prohibe que una clase tenga subclases
- Prohibe que un método sea sustituido
- En un atributo lo convierte en constante

```
public final int capacidad = 40;
```

- En el constructor se puede asignar valores a los atributos `final`

```
class Salon{
   public final int capacidad;
   Salon(int n){ capacidad = n; }
}
```

3.7. Polimorfismo

- La clase más alta de una jerarquía representa una interfáz común de todas sus subclases
- Las subclases representan diversas formas de objetos
- El usuario debe referirse a todas las formas de objetos usando la superclase.

Al invocar un método de un objeto referenciado con la superclase, el objeto seleccionará el método adecuado para su ejecución

Polimorfismo y las Colecciones

```
Barco[] flota = new Barco[1024];
flota[0] = new Velero();
flota[1] = new Velero();
flota[2] = new Lancha();
...
System.out.print(flota[0].tipo);
System.out.print(flota[1].tipo);
System.out.println(flota[2].tipo);
```

Velero Velero Lancha

3.8. Polimorfismo e `instanceof`

```
public class Persona extends Object;
public class Alumno extends Persona;
public class Profesor extends Persona;
...
public void setTrato (Persona p) {
   if (p instanceof Profesor)
      trato = "Prof. ";
   else if (p instanceof Alumno)
      trato = "Br. ";
   else
      trato = "Sr(a). ";
}
```

3.9. Polimorfismo y Casting

Si se tiene la declaración:

```
Barco b = new Velero();
```

Las partes específicas de un objeto `Velero` se mantienen ocultas en el objeto `b` por haber sido declarado tipo `Barco`
Para acceder a ellas se puede hacer casting:

```
Velero v = (Velero)b;
v.numVelas = 2;
```

o tambien:

```
((Velero)b).numVelas = 2;
```

3.10. Clases internas `inner`

Son clases declaradas dentro de otras clases

- Los nombres de la clase interna y externa deben ser diferentes
- Se definen con cualquier nivel de acceso. `private` solamente es visible por la clase externa

Al compilar la clase se generan dos archivos. El primero con el nombre:

claseExterior`.class`

y el segundo con el nombre:

claseExterior`$`*claseInterior*`.class`

4. Diseño de Interfaces Gráficas de Usuario

Al finalizar esta clase el estudiante deberá

- Crear Interfaces Gráficas de Usuario (GUI)
- Escribir código para la manipulación de eventos en las GUI
- Explicar el concepto de *applets*
- Crear aplicaciones y *applets* basados en GUI

4.1. Las Java Fundation Classes JFC

Es un conjunto de componente y servicios de GUI que simplifican el desarrollo de aplicaciones interactivas

- AWT: *Abstract Windowing Toolkit*
- Java 2D extension: Clases para gráficos 2D e imágenes
- Accessibility API
- Drag and Drop: Intercambio de datos entre aplicaciones
- JFC/Swing components: Componentes GUI

4.2. AWT: *Abstract Windowing Toolkit*

Provee los componentes básicos de las GUI.

- Un Componente es un objeto que tiene una representación gráfica.
- Casi todos los componentes heredan de la clase `java.awt.Component`
- El componente más importante es el `Container` el cual puede contener otros componentes incluyendo otros `Container`
- Los dos `Container` más usados son:`Panel` y `Frame`

AWT: *Abstract Windowing Toolkit*

- `Panel`: es un componente GUI invisible utilizado para contener otros componentes y debe estar contenido dentro de otro `Container`
- `Window`: Es una ventana nativa independiente de otros `Container`
  - `Frame`: ventana con título y esquinas que puede redimensionarse
  - `Dialog`: ventana que no tiene barra de menú, se puede mover pero no redimensionar

4.3. Frame

```
import java.awt.*;
public class MiFrame{
   private Frame elFrame;

   public MiFrame(String titulo){
      elFrame = new Frame( titulo );
   }
   public void run(){
      elFrame.setSize(200, 200);
      elFrame.setBackground(Color.blue);
      elFrame.setVisible(true);
   }
   public static void main(String args []){
      MiFrame demo = new MiFrame("HOLA");
      demo.run();
   }
}
```

4.4. Panel

```
import java.awt.*;

public class MiPanel{
   private Frame elFrame;
   private Panel elPanel;
   public MiPanel(String titulo){
      elFrame = new Frame(titulo);
      elPanel = new Panel() ;
   }
   public void run(){
      elPanel.setSize(100,100);
      elPanel.setBackground(Color.red);
      elFrame.setLayout(null) ;
      elFrame.setSize(200,200);
      elFrame.setBackground(Color.blue);
      elFrame.add(elPanel);
      elFrame.setVisible(true);
   }
   public static void main(String args []){
      MiPanel demo = new MiPanel("HOLA");
      demo.run();
   }
}
```

4.5. `Applet`

Son programas Java que corren en el ambiente de un navegador *web*.

- Siempre usan una GUI
- No tienen método `main`
- No se ejecutan mediante un comando
- Necesitan de una página web o de un interpretador applet para ser ejecutados
- En la página se especifica la forma de ejecución
- Tienen restricciones para acceder al sistema de archivos y para iniciar procesos

Corriendo `Applet`

- Crear un archivo HTML
- Incluir en el archivo HTML las etiquetas `APPLET` y `/APPLET`
- que enmarcan los artibutos:
  - `CODE=`*url*
  - `HEIGHT=`*altura*
  - `WIDTH=`*ancho*
- Cargar el archivo HTML en el navegador

```
import javax.swing.JApplet;
import java.awt.Graphics;

public class HelloWorld extends JApplet {
   public void paint(Graphics g) {
       g.drawRect(0, 0,
       getSize().width - 1,
       getSize().height - 1);
       g.drawString("Hello world!", 5, 15);
   }
}
```

Código html

```
<html>
<head>  <title>Contiene un Applet</title> </head>
  <body BGCOLOR="#D3D9E4">
  <center>
  <applet code=HelloWorld width=600 height=18>
  </applet> </center> </body>
</html>
```

Seguridad con Applets

Por lo general, los navegadores no le permiten a los Appletes:

- Acceder al sistema de archivo
- Ejecutar otros programas
- llamar a algún método nativo
- Abrir otros sockets diferentes al usado para cargar el applet

Operaciones de salida con Applets[a]

- `paint`: Se invoca cada vez que el *browser* determina que una porción del applet se debe dibujar
- `repaint`: Se invoca desde un manejador de eventos y le indica al *browser* que tiene que dibujar de nuevo al applet
- `update`: Borra y dibuja el area del applet.

Operaciones para Manejo de Applets

- `init()`: Inicializa al applet
- `start()`: Activa al applet: finaliza el `init()` y regresa a la página que contiene al applet
- `stop()`: finaliza el applet y abandona de la página del applet
- `destroy()`: Al cerrar el *browser*

[a] `System.out.println` generalmente se usa para dar información para el diagnóstico

Algunos métodos de los Applets

- `getParameter()`
- `getDocumentBase()`
- `getCodeBase()`
- `getImage()`
- `getAudioClip()`

4.6. `Dialog`

- Un componente `Dialog` está asociado a un contenedor `Frame`.
- Es una ventana autónoma.
- Se diferencia de un `Frame` porque:
  - Ofrece poca decoración y
  - Se le puede asociar un diálogo modal, lo que almacena todas las formas de entrada hasta que el `Dialog` se cierra.

4.7. Disposición de los Contenedores

AWT provee 5 clases para que implementan el `LayoutManager` y cada contenedor posee un `LayoutManager` por omisión.

- `FlowLayout`: `Panel`, `Applet`
- `BorderLayout`: `Window`, `Frame`, `Dialog`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

4.8. `FlowLayout`

- Los componentes aparecen de izquierda a derecha y de arriba hacia abajo
- El tamaño predefinido de los componentes es respetado
- Al cambiar el tamaño, la ubicación horizontal y vertical de los componentes puede cambiar

```
import java.awt.*;

public class DispFlow{
   private Frame miFrame;
   private Button Boton1;
   private Button Boton2;
   public DispFlow(){
      miFrame = new Frame("Flow");
      Boton1 = new Button("No hago nada");
      Boton2 = new Button("Yo tampoco");
   }
   public void run() {
      miFrame.setLayout(new FlowLayout());
      miFrame.add(Boton1);
      miFrame.add(Boton2);
      miFrame.pack();
      miFrame.setVisible(true);
   }
   public static void main(String args []){
      DispFlow demo = new DispFlow();
      demo.run();
   }
}
```

4.9. BorderLayout

Cada componente puede ser añadido a una de las 5 regiones del contenedor:

- `BorderLayout.NORTH`
- `BorderLayout.SOUTH`
- `BorderLayout.EAST`
- `BorderLayout.WEST`
- `BorderLayout.CENTER`

```
import java.awt.*;

public class DispBorde{
   private Frame frame;
   private Button b1, b2, b3, b4, b5;
   public DispBorde(){
      frame = new Frame("Borde");
      b1 = new Button("Norte");
      b2 = new Button("Sur");
      b3 = new Button("Este");
      b4 = new Button("Oeste");
      b5 = new Button("Centro");
   }
   public void run(){
      frame.add(b1, BorderLayout.NORTH);
      frame.add(b2, BorderLayout.SOUTH);
      frame.add(b3, BorderLayout.EAST);
      frame.add(b4, BorderLayout.WEST);
      frame.add(b5, BorderLayout.CENTER);
      frame.pack() ;
      frame.setVisible(true);
   }
   public static void main(String args []){
       DispBorde demo = new DispBorde();
       demo.run();
   }
}
```

4.10. `GridLayout`

Ubica los componentes dentro una tabla de filas y columnas. Poniendo un componente por celda.

- Las celdas son de igual tamaño
- Los componentes son agregados de izquierda a derecha y de arriba hacia abajo
- El tamaño predefinido de los componentes no se conserva

4.11. GridLayout

```
import java.awt.*;

public class DispGrid {
   private Frame frame;
   private Button boton[];
   public DispGrid(){
      frame = new Frame("Grid");
      boton = new Button[6];
      for (int i = 0; i < boton.length; i++)
          boton[i] = new Button("" + i);
   }
   public void run() {
       frame.setLayout(new GridLayout(3, 2));
       for (int i = 0; i < boton.length; i++)
          frame.add(boton[i]);
          frame.pack();
          frame.setVisible(true);
       }
   public static void main(String args[]) {
       DispGrid demo = new DispGrid();
       demo.run();
   }
}
```

4.12. `CardLayout`

Acomoda los componentes como una pila de barajas.

- Solamente acepta componentes `Container`
- El último componente está visible
- Los otros componentes son invisibles
- Los componentes pueden volverse visibles secuencialmente usando el método `next()`
- Puede volverse visible un componente cualquiera usando el método `show()`

CardLayout

```
import java.awt.*;

public class DispCard {
   private Frame frame;
   private Label etiqueta[];
   private Panel panel[];
   private CardLayout card;

   public DispCard(){
     frame = new Frame("Card");
     etiqueta = new Label[3];
     panel = new Panel[3];
     for(int i=0;i<etiqueta.length;i++){
       etiqueta[i] = new Label(""+i);
       panel[i] = new Panel();
     }
     card = new CardLayout();
   }
```

```
   public void run(){
      frame.setLayout(card);
      for(int i=0; i<etiqueta.length; i++){
         panel[i].setBackground (new Color(
            (int)(Math.random()*
            Integer.MAX_VALUE)));
         panel[i].add(etiqueta[i]);
         frame.add(panel[i],""+i);
      }
      card.show(frame, "2");
      frame.pack();
      frame.setVisible(true);
      for(int i=0,length=etiqueta.length*2;
      i<length;i++){
         try{ Thread.sleep(2000);}
         catch(InterruptedException e){}
         card.next(frame);
      }
   }
   public static void main(String args[]){
      DispCard demo = new DispCard();
      demo.run();
   }
}
```

4.13. Manejo de Eventos

Un evento es un objeto que representa a alguna acción llevada a cabo en la GUI.

- Ocurre un evento en la GUI
- La JVM crea un objeto que representa al evento
- La aplicación puede requerir al evento

El evento tiene información sobre qué componente lo originó
El paquete `java.awt.event` debe importarse cuando se quiere desarrollar GUI con manejo de eventos.

Manejo de eventos

Java implementa un modelo de manejo de eventos llamado *delegation event model*.

- Cada componente solicita que se le notifique sobre ciertos eventos
- Al occurir un evento se verifica la lista de componentes suscritos al evento
- un objeto que representa al evento es enviado a cada componente que lo solicitó
- Si no hay componentes que solicitan al evento, este se desecha

Manejo de eventos. Estrategia 1

- Identificar el origen del evento y con ayuda del API determinar la interfaz requerida para manejar el tipo de evento específico
- Definir una clase que implemente a la interfaz

```
import java.awt.*;
import java.awt.event.*;

public class ManejadorEx1 implements ActionListener {
    private int cnt;
    public void actionPerformed(ActionEvent e) {
        Button botonActivo = (Button) e.getSource();
        botonActivo.setLabel("Click : " + ++cnt);
    }
}
```

- Registrar una instancia de la clase manejadora del evento con el origen del evento

```
import java.awt.*;
public class EvBotonEx1 {
  Frame frame;
  Button boton1, boton2;
  EvBotonEx1() {
    frame = new Frame("Demo de la Estrategia 1");
    boton1 = new Button("BOTON 1");
    boton2 = new Button("BOTON 2");
  }
  public void run() {
    boton1.setFont(new Font("Serif", Font.BOLD, 16));
    boton2.setFont(new Font("Roman", Font.ITALIC, 24));
    boton1.addActionListener(new ManejadorEx1());
    boton2.addActionListener(new ManejadorEx1());
    frame.setLayout(new GridLayout(1, 2));
    frame.add(boton1);
    frame.add(boton2);
    frame.setSize(300, 125);
    frame.setVisible(true);
  }
  public static void main(String[] args) {
    EvBotonEx1 demo = new EvBotonEx1();
    demo.run();
  }
}
```

Manejo de eventos. Estrategia 2

SUMA

- Identificar el origen del evento y con ayuda del API determinar la interfaz requerida para manejar el tipo de evento específico
- Declarar a la aplicación como una implementación de la interfaz
- La aplicación implementa el método `actionPerformed`
- La aplicación se registra a sí misma como manejadora de eventos

```
import java.awt.*;
import java.awt.event.*;
public class EvBotonEx2 implements ActionListener{
  Frame frame;
  Button boton1, boton2;
  int cnt1, cnt2;
  public EvBotonEx2(){
    frame = new Frame("Demo de la Estrategia 2");
    boton1 = new Button("BOTON 1");
    boton2 = new Button("BOTON 2");
    cnt1 = 0;
    cnt2 = 0;
  }
```

```
  public void run(){
    boton1.setFont(new Font("Serif",Font.BOLD,16));
    boton2.setFont(new Font("Roman",Font.ITALIC,24));
    boton1.addActionListener(this);
    boton2.addActionListener(this);
    frame.setLayout(new GridLayout(1,2));
    frame.add(boton1);
    frame.add(boton2);
    frame.setSize(300,125);
    frame.setVisible(true);
  }
  public static void main(String[] args){
    EvBotonEx2 demo = new EvBotonEx2();
    demo.run();
  }
  public void actionPerformed(ActionEvent evento){
    Button botonActivo = (Button)evento.getSource();
    if(botonActivo.equals(boton1))
      botonActivo.setLabel("Click : " + ++cnt1);
    else
      botonActivo.setLabel("Click : " + ++cnt2);
  }
}
```

Manejo de eventos. Estrategia 3

- Identificar el origen del evento y con ayuda del API determinar la interfaz requerida para manejar el tipo de evento específico
- Se define dentro de la aplicación una clase interna que implemente a la interfaz
- Registrar una instancia de la clase manejadora del evento con el origen del evento

```
import java.awt.*;
import java.awt.event.*;
public class EvBotonEx3 {
  Frame frame;
  Button boton1, boton2;
  public EvBotonEx3(){
    frame = new Frame("Demo de la Estrategia 3");
    boton1 = new Button("BOTON 1");
    boton2 = new Button("BOTON 2");
  }
```

```
  public void run(){
    boton1.setFont(new Font("Serif", Font.BOLD, 16));
    boton2.setFont(new Font("Roman",Font.ITALIC,24));
    boton1.addActionListener(new ManejaBoton());
    boton2.addActionListener(new ManejaBoton());
    frame.setLayout(new GridLayout(1,2));
    frame.add(boton1);
    frame.add(boton2);
    frame.setSize(300, 125);
    frame.setVisible(true);
  }
  public static void main(String [] args){
    EvBotonEx3 demo = new EvBotonEx3();
    demo.run();
  }
  public class ManejaBoton implements ActionListener{
    private int cnt;
    public void actionPerformed(ActionEvent evento){
      Button botonActivo = (Button)evento.getSource();
      botonActivo.setLabel("Click : " + ++cnt);
    }
  }
}
```

4.14. Eventos, Interfaces y Métodos

Cada categoría de evento tiene una interfaz que debe ser implementada
Las interfaces tienen métodos que deben sustituirse con el código que maneja al evento

Categoría	Interfaz	Métodos
`Action`	`ActionListener`	`actionPerformed()`
`Item`	`ItemListener`	`itemStateChanged()`
`Adjustment`	`AdjustmentListener`	`adjustmentValueChanged()`
`Text`	`TextListener`	`textValueChanged()`

`Mouse`	`MouseListener`	`mousePressed()`
		`mouseReleased()`
		`mouseEntered()`
		`mouseExited()`
		`mouseClicked()`

`MouseMotion`	`MouseMotionListener`	`mouseDragged()` `mouseMoved()`

Key	KeyListener	keyPressed() keyReleased() keyTyped()
Focus	FocusListener	focusGained() focusLost()
Component	ComponentListener	componentMoved() componentHidden () componentResized() componentShown()
		windowOpened() windowIconified() windowDeiconified() windowClosed() windowActivated() windowDeactivated()
Container	ContainerListener	componentAdded() componentRemoved()

4.15. Manejo de Eventos Múltiples

Para manejar diferentes eventos asociados a una misma categoría se deben implementar cada uno de los métodos en el manejador de eventos.

- Si un evento no interesa, el método asociado se implementa vacío `{}`

Para manejar eventos de diferentes categorías la clase del manejador de eventos debe implementar todas las interfaces relacionadas con los eventos que se quieren manejar

```
public class Manejador implements
  ActionListener, WindowListener{
  ...
}
```

4.16. Adaptadores

Como puede ser molesto tener que implementar todos los métodos de una categoría cuando en realidad solamente se necesitan algunos de ellos, Java ofrece una clase adaptadora por cada categoría que tiene implementado todos los métodos sin ninguna instrucción en ellos.

Ejemplo:

```
WindowListener → WindowAdapter
MouseListener → MouseAdapter
KeyListener → KeyAdapter
```

4.17. Manejo de Eventos. Resumen

- Cree la GUI con todos sus componentes sin manejar los eventos
- Para cada categoría de evento cree el `Listener` correspondiente
- Recuerde crear el `WindowListener` para cerrar la aplicación
- Agregue al método donde se ejecuta la GUI las instancias de los `Listener` asociados a los componentes correspondientes

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class EjEventos extends WindowAdapter
        implements ActionListener, ItemListener {
    private Frame frame;
    private Choice menuColores;
    private String[] colores =
         {"Black", "Red", "Blue", "Green", "White"};
    private SketchCanvas hoja;
    private Panel panelBotones,  panelArchivo;
    private Button botonBorrar,  botonGuardar;
    private Label etiquetaArchivo;
    private TextField nombreArchivo;
    public EjEventos() {
        frame = new Frame("Ejercicio Manejo Eventos");
        menuColores = new Choice();
        hoja = new SketchCanvas();
        panelBotones = new Panel();
        botonBorrar = new Button("Borrar");
        botonGuardar = new Button("Guardar");
        panelArchivo = new Panel();
        etiquetaArchivo = new Label("Archivo");
        nombreArchivo = new TextField();
    }
```

```
    public void run() {
        for (int i = 0; i < colores.length; i++) {
            menuColores.addItem(colores[i]);
        }
        hoja.setBackground(Color.white);
        panelBotones.setLayout(new GridLayout(2, 1));
        panelBotones.add(botonGuardar);
        panelBotones.add(botonBorrar);
        panelArchivo.setLayout(new GridLayout(1, 2));
        panelArchivo.add(etiquetaArchivo);
        panelArchivo.add(nombreArchivo);
        frame.add(menuColores, BorderLayout.NORTH);
        frame.add(hoja, BorderLayout.CENTER);
        frame.add(panelBotones, BorderLayout.WEST);
        frame.add(panelArchivo, BorderLayout.SOUTH);
        menuColores.addItemListener(this);
        botonGuardar.addActionListener(this);
        botonBorrar.addActionListener(this);
        frame.addWindowListener(this);
        frame.pack();
        frame.setVisible(true);
    }
```

```
    public static void main(String[] args) {
        EjEventos demo = new EjEventos();
        demo.run();
    }
    public void actionPerformed(ActionEvent evento) {
        Button botonActivo = (Button) evento.getSource();
        if (botonActivo.equals(botonGuardar)) {
            System.out.println("No sabemos como guardar!!!");
        } else {
            hoja.borrar();
        }
    }
    public void windowClosing(WindowEvent evento) {
        System.exit(0);
    }
    public void itemStateChanged(ItemEvent evento) {
        String color = (String) evento.getItem();
        if (color.equals("Black")) {
            hoja.setForeground(Color.black);
        } else if (color.equals("Red")) {
            hoja.setForeground(Color.red);
        } else if (color.equals("Blue")) {
            hoja.setForeground(Color.blue);
        } else if (color.equals("Green")) {
            hoja.setForeground(Color.green);
        } else if (color.equals("White")) {
            hoja.setForeground(Color.white);
        }
    }
```

```
public class SketchCanvas extends Canvas
        implements MouseListener, MouseMotionListener {
    private Vector theVector;
    private int startX;
    private int startY;
    public SketchCanvas() {
        theVector = new Vector();
        setSize(500, 500);
        addMouseMotionListener(this);
        addMouseListener(this);
    }
    private void drawLine(Graphics graphics, int[] points) {
        graphics.drawLine(points[0], points[1],
                points[2], points[3]);
    }
    public void borrar() {
        Graphics graphics = getGraphics();
        Dimension dimension = getSize();
        int height = (int) dimension.getHeight();
        int width = (int) dimension.getWidth();
        graphics.clearRect(0, 0, width, height);
        theVector.clear();
    }
    public void paint(Graphics graphics) {
        Iterator iterator = theVector.iterator();
        while (iterator.hasNext()) {
            int[] points = (int[]) iterator.next();
            drawLine(graphics, points);
        }
    }
```

```
        public void mousePressed(MouseEvent evento) {
            startX = evento.getX();
            startY = evento.getY();
        }
        public void mouseDragged(MouseEvent evento) {
            Graphics graphics = getGraphics();
            int endX = evento.getX();
            int endY = evento.getY();
            int[] points = {startX, startY, endX, endY};
            drawLine(graphics, points);
            theVector.add(points);
            startX = endX;
            startY = endY;
        }
        public void mouseMoved(MouseEvent e) {}
        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
        public void mouseReleased(MouseEvent e) {}
    }
}
```

5. Manipulación y Optimización de Código OO

Al finalizar esta clase el estudiante deberá

- Manejar las excepciones que occuran durante la ejecución de un programa
- Crear un clase propia para el manejo de excepciones
- Usar hebras (`threads`) en sus códigos, extendiendo la clase `Thread` o implementando una interface `runnable`

5.1. Excepciones y Errores

Durante la ejecución de un programa pueden ocurrir eventos que no permitan que éste realice su labor. Estos eventos los detecta la JVM y si no se tratan hacen que el programa finalice.
Java maneja estos eventos con dos clases:

- `Error`: Para aquellos eventos que son fallas irrecuperables y el programa no los puede remediar.
- `Exception`: Para eventos relacionado a fallas de las que el programa se puede recuperar.

5.2. Excepciones

```
public class Division{
   public static void main(String[] args){
      for(int i = 0; i < 5; i ++) {
         int a = (int)(Math.random(*100);
         int b = (int)(Math.random(*4);
         int c = a/b;
         System.out.println(a+"/"+b+"="+c);
 }}}
```

En pantalla

```
Exception in thread "main"
        java.lang.ArithmeticException: / by zero
        at Division.main(Division.java:16)
Java Result: 1
```

Excepciones `try-catch`

Para manipular un sección de código que puede provocar alguna excepción:

- Encerramos esa porción de código en un bloque `try`
- Creamos un bloque `catch` para manejar cada posible excepción que se pueda producir en el código

```
public class Division {
    public static void main(String[] args){
       for(int i = 0; i < 100; i++){
          int a = (int)(Math.random()*100);
          int b = (int)(Math.random()*4);
          try{
              int c = a / b;
              System.out.println(a+"/"+b+"="+c);
          } catch(Exception e){
              System.out.println(a+"/"+b+" =oo");
}}}}
```

Manejo de Excepciones

Cuando una excepción/error ocurre, la JVM suspende la ejecución normal del programa y en el código el programador puede:

- incluir instrucciones en la estructura `try-catch` para manipularla
- Utilizar una clausula `throw` para pasarle el manejo de la excepción al método que invocó al método en ejecución
- No hacer nada. Con esta última alternativa solamente se pueden manejar la excepciones de tipo `RuntimeException`

Beneficio del uso de Excepciones

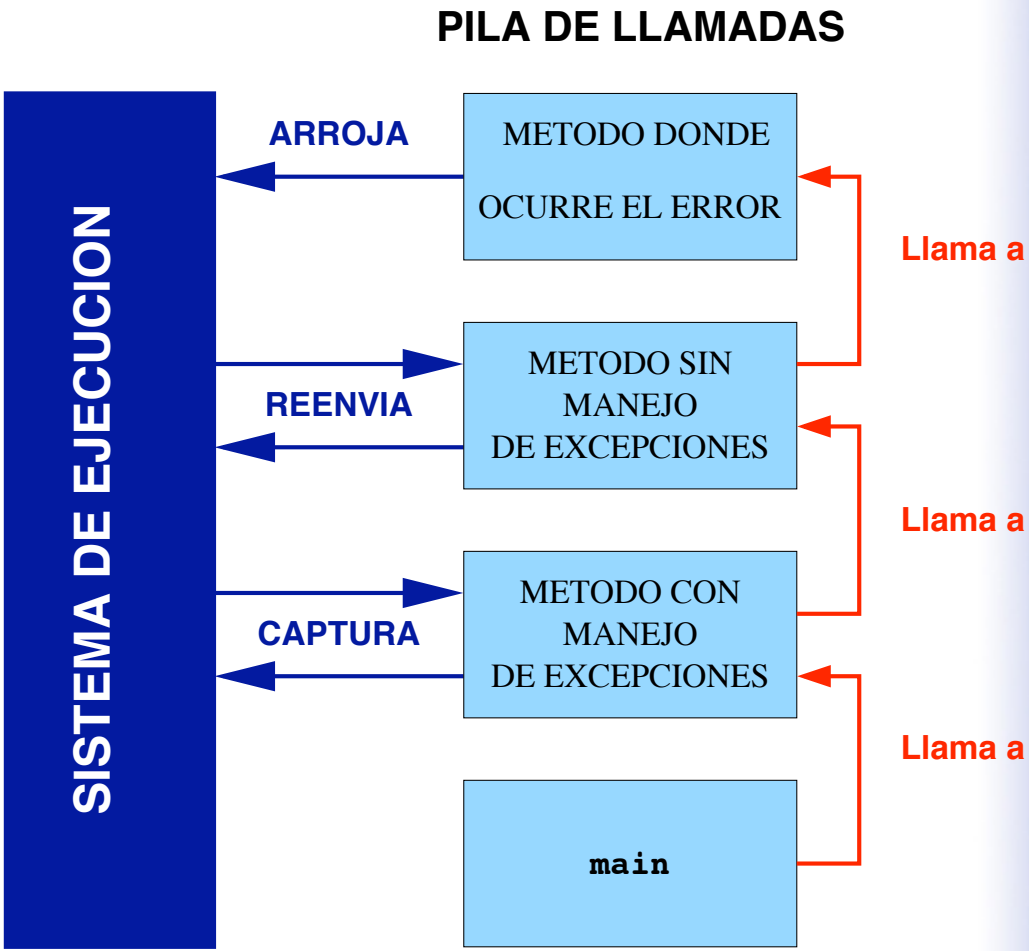
- Los mecanismos de manejo de excepciones son controlados en tiempo de compilación. Si un método arroja excepciones, el método que lo invoca está obligado a tener esto en cuenta.
- permiten separar la lógica del programa de la lógica del manejo de las excepciones

try - catch - finally

```
try{ código que produce excepciones }
catch(ClaseExcep1 objExcep1){
      código que maneja la excepción objExcep1
}
catch(ClaseExcep2 objExcep2){ ... }
catch(Exception objExcepN){
      código que maneja cualquier tipo de excepción
}
finally { código que siempre se ejecuta }
```

Excepciones y pila de metodos

SUMA



5.3. Manejo de Excepciones con `throws`

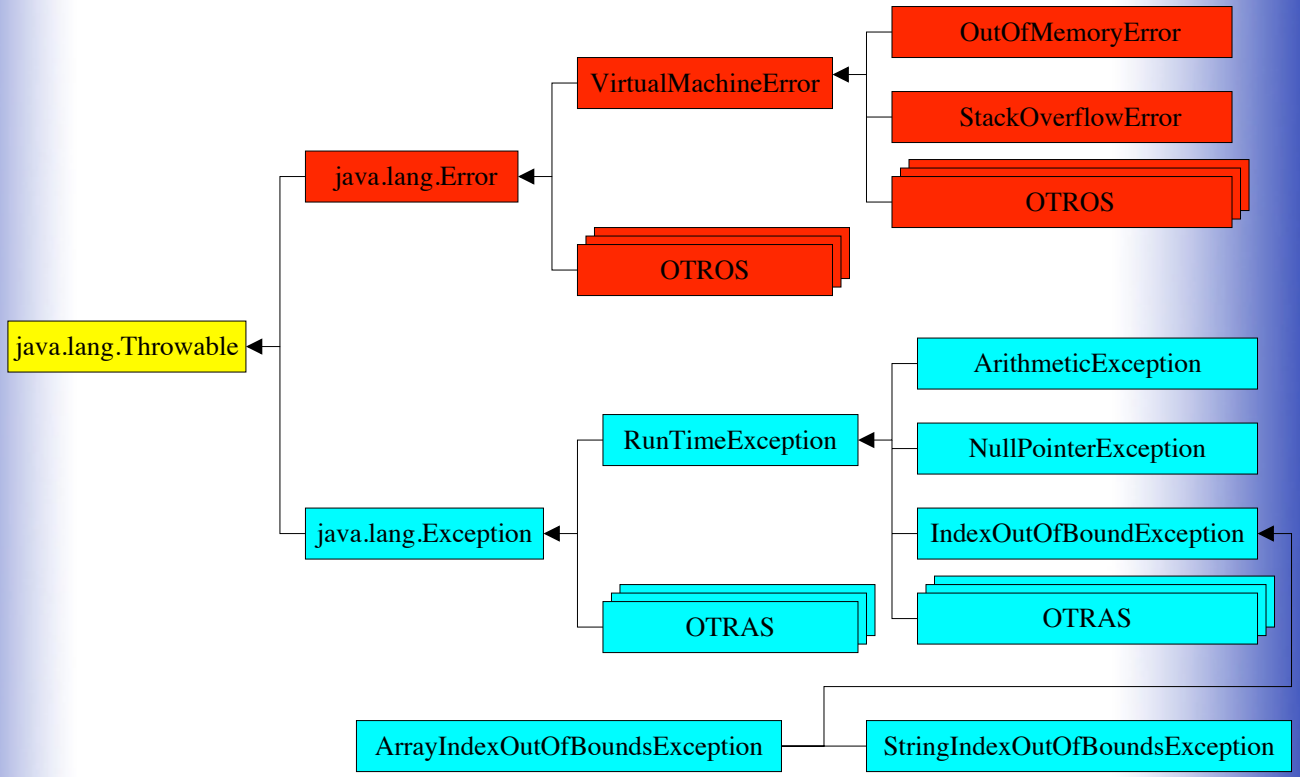
Si un método arroja alguna excepción al método que lo invoca se debe indicar en la declaración de la siguiente forma:

Sintaxis:

```
[⟨modif⟩] tipo idmetodo ([⟨args⟩]) throws lstExcep { [⟨cuerpo⟩] }
```

lstExceps: es la lista de todas las excepciones que el método pude arrojar o de susuperclase
Generalmente, el método donde se generan las excepciones no es el adecuado para manejarlas.
Estas reglas no aplican para las `RunTimeException` ni sus subclases

5.4. Jerarquía de Excepciones y Errores



5.5. Excepciones predefinidas

Las excepciones más comunes son:

- `ArithmeticException`: Resulta de la división de un entero entre cero[a].
- `NullPointerException`
- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundsException`
- `SecurityException`: Usualmente se arroja cuando un applet intenta realizar sin autorización.

[a] La división de un punto flotante entre cero devuelve la constante `NaN`

5.6. Excepciones definidas por el usuario

La declaramos así

```
public class NotaException extends Exception{
    public NotaException(String m){
    super(m);
   ...
}}
```

Y se usa así

```
if((nota > 20) || (nota < 0))
   throw new NotaException("Nota no valida");
```

5.7. Excepciones: Ejercicio

Calcule la desviación estandar de 10000 números aleatorios almacenados en un arreglo.

$$\sigma = \sqrt{\frac{\sum(x_i) - (\sum x_i)/n}{n-1}}$$

Ahora cálcule la desviación estandar cuando todos los datos valen 1/3

Excepciones: Ejercicio

```
public class Desviacion {
  public static void main(String[] args) {
    final int nDatos = 10000;
    double[] datos = new double[nDatos];
    double s = 0;
    double ss = 0;
    double sigma;
    int i;

    for(i=0; i nDatos; i++)
       datos[i]=Math.random()/3.;
    for(i=0; i < 10000; i++){
       s += datos[i];
       ss += datos[i]*datos[i];
     }
    sigma = Math.sqrt((ss-s*s/nDatos)/(nDatos - 1));
    System.out.println("Sigma = "+sigma);
   }
}
```

5.8. Aserciones

Es posible verificar una supocisión acerca de la lógica de un programa.

- Por ejemplo, la longitud de una cadena sea de al menos 1 caracter o que una variable numérica sea positiva.
- La ventaja de las aserciones es que pueden ser removidas del programa al ejecutarlo, lo que las diferencia de la Excepciones.

Uso de las aserciones

Sintaxis:

```
assert expLógica;

assert expLógica: mensaje;
```

El compilador no tiene activada su compilación por estar soportadas a partir de la versión sdk 1.4, por lo que para compilar:

```
javac -source 1.5 Clase.java
```

para ejecutar:

```
java -ea Clase
java -enableassertions Clase
```

Uso de las aserciones

Se usan para verificar:

- la lógica interna de un método
- la lógica de un grupo de métodos estrechamente relacionados
- si un método cubre con las expectativas

No se usan para:

- verificar el uso correcto del código

Para esto se usan las Excepciones

5.9. Multihebras *Threads*

Una hebra está compuesta por tres partes:

- Un CPU virtual
- Un código que se ejecuta
- Los datos con los que el código trabaja

Cada hebra tiene su propio CPU virtual pero pueden compartir código y/o datos

Diferencias entre Hebras y Procesos

- Los procesos son más pesados que las hebras y tardan más en ser cargados
- Los procesos utilizan más memoria que las hebras
- Los procesos no comparten datos por lo que hay que utilizar los mecanismos IPC. Las hebras pueden compartir datos

Creando hebras

Hay dos formas de crear hebras. La primera es:

- Creando una subclase de la clase `Thread`
  - Cree una subclase de `Thread`
  - Sobreescriba el método `run()` que contiene lo que la hebra debe hacer
  - En otra clase instancie a la subclase de `Thread` y ejecute su método `start()`
  - Al ejecutarse este último código el método `start()` llama a `run()`

Creando hebras

SUMA

```
public class DemoThread {
  public static class MiHebra extends Thread {
    int numHebra;
    public MiHebra( int n ) {
      numHebra = n;
    }
   public void run() {
     System.out.println("El n\'{u}mero de la
             hebra es " + numHebra );
    }
  }

  public static void main( String [] args ) {
    MiHebra t1 = new MiHebra( 1 );
    MiHebra t2 = new MiHebra( 2 );
    t1.start();
    t2.start();
  }
}
```

1	2
2	1

Creando hebras

La segunda es:

- Implementando una interfaz `Runnable`
  - Cree una clase que implemente `Runnable`
  - Sobreescriba el método `run()` de la interfaz `Runnable` con lo que la hebra debe hacer
  - En otra clase instancie a la clase anterior y cree una hebra pasándole al constructor `Thread()` el objeto que instancia a la clase e invoque al método `start()`
  - Al ejecutarse este último código el método `start()` llama a `run()`

Creando hebras

SUMA

```
public class DemoRunnable
  public class MiClase implements Runnable {
    int numRunnable;
    public MiClase( int n ) {
       numRunnable = n;
    }
    public void run() {
       System.out.println("En numero de la
           hebra es " + numRunnable );
    }
  }

  public static void main(String [] args ) {
      MiClase r1 = new MiClase( 1 );
      MiClase r2 = new MiClase( 2 );
      Thread t1 = new Thread( r1 );
      Thread t2 = new Thread( r2 );
       t1.start();
       t2.start();
  }
}
```

Creando hebras

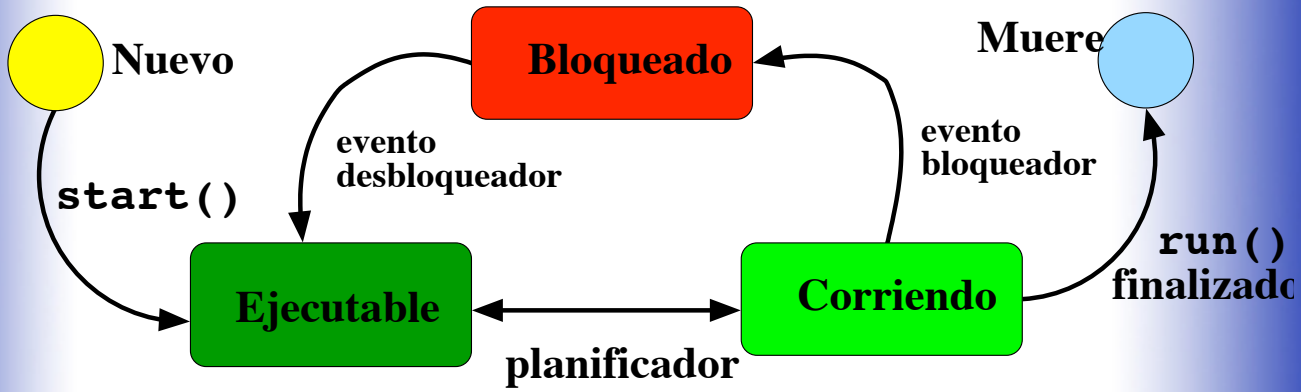
Subclase de `Thread` vs. interfaz `Runnable`

- Una subclase no puede tener dos superclases
- Con la implementación de `Runnable` un objeto puede correr varias hebras. En este caso, las hebras comparten los mismos datos

Las hebras tienen prioridad:

- `getPriority()`
- `setPriority()`

Control de Hebras



- `start()`: Pone a la hebra en el estado ejecutable
- `sleep()`: Abandona voluntariamente la ejecución y permitiéndole a otra hebra ejecutarse por un intervalo de tiempo.
- `yield()`: Abandona voluntariamente la ejecución pasando al estado ejecutable

Control de Hebras

- `isAlive`: Determina si una hebra ya paso al estado ejecutable pero aún no ha finalizado su ejecución
- `join`: hace que la hebra espere hasta que otra hebra finalice
- `interrupt`: interrumpe el estado de bloqueo de una hebra y la pasa al estado ejecutable

Para que una hebra finalice la ejecución de otra:

- Se crea una variable lógica que señala la finalización se inicializa en `false` y un método que permite cambiar el valor de la variable anterior a `true`
- En el método `run()` de la hebra se verifica periodicamente el estado de la variable, y de ser cierto se ejecuta un `break`
- Desde otra hebra se inicia e interrumpe una hebra cuando sea conveniente.

Sincronización de Hebras

Se necesita un mecanismo para garantizar que los datos compartidos entre varias hebras sean consistentes antes de que cada hebra tenga acceso a ellos y los pueda alterar.

Se necesita algún mecanismo de bloqueo

Java implementa un mecanismo de protección para datos compartidos para evitar interrupciones de ejecución de *código crítico*

```
synchronized
```

Sincronización de Hebras

- Cada objeto tiene una bandera de bloqueo
- Con la instrucción o el modificador `synchronized` se activa la bandera para evitar acceso al código que afecta a los datos compartidos
- `synchronized` puede aplicarse a bloques de código o a métodos.
- Una hebra al intentar ejecutar el código `synchronized` ve si éste no está bloqueado.
- Si está bloqueado la hebra se bloquea

Sincronización de Hebras

Cuando hebras compiten para acceder a los recursos puede ocurrir un *abrazo mortal*. Esto ocurre cuando una hebra está esperando por un objeto bloqueado por una segunda hebra, pero la segunda hebra espera por otro objeto bloqueado por la primera.
Java no detecta ni procura evitar esta condición. La regla general para evitar un *abrazo mortal* es:

- Establecer un orden en la cual los objetos se bloquean.
- Desbloquear los objetos en la orden inverso.

Interacción entre Hebras

`java.lang.Object` ofrece tres métodos para coordinar:

- `wait`
- `notify`
- `notifyAll`

6. Manejo de Entrada/Salida

- Definir flujo de datos entrada/salida
- Mostrar las características del paquete `java.io`
- Distinguir los canales de entrada y salida
- Construir canales entrada/salida

6.1. Fundamentos flujo Entrada/Salida

- Un canal *(stream)* puede ser considerado como un flujo de datos que representa datos provenientes de origen y/o datos a enviar a un destino.
- Un canal origen inicia el flujo de datos *(canal entrada - input stream)*.
- Un canal destino culmina el flujo de datos *(canal salida - output stream)*.
- Según el tipo de dato que manejan los canales pueden clasificarse en canales tipo caracter y tipo byte.
- Según su función los canales pueden ser clasificados como canales nodo y canales procesos.
- Existen varios tipos de canales nodo (archivos, arreglos memoria, conectores entre hebras, procesos) y canales proceso (buffer, filtro, conversión, numeración, búsqueda, impresión).
- Los canales pueden ser encadenados para facilitar el procesamiento de datos.

6.2. Tipos de Canales nodo

Java soporta canales tipo caracter y tipo byte.
Las clases java asociadas a los canales nodo son:

Tipo	Caracter	Byte
Origen	`Reader`	`InputStream`
Destino	`Writer`	`OutputStream`

6.3. Métodos asociados a canales Origen

`Reader`

```
int read()
int read(char[] buffer)
int read(char[] buffer, int offset, int length)
close()
boolean ready()
skip(long n)
```

`InputStream`

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
close()
int available()
skip(long n)
```

6.4. Métodos asociados a canales Destino

`Writer`

```
int write(int c)
int write(char[] buffer)
int write(char[] buffer,int offset,int length)
void write(String string)
void write(String string,int offset,int length)
void close()
void flush()
```

`OutputStream`

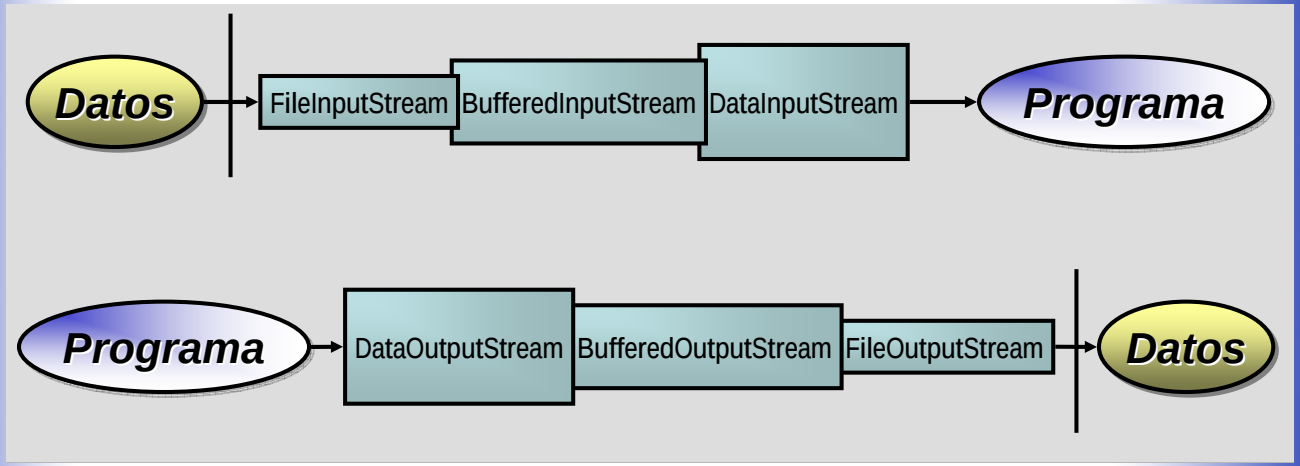
```
int write(int c)
int write(byte[] buffer)
int write(byte[] buffer,int offset,int length)
void close()
void flush()
```

6.5. Canales nodo

Java soporta canales nodo tipo archivo, arreglos, cadenas y tuberías, las clases java asociadas a los canales son:

Tipo	Caracter	Byte
Archivos	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Arreglos	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Cadenas	`StringReader` `StringWriter`	
Tuberías	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

6.6. Encadenamiento



6.7. Tipos de Canales de Procesamiento

Java soporta canales tipo caracter y tipo byte, las clases java asociadas a los canales de procesamiento son:

Tipo	Caracter	Byte
Buffer	`BufferedReader` `BufferedWriter`	`BufferedInputStream` `BufferedOutputStream`
Filtro(*)	`FilterReader` `FilterWriter`	`FilterInputStream` `FilterOutputStream`
Conversión	`InputStreamReader` `InputStreamWriter`	
		`ObjectInputStream` `ObjectOutputStream`
		`DataInputStream` `DataOutputStream`
Contador	`LineNumberReader`	`LineNumberInputStream`
Revisión	`PushbackReader`	`PushbackInputStream`
Impresión	`PrintWriter`	`PrintStream`

(*) Clase Abstracta

6.8. Ejemplo Entrada Salida Estándard

```
import java.io.*;

public class IO {
  public static void main(String[] args) throws IOException {
    byte buf[] = new byte[100];
    int len;

    //lee de la entrada estandar tres datos
    System.out.println("Nombre: ");
    len = System.in.read(buf);
    String name = new String(buf, 0, 0, len);

    System.out.println("Telefono: ");
    len = System.in.read(buf);
    String phone = new String(buf, 0, 0, len);

    System.out.println("Direccion: ");
    len = System.in.read(buf);
    String address = new String(buf, 0, 0, len);

    System.out.println("---------------------------------------");
    System.out.print("Nombre: " + name);
    System.out.print("Telefono: " + phone);
    System.out.print("Direccion: " + address);
  }
}
```

6.9. Ejemplo Entrada Salida con caracteres

```
import java.io.*;

public class IOChar {

  public static void main(String[] args) {
    try{
      FileReader input = new FileReader(args[0]);
      FileWriter output = new FileWriter(args[1]);
      char[] buffer = new char[128];
      int charsRead;

      // Lectura del archivo
      // se extraen datos usando el buffer
      // se almacenan en el nuevo archivo
      // hasta encontrar el fin de archivo
      charsRead = input.read(buffer);
      while (charsRead != -1){
        output.write(buffer,0,charsRead);
        charsRead = input.read(buffer);
      }
      input.close();
      output.close();
    } catch (IOException e){
      e.printStackTrace();
    }
  }
}
```

6.10. Ejemplo Entrada Salida usando Buffer

```
import java.io.*;

public class IOBuffer {
  public static void main(String[] args) {
    try{
      FileReader input = new FileReader(args[0]);
      BufferedReader bufInput = new BufferedReader(input);
      FileWriter output = new FileWriter(args[1]);
      BufferedWriter bufOutput = new BufferedWriter(output);
      String line;

      // Lectura del archivo
      // se extraen datos usando el buffer
      // se almacenan en el nuevo archivo
      // hasta encontrar el fin de archivo
      line = bufInput.readLine();
      while (line != null){
        bufOutput.write(line,0,line.length());
        bufOutput.newLine();
        line = bufInput.readLine();
      }
      bufInput.close();
      bufOutput.close();
    } catch (IOException e){
      e.printStackTrace();
    }
  }
}
```

7. Redes

- Describir la importancia de los protocolos en la comunicación en red.
- Describir las capas de funcionalidad en una red.
- Describir el Transmission Control Protocol (TCP) y el User Datagram Protocol (UDP)
- Mostrar las API disponibles para redes.
- Mostrar la importancia de los canales (stream) en la programación en redes.
- Desarrollar código para configurar una conexión a la red.
- Implementar el modelo de redes Java para realizar conexiones de red.

7.1. Compartir recursos

Frecuentemente, los programas necesitan acceso a recursos que residen en otros computadores (bases de datos, archivos, páginas web, mensajes de correo, impresoras, usuarios). Fundamentalmente el compartir información y recursos es la razón de la existencia de las redes de computadoras.

Desafortunadamente, los recursos de un computador no están directamente disponibles para los otros computadores, a veces no es posible determinar cuales/cuantos recursos están disponibles

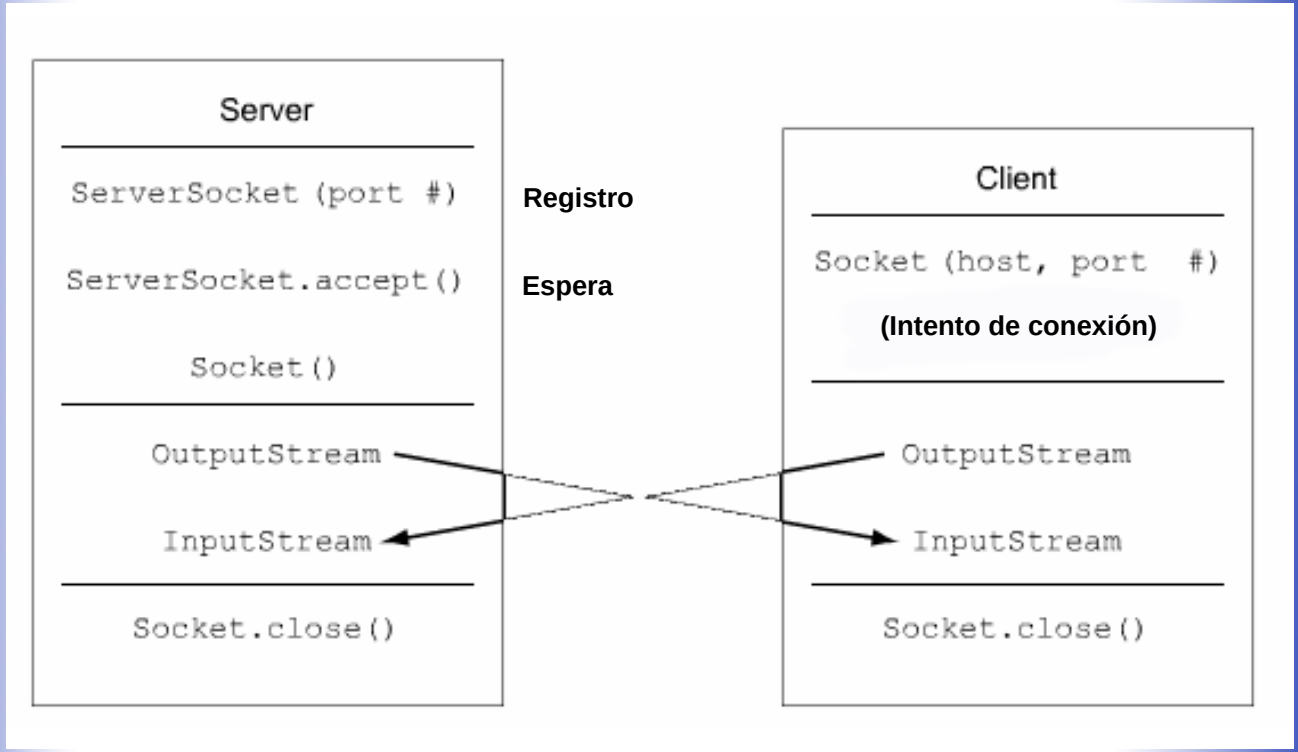
7.2. Protocolos de comunicación

Las computadoras se comunican usando los protocolos de comunicación. Un protocolo es un conjunto de reglas que controla la participación de dos computadoras en un mecanismo de comunicación. Un protocolo determina:

- Términos reconocidos
- Errores reportados
- Mecanismo de comunicación (transición)

Los protocolos son parte esencial de una comunicación. Si dos programas no usan el mismo protocolo no podrán comunicarse.

7.3. Modelo de red



Las Computadoras trabajan en diferentes niveles de abstracción, que corresponden a diferentes capas de una arquitectura de red. Las API del lenguaje son las encargadas de manejar los diferentes niveles.

7.4. Protocolos de Transporte

TCP Transport Control Protocol. Protocolo estándar para manejo de sesiones continuas sobre la red. Conexión Virtual. Intercambio intermitente.

UDP User Datagram Protocol. Protocolo no orientado a conexión. No verifica integridad de paquetes. Movimiento rápido de información.

7.5. Direccionamiento IP

Buena parte de la infraestructura de una red esta dedicada a la localización de los recursos de red. Los constructos asociados a la localización de recursos son:

IP address: Dirección lógica que identifica unívocamente a cada uno de los equipos conectados en la red. Cadena de 4 números en el rango 0-255 separados por punto.

Domain name: Nombre de dominio, alias que asocia uno o más nombres a la dirección lógica. Utiliza el servicio DNS.

URL: Una variación de una dirección IP que facilita el acceso a los recursos de Internet, que combina:

- Un protocolo de aplicación (http)
- Un dominio de nombre o dirección ip (www.sun.com)
- Un puerto (80)
- La ubicación del recurso o camino (public_html/home)
- El nombre del recurso (index.html)

Port: Puerto. Canal de Identificación. Un computador multitarea puede proporcionar mas de un servicio a la vez, o incluso el mismo servicio a multiples clientes.

7.6. APIs de redes

El paquete `java.net`, junto con el paquete `java.io` proporcionan los componentes requeridos para desarrollar una aplicación Java para redes. Las clases más frecuentemente usadas son:

`java.net.Socket`
: Conector socket de red. Comunicación bidireccional.

`java.net.ServerSocket`
: Socket para aplicaciones tipo servidor. Escucha solicitudes.

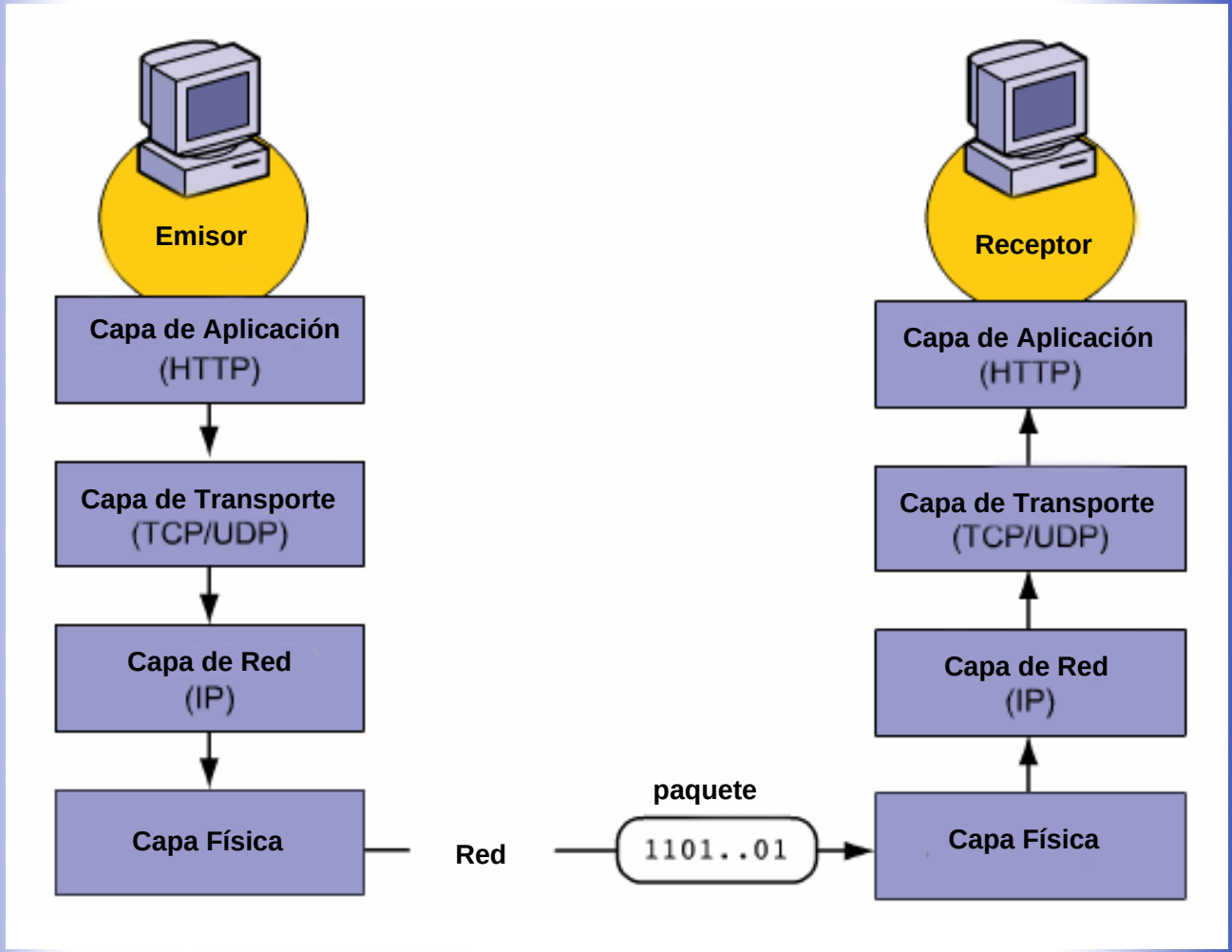
`java.net.URL`
: Clase URL, permite acceder a recursos en Internet.

`java.net.InetAddress`
: Direccion IP, permite acceder a recursos en Internet.

`java.io.*`
: Entrada/Salida

7.7. Modelo de redes

SUMA



7.8. Ejemplo Servidor

SUMA

```
import java.io.*;
import java.net.*;

public class SimpleServer {
   public static void main(String[] args) {
      ServerSocket s = null;
      // servicios por puerto 5432
      try {
        s = new ServerSocket(5432);
      } catch (IOException e) {
        e.printStackTrace();
      }
      // ciclo escucha/acepta
      while(true){
        try{
          //espera y escucha
          Socket s1 = s.accept();
          //asocia canal al socket
          OutputStream slout = s1.getOutputStream();
          BufferedWriter bw =
            new BufferedWriter(new OutputStreamWriter(slout));
          //envia cadena
          bw.write("Hola mundo!!!\n");
          //cierra conexin
          bw.close();
          s1.close();
        } catch (IOException e){
          e.printStackTrace();
        }
      }
  }
}
```

7.9. Ejemplo Cliente

```
import java.io.*;
import java.net.*;

public class SimpleClient {
   public static void main(String[] args) {
        try {
      // abre conexion en puerto 5432
      Socket s1 = new Socket("127.0.0.1",5432);
      // lectura-escritura
      BufferedReader br = new BufferedReader(
        new InputStreamReader(s1.getInputStream()));
      System.out.println(br.readLine());
      // cierra conexion
      br.close();
      s1.close();
    } catch (ConnectException connExc){
      System.err.println(
        "No es posible conectarse con el servidor");
    } catch (IOException e) {}
  }
}
```