Lenguaje Java con Bases de Datos

Facultad de Ingeniería UNAM

Agosto de 2006

Objetivos del Curso

- Obtener los conocimientos teórico-prácticos necesarios para:
  - ❖ Desarrollar y ejecutar programas de Java de complejidad media.
  - ❖ Poder utilizar las principales clases de la API de Java.
  - ❖ Utilizar Bases de Datos Relacionales desde programas Java, mediante la interfaz JDBC.
- Preparase para poder obtener la certificación de SUN Microsystems como Java Programmer.
- Prepararse para poder desarrollar aplicaciones WEB usando servlets, JSPs y EJBs.

Prerrequisitos

- Experiencia general en Programación.
- Conocimientos de Programación Orientada a Objetos.
- Conocimientos de conceptos de Bases de Datos Relacionales.
- Conocimientos básicos de SQL.

Contenido

- Módulo 1 – Introducción a la Tecnología java.
- Módulo 2 – Conceptos Básicos de Programación Orientada a Objetos.
- Módulo 3 – Identificadores, palabras clave y otros conceptos básicos.
- Módulo 4 – Expresiones y Estructura del programa.
- Módulo 5 – Manejo de Arreglos.
- Módulo 6 – Conceptos avanzados de clases (parte 1).
- Módulo 7 – Conceptos avanzados de clases (parte 2).
- Módulo 8 – Manejo de Excepciones.

Contenido (2)

- Módulo 9 – Archivos y Colecciones.
- Módulo 10 – Conceptos básicos de GUIs.
- Módulo 11 – Control de Eventos.
- Módulo 12 – Componentes y Menús en GUIs.
- Módulo 13 – Manejo de Bases de Datos con JDBC.
- Módulo 14 – Conceptos de Networking con Java.
- Módulo 15 – Conceptos de Multithreading en Java.
- Proyecto Final. Desarrollo de una Aplicación en Java con Interfaz Gráfica y Acceso a Bases de Datos.

Módulo 1.

Introducción a la Tecnología Java.

Objetivos.

- Entender qué es la Tecnología Java en sus diferentes versiones.
- Distinguir los principales tipos de programas Java.
- Conocer los objetivos de diseño de Java.
- Entender como se logran los objetivos de diseño de Java.

¿Qué es la Tecnología Java?

- Lenguaje de Programación.
- Ambiente de desarrollo.
- Ambiente de ejecución de aplicaciones.
- Ambiente de distribución de aplicaciones.

Tres versiones.

- Java SE, Java Standard Edition
  Conocida también como J2SDK, Standard Development Kit. Para desarrollo y deployment de aplicaciones en Java. Es el fundamento para J2EE.
- Java EE. Java Enterprise Edition.
  Para desarrollo y deployment en un Servidor de Aplicaciones de aplicaciones empresariales basadas en Servlets, Java Server Pages y Enterprise Java Beans.
- Java ME. Java Micro Edition.
  Para desarrollo y deployment de aplicaciones de Java en dispositivos móviles, como teléfonos, PDAs, TVs, etc.

Tipos de Programas Java.

- Aplicaciones.
  Programas convencionales que corren bajo control del Sistema Operativo.

- Applets.
  Programas que corren bajo un browser de Web (Explorer o Netscape).

- Java Beans.
  Componentes (en muchos casos gráficos) que siguen una serie de convenciones pre-establecidas.

- Servlets.
  Aplicaciones que se ejecutan en un Servidor de Aplicaciones y manejan, en general, la presentación gráfica del Sistema.

Tipos de Programas Java (2).

- JSPs (Java Server Pages).
  Elementos tipo HTML que son convertidos a Servlets por el Servidor de Aplicaciones.

- EJBs (Enterprise Java Beans).
  Aplicaciones que se ejecutan en un Servidor de Aplicaciones que implementan generalmente la lógica empresarial del Sistema.

Objetivos Primarios de Java.

- Facilidad de Uso.
- Portable entre plataformas diversas.
- Orientado a Objetos.
- Multithreading.

Facilidad de uso.

- Sintaxis basada en C++.
- Semántica basada en Smalltalk.
- Simplifica ambos lenguajes.
- Autodocumentable.

Portabilidad.

- Compilación e Interpretación.
- Corre en cualquier ambiente que soporte Máquina Virtual de Java.

Orientación a Objetos.

- Tecnología probada.
- Permite representar situaciones de la vida real naturalmente.
- En Java todo se basa en clases y objetos.

Multithreading.

- Facilidades integradas en la API de Java.
- Permite aprovechar la velocidad del procesador.
- Permite hacer programas complejos y profesionales fácilmente.

Logro de los objetivos.

Por medio de 3 mecanismos principales:

- Máquina Virtual de Java.
- Recolección de basura.
- Seguridad del código.

Máquina Virtual de Java.

“ Máquina imaginaria que es implementada ya sea mediante emulación de software o en una máquina real. El código que ejecuta la máquina virtual se encuentra en archivos *.class* que son resultado de la compilación del programa fuente”

Máquina Virtual de Java (2).

- Contiene especificaciones de hardware.
  - Instruction set
  - Register set
  - Stack
  - Heap ("garbage-collected")
  - Memoria
  - Formato de los archivos *.class*

Máquina Virtual de Java (3).

- Sus instrucciones son los “byte codes”, resultado de la compilación.
- Se implementa en software o en hardware.
- Se implementa como software en diversas plataformas o en browsers de la Web.

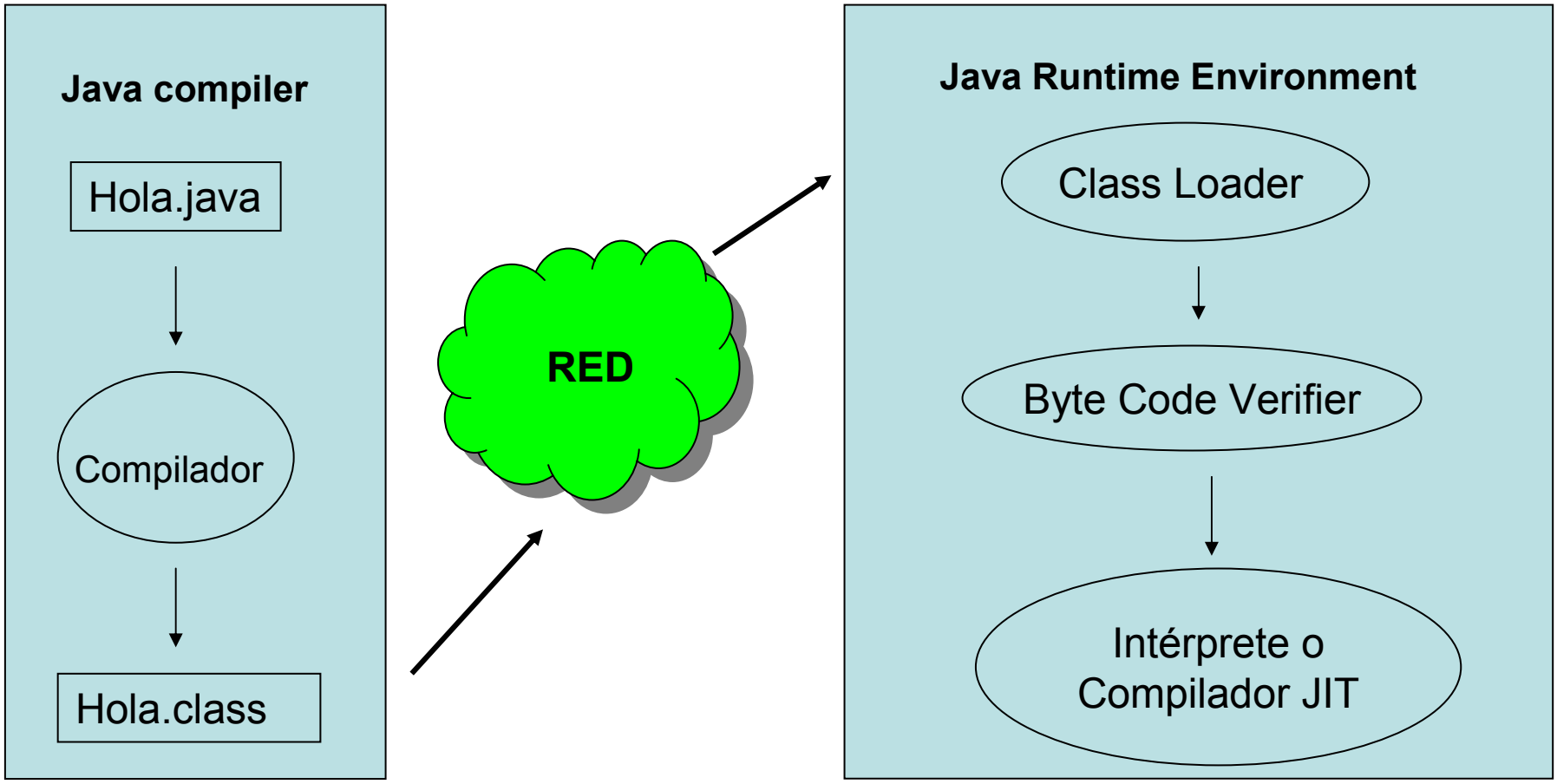
Máquina Virtual de Java (4).

- El formato de los “byte codes” está claramente especificado y es sencillo.
- Cada implementación de la JVM debe poder ejecutar cualquier archivo *.class*.
- Existen implementaciones en muchas plataformas: Unix, Windows, Mac, Mainframes, etc.

Recolección de basura.

- Libera memoria no utilizada sin intervención del programador.
  (En otros lenguajes es responsabilidad del programador).
- Es una tarea (thread) de la Máquina Virtual.
- Se hace en forma totalmente automática.

Compilación y Seguridad del código.



Java Runtime Environment.

- Carga el código del archivo *.class*
  - Class loader
- Verifica que esté correcto.
  - Bytecode verifier
- Ejecuta el código
  - Runtime interpreter

Bytecode Verifier.

Nos asegura que:

- El código cumple las especificaciones de la JVM.
- No viola la integridad del sistema.
- No causa overflows de memoria
- Los tipos de parámetros son los correctos
- No hay conversiones ilegales de datos.

Ejemplo de Aplicación.

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

Archivo HolaMundo.java

Compilación y Ejecución.

- Compilación:

```
javac HolaMundo.java
```

(produce archivo HolaMundo.class)

- Ejecución:

```
java HolaMundo
```

La API de java.

- Cientos de clases agrupadas en paquetes.

- Principales paquetes:

  - java.lang
  - java.applet
  - java.net
  - java.io
  - java.util

Repaso.

- Tecnología Java.
- Tipos de programas Java.
- Objetivos de diseño de Java.
- Herramienta de desarrollo de aplicaciones modernas tanto para Web como para desktop.
- La API de java proporciona cientos de clases que permiten resolver gran cantidad de situaciones de programación.

Módulo 2.

Programación Orientada a Objetos.

Objetivos.

- Definir los conceptos de: *abstraccción* y *encapsulamiento.*
- Entender el concepto de *paquete.*
- Definir *clase*, *miembro*, *atributo*, *método y constructor.*
- Usar los modificadores de acceso *private* y *public* para implementar el encapsulamiento.
- Invocar un método de un objeto.

Objetivos (2).

- Identificar los siguientes componentes de un Programa de Java:
  - El postulado package .
  - El postulado import.
  - Clases, métodos, y atributos.
  - Constructores.
- Usar la documentación online de la interfaz de programación de Java (API).

Abstracción.

- Esconder los detalles de la implementación dentro de una “caja negra”.

- Proporcionar una interfaz pública fácil de usar y de entender.

Tipos de Abstracción.

- Funciones.

  - Escribir un algoritmo una vez para poder utilizarlo en múltiples ocasiones.

- Datos.

  - Múltiples representaciones en grupos de bytes (binario, decimal, texto).

- Objetos.

  - Agrupar un conjunto de atributos y comportamiento relacionados en una clase.

- Frameworks y APIs.

  - Grandes grupos de Objetos que soportan una actividad compleja. Pueden ser usados directamente (“as is”) o extendidos por el usuario.

Concepto de Clase.

- Es la descripción o definición de un objeto; describe:
  - las características de los objetos, llamadas *atributos* o *variables de instancia* en java.
  - las acciones (comportamiento) de los objetos, llamadas *métodos* en java.
- Análoga a un plano, patrón o molde para crear objetos.
- **NO** es un *conjunto o grupo* de objetos.

Concepto de Clase (2).

- El proceso de crear objetos a partir de la clase se llama *instanciación*.
- En java se lleva a cabo mediante la palabra clave *new*.
- Por ejemplo:

```
Ventana vent1 = new Ventana();
```

Declaración de una Clase.

- Sintaxis Básica:

```
[modificadores] class  className  {
        [declaración de atributos]
        [declaración de constructores]
        [declaración de métodos]
}
```

- Ejemplo:

```
public class Vehicle {
        private double maxLoad;
        public void setMaxLoad(double value) {
                maxLoad = value;
        }
}
```

Declaración de Atributos.

- Sintaxis básica:

  [modificadores] tipo nombre [= valorInicial] ;

- Ejemplos:

```
public class Foo {
        private int x;
        private static float y = 10000.0F;
        private String name = “Universidad";
}
```

Declaración de Métodos.

- Sintaxis básica:

```
[ modificadores] tipoDeRetorno nombre ([listaDeArgumentos]) {
        [ bloque de postulados]
}
```

- Ejemplos:

```
public class Dog {
        private int weight;
        public int getWeight() {
                return weight;
        }
public void setWeight(int newWeight) {
        weight = newWeight;
}
}
```

Acceso de métodos y atributos.

- Notación de punto:

```
objeto.miembro = valor;          // si miembro es atributo
objeto.miembro(valor);           // si miembro es método
```

- Miembro es el nombre genérico de métodos y/o atributos.

- Ejemplo:

```
Dog d = new Dog();
d.setWeight(42);
d.weight = 42;        // sólo válido si weight es public
```

Encapsulamiento.

- La definición de la clase:

```
public class MyDate {
        public int day;
        public int month;
        public int year;
}
```

- El Problema:

```
MyDate d = new MyDate();
d.day = 32;                     // inválido
d.month = 2; d.day = 30;        // plausible pero inválido
d. day = d.day +1;              // no checa por posible invalidez
```

Encapsulamiento (2).

- La definición de la clase:

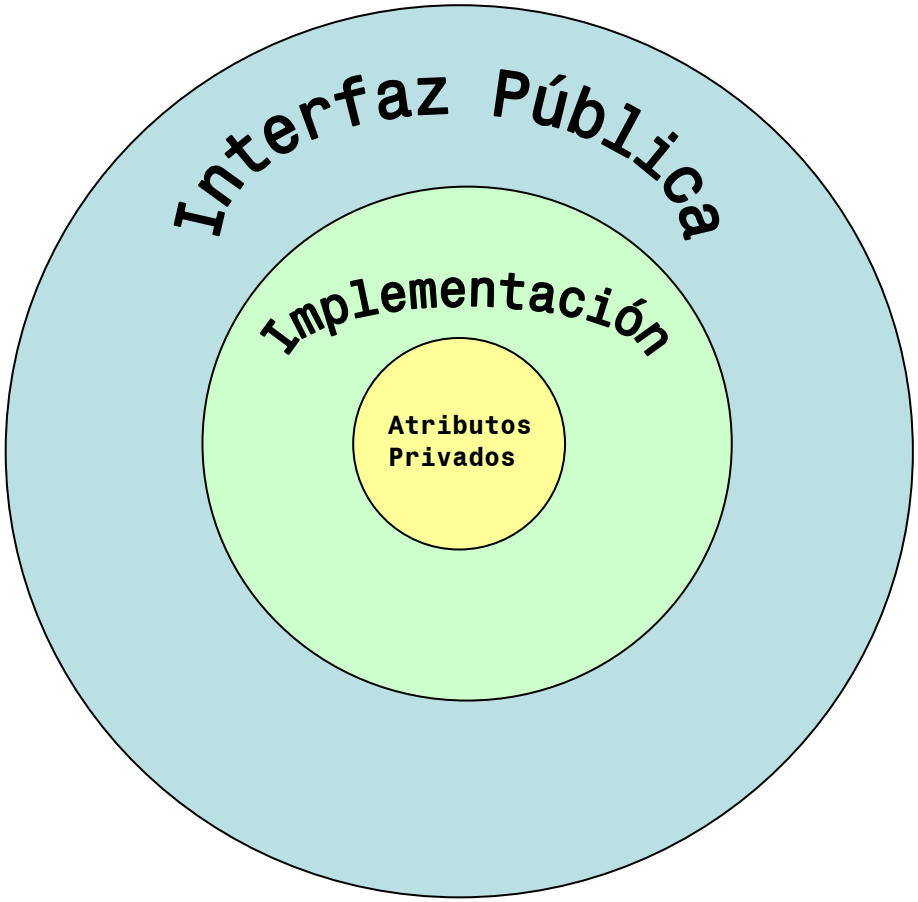
```
public class MyDate {
        private int day;
        private int month;
        private int year;
        public int getDay() {
                return day,
        }
        public boolean setDay(int day) {
                // valida day y regresa falso si es inválido
        }
        // métodos similares para month y year
}
```

Encapsulamiento (3).

- La Solución del Problema:

```
MyDate d = new MyDate();
d.setDay(32);                    // regresa false
d.setDay(25);                    // regresa true

int goodDay = d.getDay()         // entrega day validado
```

Encapsulamiento (4).



Declaración de Constructores.

- Sintaxis Básica:

```
[modificadores] nombreDeLaClase ([listaDeArgumentos]) {
    [bloque de postulados]
}
```

- Ejemplo

```
public class Dog {
    private int weight;
    public Dog() {                  // Constructor 1
        weight = 42;
    }
    public Dog(int kilos) {         // Constructor 2
        weight = kilos;
    }
    public int getWeight() {
        return weigth;
    }
}
```

Uso de Constructores.

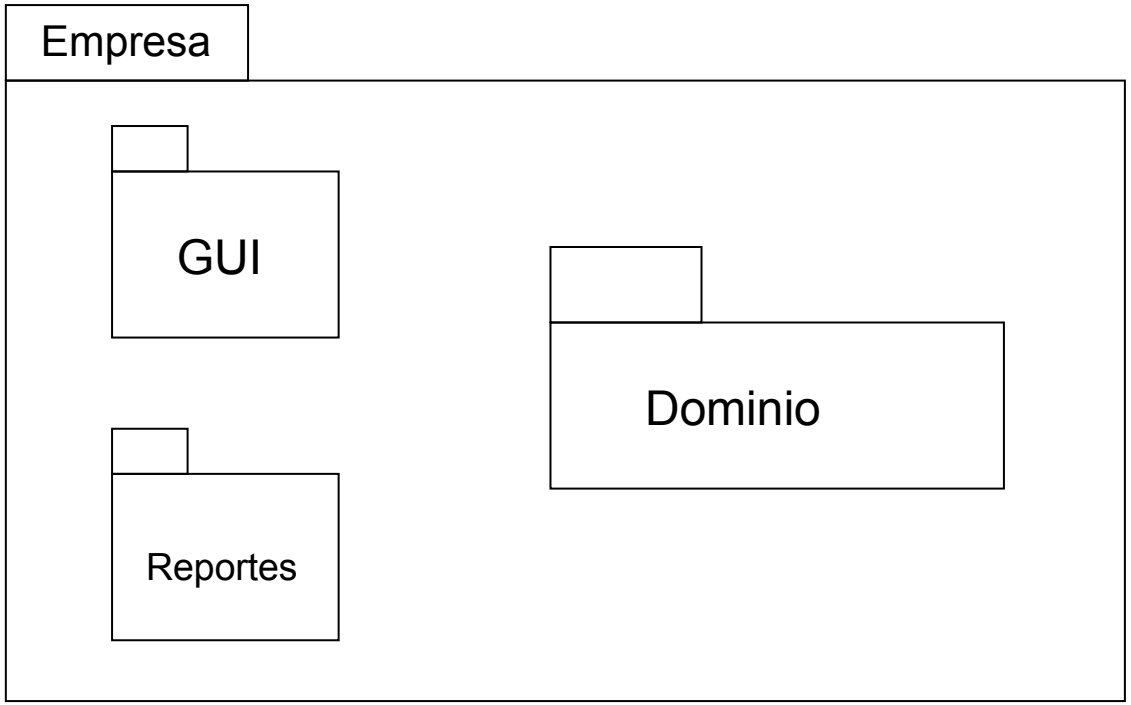
```
Dog oneDog = new Dog();
oneDog.getWeight();          // entrega 42

Dog anotherDog = new Dog(80);
oneDog.getWeight();          // entrega 80
```

El Constructor Default.

- Si el desarrollador de la clase **no incluye ningún constructor**, Java inserta uno:
  - sin argumentos
  - con cuerpo nulo.
- Si el desarrollador incluye cualquier constructor, **no hay** Constructor Default.
- Esto permite usar **new Xyz();** sin preocuparse por el constructor.

Paquetes.

- Conjuntos de clases y/o otros paquetes (subpaquetes) agrupados por funcionalidad.
- Implementados en jerarquías de subdirectorios.



El Postulado Package.

- Syntaxis:

  package nombrePackage[.nombreSubPackage …];

- Ejemplo:

  package empresa.reports;

- Debe ser el primer postulado del archivo. (excepto comentarios)
- Sólo un postulado package por archivo fuente.
- Si no se declara paquete, la clase pertenece al package default (implementado en directorio actual).

El Postulado Import.

- Syntaxis:

```
import nombrePackage[.nombreSubPackage…].nombreClase;
import nombrePackage[.nombreSubPackage…].*;
```

- Ejemplo:

```
import shipping.domain.*;
import java.util.List;
import java.io.*;
```

- Precede al postulado class.

- Indica al compilador dónde encontrar clases utilizadas en el programa.

Formato del archivo fuente.

- Sintaxis básica:

  [postulado package]
  [postulados import]
  declaración de clase

- Un postulado package solamente (opcional).
- Uno o varios postulados import (opcionales).
- Una o más definiciones de clase, pero sólo una pública.
- El nombre de la clase pública (o única) debe ser igual al nombre del archivo.

Ejemplo de archivo fuente.

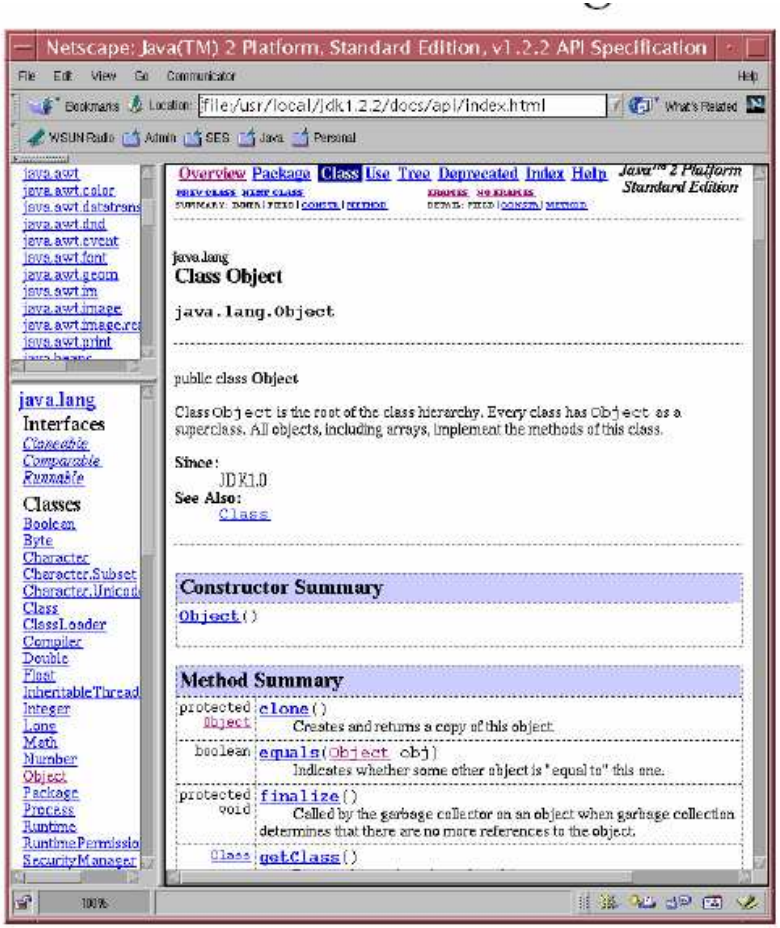
```
package bank.account;
import bank.domain.*;
import java.util.List;
import java.io.*;
public class CheckingAccount {
   private List history;
   public void getBalance(int accNumber) {...}
}
```

Archivo CheckingAccount.java

Documentación de la Java API.

- Conjunto de archivos HTML con links.
- Proporciona toda la información de cientos de clases que componen la API.
- Cada archivo de clase contiene descripción de la propia clase, atributos, constructores y métodos.

Ejemplo de la Documentación.



Ejercicios.

1. Utilización de la Documentación de la Java API.

2. Encapsulamiento.

3. Creación de la estructura básica de un Sistema Bancario.

Repaso.

- Definir los conceptos de abstracción y encapsulamiento.
- Definir atributo, método, constructor y paquete.
- Invocar un método de una clase.
- Identificar lo siguiente en un programa de Java:

    - el postulado package.
    - los postulados import.
    - clases, métodos, atributos y constructores.

- Utilizar la documentación de la API de Java.

Módulo 3.

Identificadores, palabras clave y otros conceptos básicos.

Objetivos.

- Utilizar comentarios en el programa.
- Distinguir entre identificadores válidos e inválidos.
- Reconocer las palabras claves de Java.
- Listar los 8 tipos primitivos de Java.
- Definir valores literales de los primitivos.
- Definir los términos variables primitiva y variable de referencia.

Objetivos (2).

- Declarar variables de tipo clase.
- Construir objetos utilizando **new.**
- Describir la inicialización de default.
- Entender el significado de las variables de referencia.
- Manejar la asignación de variables de referencia.

Comentarios.

Tres tipos de comentarios permitidos.

- Estilo C++

```
// comentario en una sola línea.
```

- Estilo C

```
/* comentario en una o
   varias líneas */
```

- Estilo JavaDoc

```
/** comentario que será procesado por
    JavaDoc   */
```

Postulados y Bloques.

- Un postulado es una o más líneas de código terminadas por punto y coma.

```
totals = a + b + c
+ d + e + f;
```

- Un bloque es un conjunto de postulados delimitados por llaves { }.

```
{ x = y + 1;
  y = x + 1;
}
```

Bloques y Espacio en Blanco.

- Se puede usar un *block* en una definición de clase o de método:

```
public class MyDate {
   private int day;
   private int month;
   private int year;
   public int getDay() {
         return day;
   }
}
```

- Se pueden anidar bloques.
- Se permite cualquier cantidad de "espacio en blanco" (*white space)* en un programa.
- Espacio en blanco puede ser:
  - espacios.
  - carriage return.
  - tabs.

Identificadores

- Son los nombres de clases, variables o métodos.
- Pueden empezar con una letra Unicode[1], guión bajo o signo de pesos.
- Ejemplos:
  - identifier
  - userName
  - user_name
  - _sys_var1
  - $change
- Sin embargo, no se recomienda usar _ o $ en general.
- No tienen límite de longitud.

[1] Ver www.unicode.org

Palabras reservadas.

abstract	default	*goto*	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	t*rue*
case	extends	int	short	try
catch	*false*	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	*null*	synchronized	

- sólo minúsculas.
- *goto* y *const* son reservadas pero no se usan (existen por legado de C++).
- *true* y *false* son literales booleanas.
- *null* es literal.
- las demás son palabras clave (keywords).
- enum es nueva keyword en versión 1.5

Tipos Primitivos.

- Existen 8 tipos de datos primitivos:
  - Lógico – boolean
  - Textual – char
  - Entero – byte, short, int y long
  - Flotante – double y float
- Todos los demás tipos de datos son objetos

Tipo Boolean.

- Dos literales:
  - true
  - false

- Ejemplo

```
boolean truth = true;
```

- declara la variable truth como tipo boolean y le asigna el valor true.

Tipo char.

- Representa un carácter Unicode de 16 bits.
- Literales deben ir entre apóstrofos. (' ')
- Notaciones especiales:
  - '\t' // tab
  - '\n' // new line
  - '\uxxxx' // carácter Unicode hexadecimal xxxx
- Ejemplos

```
'a'          // la letra a
```

Tipos enteros: byte, short, int, long.

- Tres formas literales: decimal, octal, o hexadecimal.
- El default es int.
- Se puede usar el sufijo L (o ℓ) para forzar a long.
- Ejemplos:
  - 2 // el valor decimal 2.
  - 24L // el valor decimal 24 en formato long.
  - 077 // el valor octal 77 = 81 decimal.
    // (el 0 al principio indica octal).
  - 0xBAAC // El valor hexadecimal BAAC.
    // (0x al principio indica hexadecimal.)

Tamaño y rango de los enteros.

tipo	longitud	rango
byte	8 bits	-2^{7} a $2^{7}-1$
short	16 bits	-2^{15} a $2^{15}-1$
int	32 bits	-2^{31} a $2^{31}-1$ (*)
long	64 bits	-2^{63} a $2^{63}-1$ (**)

(*) aprox. 2.15×10^{9}

(**) aprox. 9.22×10^{18}

Tipos flotantes: float y double

- Las literales incluyen punto decimal o los sufijos:
  - E o e (notación científica)
  - F o f (float)
  - D o d (double)
- Default is double
- Ejemplos
  - 3.14 // valor 3.14 en formato double.
  - 6.02E23 // valor 6.02 x 10^{23}
  - 2.718F // valor 2.781 en formato float.
  - 23.4D // valor 23.4 en formato double (la D es redundante)

Tamaño de los flotantes.

tipo	longitud
float	32 bits
double	64 bits

Nota. El rango depende de la precisión.

Valores de Default.

- Los atributos primitivos siempre tienen valores de default:

byte, short, int, long	0
float, double	0.0
char	null
boolean	false

Tipo String.

- No es un primitivo, es un objeto.
- Literales entre comillas (" ")
- Ejemplos:
  - String saludo = "Buenos días\n";
  - String errorMessage = "Record Not Found !";

Ejemplos de declaraciones y asignaciones.

```
public class Assign {
   public static void main (String args []) {
     int x, y;                // declaración de 2
   variables tipo int.
     float z = 3.414f      // declaración e
   inicialización de una variable tipo float.
     double w = 3.1415;  // declaración e
   inicialización de una variable tipo double.
     boolean truth = true; // declaración e
```

Tipos de referencia.

- Excepto los 8 primitivos, todos los demás tipos de datos en Java son referencias.
- Una variable de referencia contiene la dirección de memoria de un objeto.
- La *keyword* new consigue memoria en forma dinámica y entrega la dirección.
- Normalmente, esta dirección es asignada a una variable de referencia.
- La variable de referencia apunta al objeto.
- Generalmente se emplean como sinónimos variable de referencia y objeto.

Creación e inicialización de objetos.

- Ejemplo:

```
public class MyDate {
   private int day = 1;
   private int month = 1;
   private int year = 2003;
   public MyDate() {
      // postulado nulo
   }
   public MyDate (int day, int month, int year) {
      this.day = day;
      this.month = month;
      this.year = year;
    }
   public void print( ) {
      // imprime los atributos del objeto
   }
}
```

Creación e inicialización de objetos (2).

- Declaración de la variable de referencia:

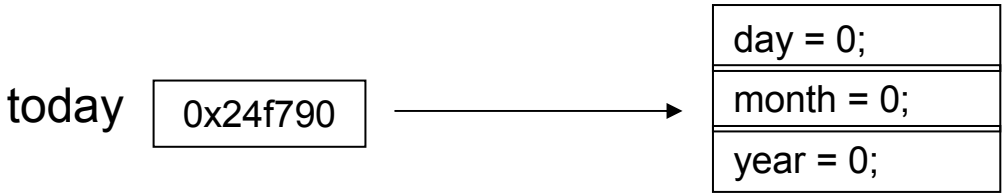
```
MyDate today;     // define variable tipo MyDate
```

Creación e inicialización de objetos (3).

- Caso 1, constructor sin argumentos:

  today = new MyDate();

- new consigue la memoria y regresa la dirección.
- se inicializan las variables de instancia con valores de default.



- se inicializa el objeto con los valores de default, definidos en la clase.

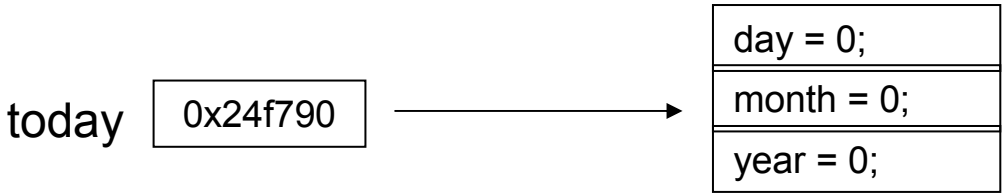
- se ejecuta el constructor (en el ejemplo, no hace nada).

Creación e inicialización de objetos (4).

- Caso 2, constructor con argumentos:

  today = new MyDate(31, 12, 2003);

- new consigue la memoria y regresa la dirección.
- se inicializan las variables de instancia con valores de default.

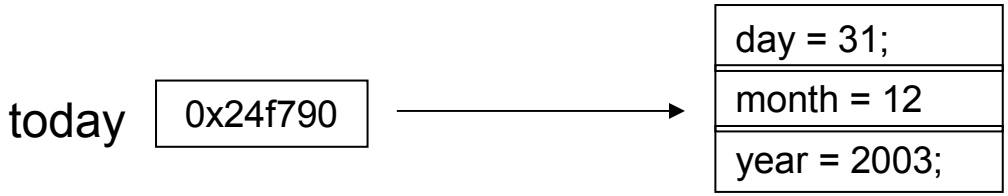


- se inicializa el objeto con los valores de default, definidos en la clase.

... continúa en la siguiente página

Creación e inicialización de objetos (5).

- Se ejecuta el constructor que inicializa el objeto con los valores pasados como parámetros.



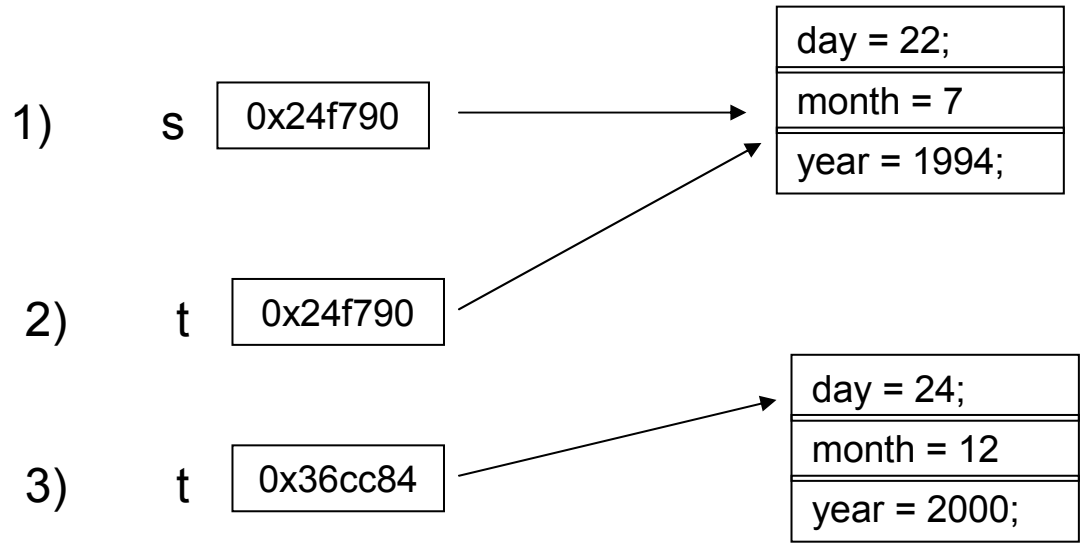
- Se puede declarar la variable y construir el objeto en un solo postulado:

```
MyDate today = new MyDate(31,12,2003);
```

Asignación de variables de referencia.

- Considere el siguiente fragmento de código:

```
1)  MyDate s = new MyDate(22, 7, 1994);
2 ) MyDate t = s;
3)  t = new MyDate(24, 12, 2000);
```



Pase de argumentos.

- Java sólo pasa argumentos a métodos por valor.
- Esto significa que en realidad se pasa una copia de los argumentos al método llamado.
- En el caso de argumentos primitivos, el método llamado recibe una copia del argumento y no puede modificar el valor original, sólo la copia.
- En el caso de variables de referencia, el valor que se pasa es la referencia al objeto.
- No se puede modificar el valor de la referencia, pero sí se puede modificar el contenido del objeto original.

Pase de argumentos (2).

```
public class PassTest {
      // Methods to change the current values
       public static void changeInt(int value) {
       value = 55;
      }
      public static void changeObjectRef(MyDate ref) {
          ref = new MyDate(1, 1, 2000);
      }
      public static void changeObjectAttr(MyDate ref) {
       ref.setDay(4);
      }

      public static void main(String args[]) {
          MyDate date;
          int val;
          // Assign the int
          val = 11;
```

... continúa en la siguiente página

Pase de argumentos (3).

```
        // Try to change it
        changeInt(val);
        // What is the current value?
        System.out.println("Int value is: " + val);
        // Assign the date
        date = new MyDate(22, 7, 1964);
        // Try to change it
         changeObjectRef(date);
        // What is the current value?
         date.print();
        // Now change the day attribute
        // through the object reference
        changeObjectAttr(date);
        // What is the current value?
        date.print();
         }
}
```

La referencia this.

- Todos los objetos tienen una referencia llamada *this* que apunta al objeto en cuestión.
- Puede utilizarse principalmente para:
  - Distinguir entre nombres de atributos y de argumentos dentro de un objeto.
  - Pasar el objeto actual como parámetro a otro método o constructor.

La referencia this (2).

```
public class MyDate {
      private int day = 1;
      private int month = 1;
      private int year = 2000;

      public MyDate(int day, int month, int year) {
                  this.day = day;
                  this.month = month;
                  this.year = year;
      }

       public MyDate(MyDate date) {
                   this.day = date.day;
                   this.month = date.month;
                   this.year = date.year;
}
```

La referencia this (3).

```
    public MyDate addDays(int moreDays) {
            MyDate newDate = new MyDate(this);
            newDate.day = newDate.day + moreDays;
            // código para checar cambio de mes, etc.
            return newDate;
    }

    public void print() {
            System.out.println("MyDate: " + day + "-" + month + "-" +
                                    year);
    }

}
```

La referencia this (4).

```
public class TestMyDate {

    public static void main(String[ ] args) {
            MyDate myBirth = new MyDate(22, 7, 1964);
            MyDate theNextWeek = myBirth.addDays(7);
            theNextWeek.print();
    }

}
```

Convenciones de nombres.

- Packages:

  package banking.domain;

- Clases:

  class SavingsAccount

- Interfaces:

  interface Account

- Methods:

  balanceAccount()

Convenciones de nombres (2).

- Variables:

  currentCustomer

- Constantes:

  HEAD_COUNT

  MAXIMUM_SIZE

Ejercicios.

1. Investigar asignación de referencias.

2. Crear clientes en el Sistema Bancario.

Repaso.

- Comentarios.
- Identificadores válidos e inválidos.
- Palabras claves de Java
- Los 8 tipos primitivos.
- Valores literales de los
- Variables primitivas y de referencia.
- Declarar variables de tipo clase.
- Construir objetos utilizando new.
- Inicialización de defaults.
- Importancia de las variables de referencia.
- Consecuencias de la asignación de variables de referencia.
- Convenciones de nombres.

Módulo 4.

Expresiones y Estructura del programa.

Objetivos.

- Distinguir entre variables locales y variables de instancia.
- Reconocer, describir y utilizar los principales operadores.
- Distinguir entre asignaciones legales e ilegales de primitivos.
- Identificar expresiones booleanas y sus requerimientos en las estructuras de control.
- Entender las asignaciones compatibles y el *casting* en tipos primitivos.
- Utilizar las estructuras de control if, switch, while y do.
- Usar los postulados break y continue en las estructuras del programa.

Variables locales y de instancia.

- Las variables de instancia son los atributos de la clase, que definen las características del objeto.

- Las variables locales son las que se definen dentro de métodos o constructores.

  – También se llaman automáticas, temporales o de stack.
  – Se crean cuando se ejecuta el método y se destruyen cuando termina su ejecución.
  – Sólo son visibles dentro del método en que se definen.
  – Deben ser inicializadas antes de poderse utilizar. No tienen valores de default.
  – No tienen modificadores de acceso.
  – Los argumentos de los métodos son un tipo de variable local.

Scope de variables.

```
public class ScopeExample {
      private int i=1;
      public void firstMethod() {
          int i=4, j=5;
          this.i = i + j;
          secondMethod(7);
      }
      public void secondMethod(int i) {
          int j=8;
          this.i = i + j;
      }
}

public class TestScoping {
      public static void main(String[] args) {
       ScopeExample scope = new ScopeExample();
       scope.firstMethod();
      }
}
```

Operadores.

Unarios:	++	--	+	-	~	!	(dataType)
Aritméticos:	*	/	%	+	-		
Corrimientos:	<<	>>	>>>				
Comparación:	<	>	<=	>=	instanceof		
	==	!=					
Bitwise:	&	^	|				
Short-Circuit	&&	||					
Ternario:	?:						
Asignación:	=	*=	/=	%=	+=	-=	<<=
	>>=	>>>=	&=	^=	|=		

Short-Circuit.

- Los operadores & y | evalúan ambos operandos siempre.
- Los operadores && y || evalúan el operador izquierdo primero y el segundo sólo en caso necesario.
- Ejemplo:

```
MyDate d;
if ((d != null) && (d.day > 31)) {
    // do something with d
}
```

Corrimientos a la derecha.

- Corrimiento *aritmético* (>>):
  128 >> 1 regresa $128/2^1 = 64$
  256 >> 4 regresa $256/2^4 = 16$
  -256 >> 4 regresa $-256/2^4 = -16$
  – El bit de signo se copia en el corrimiento.

- Corrimiento *lógico* (>>>):
  256 >>> 4 regresa $256/2^4 = 16$
  -256 >>> 4 regresa 4080 (?)
  – Siempre inserta ceros.
  – Se usa para analizar patrones de bits.

- Hacia la izquierda no hay problema

Ejemplos de corrimientos.

1357	=	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1
-1357	=	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1
1357 >> 5	=	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
-1357 >> 5	=	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1
1357 >>> 5	=	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
-1357 >>> 5	=	0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1
1357 << 5	=	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 1 0 1 0 0 0 0 0
-1357 << 5	=	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 0 0 1 1 0 0 0 0 0

Concatenación de Strings con +.

- El operador + aplicado a operandos tipo String, produce la concatenación de ambos Strings.
- Si un operando no es String, se convierte automáticamente a String y se concatena con el otro.
- Un operando tiene que ser String necesariamente para que haya concatenación.
- Ejemplo:

```
String salutation = "Dr.";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```

Casting.

- Si hay posibilidad de pérdida de información en una asignación, se tiene que usar el operador de Casting.

- Ejemplos:

```
long bigValue = 99L;
int squashed = bigValue;        // inválido, error de compilación.
int squashed = (int) bigValue; // OK
int squashed = 99L;             // inválido, error de compilación.
int squashed = (int) 99L;       // OK, pero …
int squashed = 99;              // OK, int es el default
```

Promoción.

- En operaciones aritméticas, los operandos se promueven automáticamente a la forma más larga. (byte y short siempre se promueven a int).

- Ejemplo:

```
int x, y, z;  long X;
x + y + z;      // se suman como int los 3 operandos
X + y + z;      // y, z se convierten a long y el resultado es long
```

Postulado if.

- Sintaxis 1

```
if ( booleanExpr) {
   statement or block;
 }
```

- Sintaxis 2

```
 if ( booleanExpr) {
      statement or block;
 } else if ( boolean expression) {
      statement or block;
 } else {
      statement or block;
 }
```

Ejemplo de if.

```
int count;
count = getCount();       // un método definido en el programa
if (count < 0) {
   System.out.println("Error: count es negativo.");
} else if (count > getMaxCount()) {
   System.out.println("Error: count excede el máximo permitido.");
} else {
   System.out.println(“Habrá " + count + " personas en la comida.");
}
```

Postulado switch.

```
switch (expr) {
case constant2:
    // statements;
break;
case constant3:
    // statements;
break;
default:
    // statements;
break;
}
```

Nota. expr debe ser de tipo byte, short, char o int.

Ejemplo switch (1).

```
switch ( carModel ) {
    case DELUXE:
        addAirConditioning();
        addRadio();
        addWheels();
        addEngine();
        break;
    case STANDARD:
        addRadio();
        addWheels();
        addEngine();
        break;
    default:
        addWheels();
        addEngine();
}
```

Ejemplo switch (2).

```
switch ( carModel ) {
    case THE_WORKS:
        addGoldPackage();
        add7WayAdjustableSeats();
    case DELUXE:
        addFloorMats();
        addAirConditioning();
    case STANDARD:
        addRadio();
        addDefroster();
    default:
        addWheels();
        addEngine();
}
```

Postulado for.

- Sintaxis

```
for (initExpr; booleanTestExpr; alterExpr) {
    statement or block;
}
```

- Ejemplo:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
```

Nueva versión del for.

- Introducido en java 1.5.
- Conocido como for-each en otros lenguajes.
- Sintaxis

```
for (tipo varIteracion: coleccion) {
    statement or block;
}
```

- Ejemplo:

```
int nums[ ] = {1,2,3,4,5,6,7,8,9,10};
int sum = 0;
for (int x: nums) {
    sum += x;
}
```

Postulado while.

- Sintaxis

```
while (booleanExpr) {
   statement or block;
}
```

- Ejemplo:

```
int i = 0;
while (i < 10) {
   System.out.println("Are you finished yet?");
   i++;
}
System.out.println("Done");
```

Postulado do.

- Sintaxis

```
do {
   statement or block;
} while (booleanExpr);
```

- Ejemplo:

```
int i = 0;
do {
   System.out.println("Are you finished yet?");
   i++;
} while (i < 10);
System.out.println("Done");
```

break y continue.

- break

```
do {
  statement;
  if (booleanExpr) {
     break;
  }
  statement;
} while (booleanExpr);
```

- Se sale del do.

- continue:

```
do {
statement;
if ( booleanExpr) {
continue;
}
statement;
} while (booleanExpr);
```

- Se sale del if, continúa en el do..

break con etiquetas.

```
outer:
    do {
      statement;
      do {
           statement;
           if (booleanExpr) {
              break outer;
           }
      statement;
      } while (booleanExpr);
      statement;
    } while (booleanExpr);
```

continue con etiquetas.

```
test:
    do {
        statement;
        do {
            statement;
            if (booleanExpr) {
               continue test;
            }
            statement;
        } while ( booleanExpr);
    statement;
    } while (booleanExpr);
```

Ejercicios.

1. Utilización de estructuras de control.

2. Modificaciones a métodos del Sistema Bancario.

3. Uso de ciclos anidados.

Repaso.

- Variables de instancia y variables locales.
- Inicialización de variables locales.
- Operadores de Java.
- Asignaciones legales e ilegales de tipos primitivos.
- Expresiones booleanas en estructuras de control.
- Compatibilidad de asignaciones y casting.
- Promociones de tipos
- Uso de if, switch, for, while, and do.
- Uso de break y continue con etiquetas.

Módulo 5.

Arreglos.

Objetivos.

- Declarar y construir arreglos de tipos primitivos o de objetos.
- Explicar como se inicializan los elementos de un arreglo.
- Determinar el número de elementos de un arreglo.
- Crear arreglos multidimensionales.
- Utilizar métodos para copiar valores de un arreglo a otro.

Declaración de arreglos.

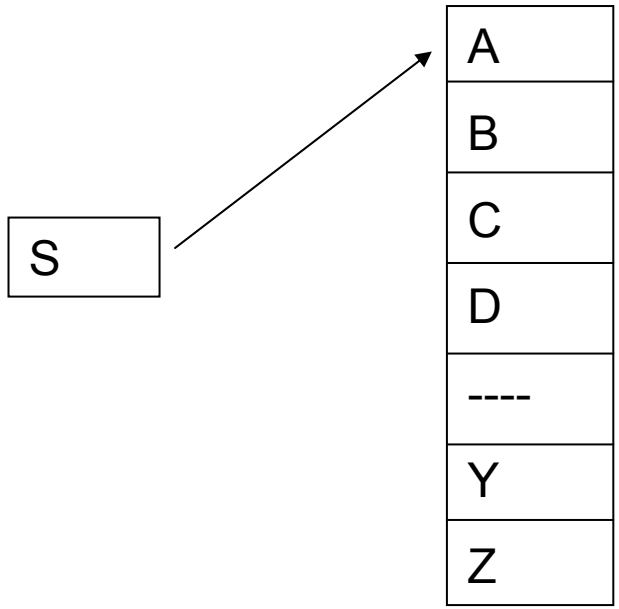
- Arreglos son colecciones de objetos del mismo tipo.
- Dos maneras (equivalentes) de declarar arreglos:

```
char s[ ];          char[ ] s;
Point p[ ];         Point[ ] p;
```

- Se crea solamente la variable de referencia.
- Un arreglo es un objeto, se debe crear con new.

Creación de un arreglo de primitivos.

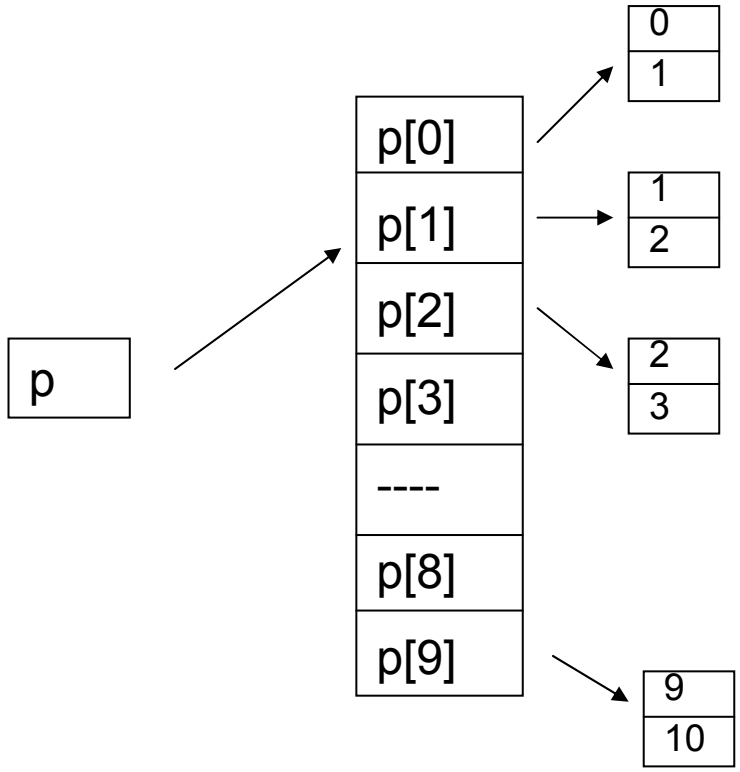
```
public char[ ] createArray() {
    char[ ] s;
    s = new char[26];
    for (int i=0; i<26; i++ ) {
       s[i] = (char) ('A' + i);
    }
    return s;
}
```



Creación de un arreglo de objetos.

```
public Point[ ] createArray() {
     Point[ ] p;
     p = new Point[10];
     for (int i=0; i<10; i++ ) {
        p[i] = new Point(i, i+1);
     }
     return p;
}
```

```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```



Formas de inicialización.

- Forma normal:

```
String[ ] names;
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

- Forma abreviada:

```
String[ ] names = {
"Georgianna",
"Jen",
"Simon"
}
```

- Forma normal:

```
MyDate[ ] dates;
 dates = new MyDate[3];
 dates[0] = newMyDate(22,7,1964);
 dates[1] = new MyDate(1, 1,2000);
 dates[2] = newMyDate(22,12,1964);
```

- Forma abreviada:

```
MyDate[ ] dates = {
  new MyDate(22, 7, 1964),
  new MyDate(1, 1, 2000),
  new MyDate(22, 12, 1964)
}
```

Formas de inicialización (2)

- Paso a paso:

```
int[ ]  y ;
y = new  int[3] ;

y[0] = 2; y[1] = 4; y[2] = 5;
```

- Abreviado:

```
int[ ]  y = {2,4,5};
```

- También válido:

```
int[ ]  y = new  int[ ] {2,4,5};
```

Arreglos Multidimensionales.

- Arreglos de arreglos:

```
int[ ][ ] twoDim = new int [3][ ];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
twoDim[2] = new int[5];
```

- Sin embargo:

```
int [ ][ ] twoDim = new int [ ][4];     // es inválido
```

Arreglos Multidimensionales (2).

- Arreglo de arreglos no rectangulares:

```
int[ ][ ] twoDim = new int [4][ ];
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

- Arreglo de cuatro arreglos de cinco enteros cada uno:

```
int twoDim[][] = new int[4][5];
```

Longitud de los arreglos.

```
int[ ] list1 = new int[10];
for (int i = 0; i < list1.length; i++) {
      System.out.println(list[i]);
}
```

- ¿Qué pasa si se excede el índice? e.g.

```
System.out.println(list[20]);
```

- Excepción: ArrayIndexOutOfBoundsException.

Ejemplo usando for-each.

```
public class printArray {
   public static void main(String [] args) {
      int[ ] primos = { 1, 2, 3, 5, 7, 11, 13, 17 };
      for (int elemento: primos) {
         System.out.println(elemento);
      }
   }
}
```

Inmutabilidad de los arreglos.

- No se puede cambiar el tamaño de un arreglo una vez creado.

- Se puede usar la misma variable de referencia para un nuevo arreglo:

```
int myArray[] = new int[6];
myArray = new int[10];
```

Copia de arreglos.

- Método System.arraycopy.

```
// arreglo original
int[ ] elements  = { 1, 2, 3, 4, 5, 6 };

// nuevo arreglo más grande
int [ ] hold = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copia todos los elementos del arreglo elements
// empezando con el elemento 0, al arreglo hold.

System.arraycopy(elements, 0, hold, 0, elements.length);
```

Ejercicios.

1. Manipulación de arreglos.

2. Uso de arreglos para representar múltiples clientes en el Sistema Bancario.

Repaso.

- Declaración y creación de arreglos de primitivos y de objetos.
- Inicialización de los elementos de un arreglo.
- Determinación del número de elementos en un arreglo.
- Creación de arreglos multidimensionales.
- Copia de elementos de arreglos.

Módulo 6.

Diseño de clases: herencia, polimorfismo y otras funciones.

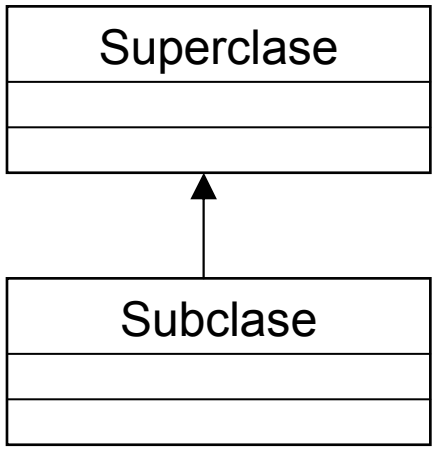
Objetivos.

- Comprender los conceptos de *herencia* y *polimorfismo*.
- Utilizar los modificadores de acceso protected y "*friendly*".
- Entender el proceso completo de creación e inicialización de objetos.
- Sobrecargar métodos y constructores.
- Utilizar la palabra clave *this* para llamar a otros constructores.
- Substituir métodos.
- Invocar métodos de superclases.
- Invocar constructores de superclases.

Herencia.

- Mecanismo para extender clases.
- Una clase adquiere o hereda los atributos y métodos de otra.
- La clase que hereda se llama subclase.
- La clase base se llama superclase.
- Prueba de validez:

"Un objeto de tipo subclase debe ser también de tipo superclase."



Herencia (2).

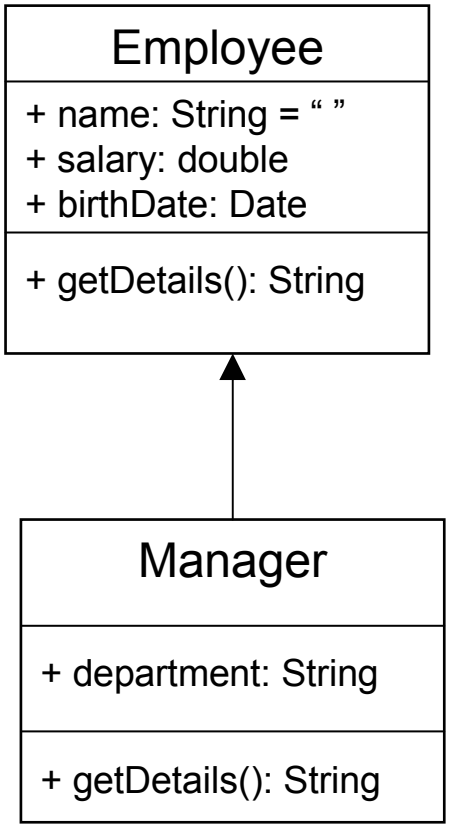
```
public class Employee {
    public String name = " ";
    public double salary;
    public Date birthDate;
    public String getDetails()
      { // despliega info Employee }
}
```

```
public class Manager {
    public String name = " ";
    public double salary;
    public Date birthDate;
    public String department;
    public String getDetails()
      { // despliega info Manager }
}
```

- Atributos y métodos duplicados.

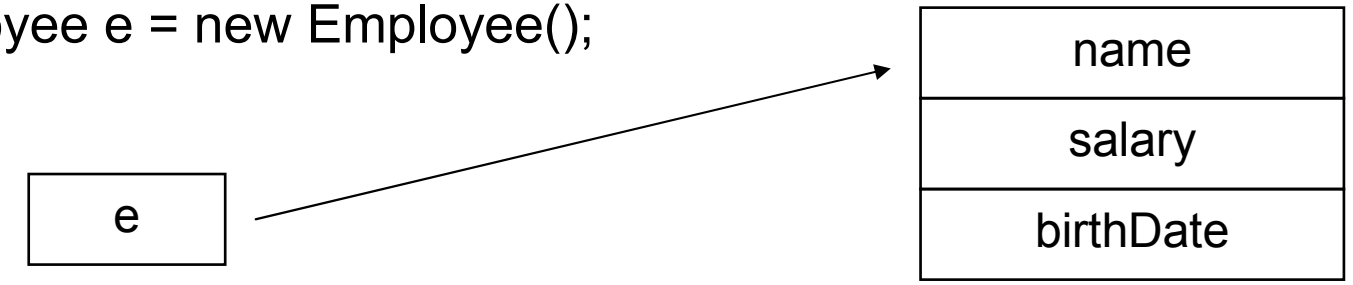
Herencia (3).

```
public class Employee {
      public String name = “ ”;
      public double salary;
      public Date birthDate;
      public String getDetails()
       {// despliega info Employee}
}

public class Manager  extends Employee {
      public String department;
      public String getDetails()
       {// despliega info Manager}
}
```

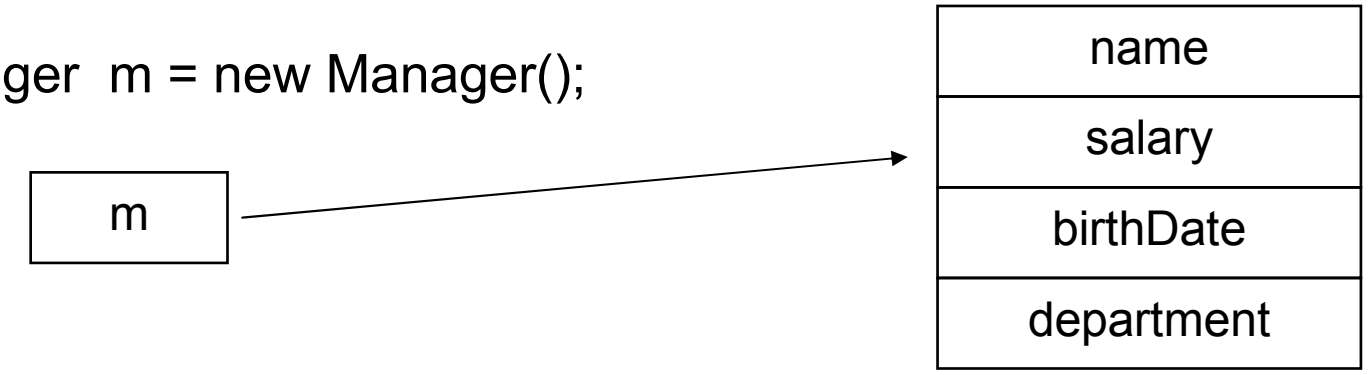


Herencia (4).

Employee e = new Employee();

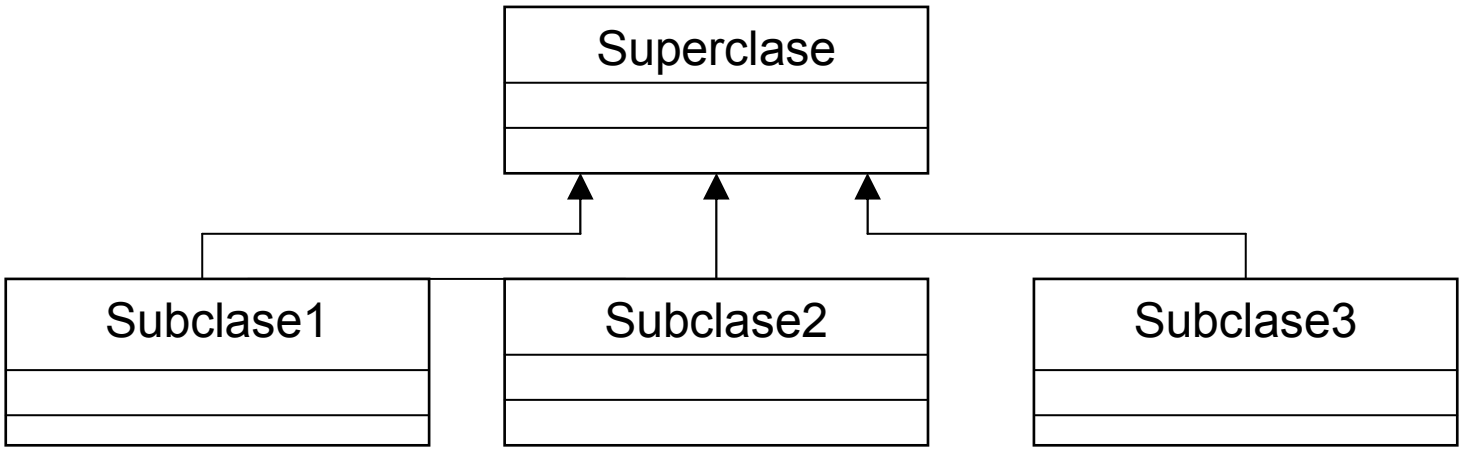


Manager m = new Manager();

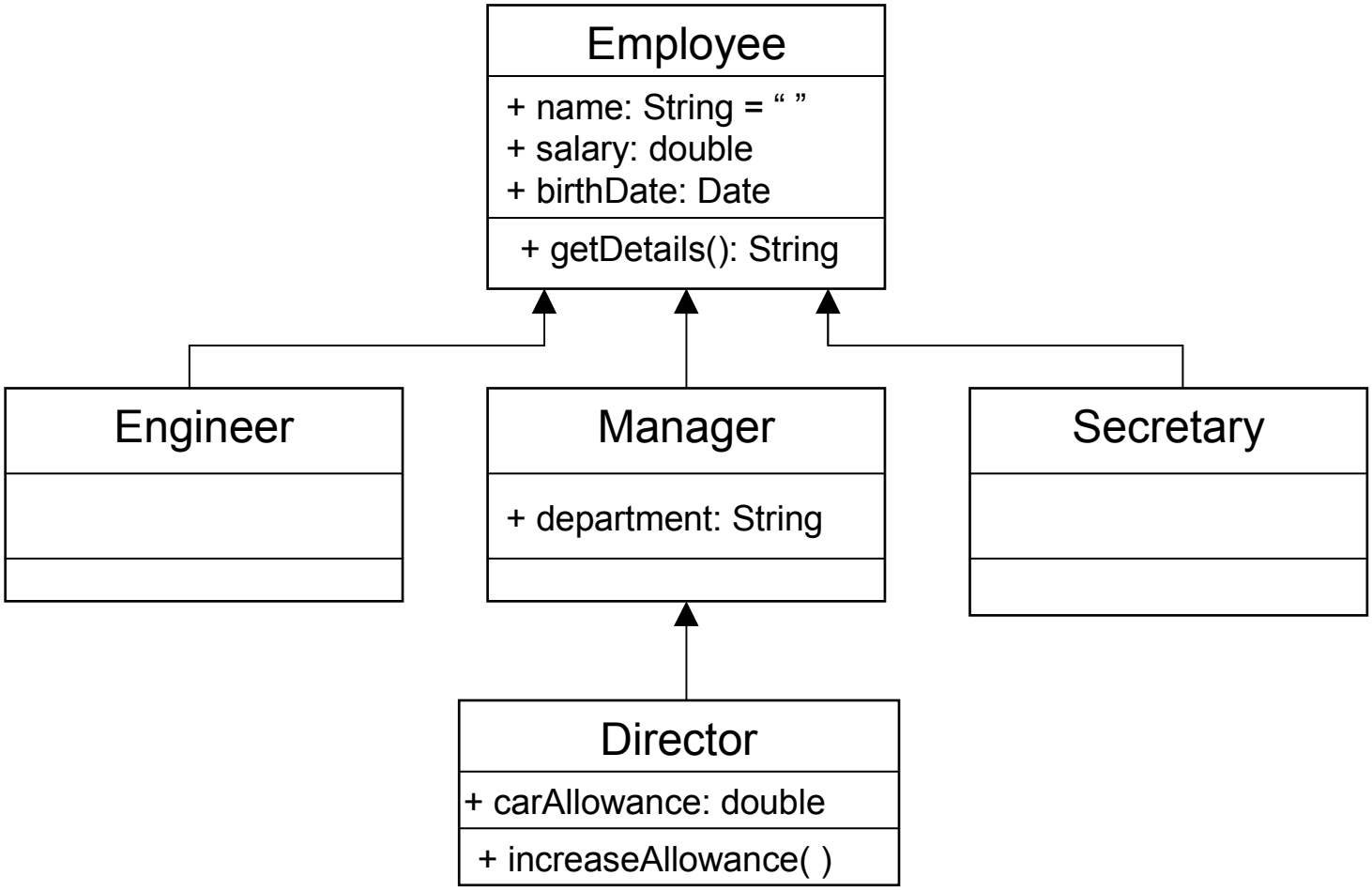


Herencia Simple.

- Una subclase sólo hereda de una superclase.
- Única forma permitida en Java.
- Hace el código más simple.
- Las interfaces proporcionan las ventajas de la herencia múltiple sin sus desventajas.



Herencia Simple (ejemplo).



Modificadores de acceso.

Modificador	Misma clase	Mismo paquete	subclase	programa
private	sí	no	no	no
default *"friendly"*	sí	sí	no	no
protected	sí	sí	sí	no
public	sí	sí	sí	sí

Substitución de métodos.

- Una subclase puede modificar el comportamiento heredado de la superclase, substituyendo (*overriding*) métodos.
- La subclase crea un método funcionalmente diferente, pero con el mismo:
  - Nombre
  - Tipo de retorno
  - Lista de Argumentos

Substitución de métodos (2).

```
public class Employee {
      protected String name;
      protected double salary;
      protected Date birthDate;
      public String getDetails() {
            return “Name: “ + name + “\n” + “Salary: “ + salary;
      }
}

public class Manager extends Employee {
      protected String department;
      public String getDetails() {
            return “Name: “ + name + “\n” + “Salary: “ + salary + "\n" +
                  “Manager of: “ + department;
      }
}
```

Substitución de métodos (3).

- Reglas para la substitución:
  - Ambos métodos deben tener nombre, tipo de retorno* y lista de argumentos idénticos.
  - El método de la subclase no puede ser menos accesible que el de la superclase.

* A partir de la versión 1.5 el tipo de retorno del método substituyente puede ser una subclase del tipo de retorno del método substuido.

Substitución de métodos (4).

```
public class Parent {
     public void doSomething() { }
}

public class Child extends Parent {
     private void doSomething() { }       // inválido, doSomething de Child es menos
                                          // accesible que doSomething de Parent
}

public class UseBoth {
     public void doOtherThing() {
        Parent p1 = new Parent();
        Parent p2 = new Child();
           p1.doSomething();
           p2.doSomething();       // inválido, doSomething de Child no es visible
}
}
```

La palabra clave *super.*

- *super* se utiliza en una clase para hacer referencia a miembros (atributos y métodos) de una superclase
- No necesariamente la superclase inmediata superior.
- También para constructores.

La palabra clave *super* (2).

```
public class Employee {
     private String name;
     private double salary;
     private Date birthDate;
     public String getDetails() {
        return "Name: " + name + "\nSalary: " + salary;
     }
}

public class Manager extends Employee {
     private String department;
     public String getDetails() {
     return super.getDetails() + "\nDepartment: " + department;
     }
}
```

Polimorfismo.

- *Polimorfismo* es la habilidad de ejecutar diferentes métodos basándose en el tipo real del objeto en cuestión.
- Válido solamente en objetos relacionados mediante herencia.
- Los objetos NO son polimórficos.
- Las variables de referencia son polimórficas porque pueden apuntar a objetos de diferentes clases.

Polimorfismo (2).

```
Employee e = new Employee();
e.getDetails();          // ejecuta getDetails() de Employee.

Manager m = new Manager();
m.getDetails();          // ejecuta getDetails() de Manager.

e = new Manager();       // válido por la relación de herencia
e.getDetails();          // ejecuta getDetails() de Manager.
```

Nota. e.department = “Sales” es inválido

Colecciones heterogéneas.

- Colecciones de objetos del mismo tipo se llaman colecciones homogéneas.

```
MyDate[ ] dates = new MyDate[2];
dates[0] = new MyDate(22, 12, 1964);
dates[1] = new MyDate(22, 7, 1964);
```

- Colecciones de objetos de diferente tipo se llaman colecciones heterogéneas.

- El polimorfismo permite colecciones heterogéneas de objetos relacionados por herencia.

```
Employee [ ] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

Argumentos polimórficos.

- Manager es subclase de Employee *(Un manager **es** un Employee).*

```
// en la clase Employee

        public TaxRate findTaxRate(Employee e) { … }

// en otra clase

        Manager m = new Manager();
        TaxRate t = findTaxRate(m);
```

El operador *instanceof*.

```
public class Employee { … }
public class Manager extends Employee { … }
public class Engineer extends Employee { … }
```

```
public void doSomething(Employee e) {
   if (e instanceof Manager) {
      // procesa Manager
   } else if (e instanceof Engineer) {
      // procesa Engineer
   } else {
      // procesa otro tipo de Empleado
    }
}
```

Casting de objetos.

- Usar **instanceof** para probar el tipo del objeto.
- Restaurar funcionalidad mediante casting:

```
Manager m = (Manager)e;
```

- Reglas para validez:
  - Casting “hacia arriba” en la jerarquía es implícito.
  - Casting “hacia abajo” debe hacerse en forma explícita y es checado por el compilador.
  - Sólo es válido entre objetos relacionados por herencia.
  - Aunque compile correctamente, puede haber errores en tiempo de ejecución. (¡usar instanceof!).

Sobrecarga de métodos.

- Consiste en varios métodos con el mismo nombre, pero:
  - La lista de argumentos ***debe*** ser diferente en número y/o tipo de argumentos.
  - El tipo de retorno ***puede*** ser diferente.
- Conocido también como *polimorfismo pasivo.*

```
public void println(int i)
public void println(float f)
public void println(String s)
```

Métodos con argumentos variables.

- Forma tradicional

```
public class Statistics {
   public float average(int x1, int x2) {  }
   public float average(int x1, int x2, int x3) {  }
   public float average(int x1, int x2, int x3, int x4) {  }
}
```

- Los métodos se invocan de la siguiente manera:

```
Statistics stats = new Statistics( );
float prom1 = stats.average(4, 5, 6);
float prom2 = stats.average (4, 5, 6, 7);
```

Métodos con argumentos variables (2).

- En la versión 1.5

```
public class Statistics {
   public float average(int … nums) {
      int sum = 0;
      for(int x: nums) {
         sum += x;
         }
   return ((float) sum) / nums.length;
}
```

- Los métodos se invocan de la misma manera:

```
Statistics stats = new Statistics( );
float prom1 = stats.average(4, 5, 6);
float prom2 = stats.average (4, 5, 6, 7);
```

Sobrecarga de constructores.

- Igual que los métodos, los constructores pueden ser sobrecargados.

- Ejemplo:

```
public Employee(String name, double salary, Date DoB)
public Employee(String name, double salary)
public Employee(String name, Date DoB)
```

- Se puede usar la referencia this en la primera línea de un constructor para llamar a otro constructor

Sobrecarga de constructores (2).

```
public class Employee {
    private static final double BASE_SALARY = 15000.00;
    private String name;
    private double salary;
    private Date birthDate;
    public Employee(String name, double salary, Date DoB) {
        this.name = name;
        this.salary = salary;
        this.birthDate = DoB;
    }
    public Employee(String name, double salary) {
        this(name, salary, null);
    }
    public Employee(String name, Date DoB) {
        this(name, BASE_SALARY, DoB);
    }
    public Employee(String name) {
        this(name, BASE_SALARY);
    }
}
```

Invocación de constructores.

- Para invocar al constructor de la superclase, se debe colocar un postulado super en la primera línea del constructor de la subclase.
- Se puede invocar un constructor en particular utilizando diferente lista de argumentos en super.
- Si no aparece super como primera línea, el compilador inserta una llamada al constructor sin argumentos de la superclase.
- Si la superclase no tiene constructor sin argumentos, marca error de compilación.

Invocación de constructores (2).

```
public class Employee {
    private static final double BASE_SALARY = 15000.00;
    private String name;
    private double salary;
    private Date birthDate;
    public Employee(String name, double salary, Date DoB) {
        this.name = name;
        this.salary = salary;
        this.birthDate = DoB;
    }
    public Employee(String name, double salary) {
        this(name, salary, null);
    }
    public Employee(String name, Date DoB) {
        this(name, BASE_SALARY, DoB);
    }
    public Employee(String name) {
        this(name, BASE_SALARY);
    }
}
```

Invocación de constructores (3).

```
public class Manager extends Employee {
    private String department;

    public Manager(String name, double salary, String dept) {
       super(name, salary);
       department = dept;
    }
    public Manager(String n, String dept) {
       super(name);
       department = dept;
    }
    public Manager(String dept) {      // error, no super()
       department = dept;
    }
 }
```

Construcción e inicialización de objetos.

- New consigue la memoria y se lleva a cabo la inicialización de default de todas las variables de instancia, incluyendo las heredadas de las superclases.

- Se invocan los constructores, ejecutando los siguientes pasos, en forma recursiva:

  1. Se examinan los argumentos del constructor correspondiente.
  2. Si hay un this como primera línea, se llama al constructor correspondiente.
  3. Se llama recursivamente al constructor de la superclase implícita o explícitamente. (Excepto para la clase Object).
  4. Se ejecuta la inicialización explícita de las variables de instancia.
  5. Se ejecuta el cuerpo del constructor correspondiente.

Construcción e inicialización de objetos. (Ejemplo)

```
public class Object {
      public Object() { }
}
public class Employee extends Object {
      private String name;
      private double salary = 15000.00;
      private Date birthDate;
      public Employee(String n, Date DoB) {
          // implicit super();
          name = n;
          birthDate = DoB;
      }
      public Employee(String n) {
          this(n, null);
      }
}
public class Manager extends Employee {
      private String department;
      public Manager(String n, String d) {
      super(n);
      department = d;
      }
}
```

La Clase Object.

- La clase Object es la clase raiz de todas las clases en Java.
- Todas las clases descienden de Object.
- Una declaración de clase implícitamente incluye “extends Object”.

```
public class Employee {
    ...
}
```

es equivalente a:

```
public class Employee extends Object {
    ...
}
```

- Útil sobretodo por los métodos que contiene, los cuales pueden ser usados por cualquier clase.

El operador == y el método equals.

- El operador **==** aplicado a variables de referencia, determina si dos referencias tienen la misma dirección y por tanto apuntan al mismo objeto. (No muy útil).
- El método **equals** se usa para determinar si los objetos son iguales, según un criterio específico.
- La clase Object contiene un método **equals,** que se comporta igual que el operador ==.
- Se puede desarrollar un método **equals** en cada clase que necesite comparar objetos, el cual substituye al de la clase Object. (e.g. String incluye método equals).
- Si se implementa equals se recomienda implementar también el método hashCode.

Ejemplo del uso de equals.

```
public class MyDate {
      private int day;
      private int month;
      private int year;
      public MyDate(int day, int month, int year) {
          this.day = day;
          this.month = month;
          this.year = year;
       }
       public boolean equals(Object o) {
          boolean result = false;
          if ( (o != null) && (o instanceof MyDate) ) {
              MyDate d = (MyDate) o;
              if ( (day == d.day) && (month == d.month) && (year == d.year) ) {
                  result = true;
              }
          }
          return result;
       }
      public int hashCode() {
          return ( (new Integer(day).hashCode()) ^ (new Integer(month).hashCode())
          ^ (new Integer(year).hashCode()) );
      }
 }
```

Ejemplo del uso de equals (2).

```
public class TestEquals {
      public static void main(String[] args) {
          MyDate date1 = new MyDate(14, 3, 1976);
          MyDate date2 = new MyDate(14, 3, 1976);
          if (date1 == date2) {
                System.out.println("date1 is identical to date2");
           } else {
                System.out.println("date1 is not identical to date2");
          }

          if ( date1.equals(date2) ) {
              System.out.println("date1 is equal to date2");
           } else {
              System.out.println("date1 is not equal to date2");
          }
          System.out.println("set date2 = date1;");
          date2 = date1;
          if ( date1 == date2 ) {
          System.out.println("date1 is identical to date2");
          } else {
               System.out.println("date1 is not identical to date2");
          }
      }
}
```

El método toString.

- La clase Object contiene un método toString que convierte un objeto a String.
- El operador + lo invoca durante la concatenación de Strings.
- Se debe substituir (override) cuando se quiera desplegar información del contenido de un objeto en forma legible.
- Los tipos primitivos se convierten a String usando la implementación del método toString de las clases Wrapper.

Las Clases Wrapper.

- Son clases de la API que contienen un primitivo como atributo.
- Extremadamente útiles por los métodos que contienen.

Primitivo	Clase Wrapper
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Boxing y unboxing.

- Boxing es el proceso de cambiar un primitivo a su equivalente en un objeto de la clase wrapper correspondiente:

```
int num = 420;
Integer objNum = new Integer(num);
```

- Unboxing es el proceso contrario:

```
int n = objNum.intValue();
```

Autoboxing y autounboxing.

- Nueva funcionalidad en versión 1.5

```
int num = 420;
Integer objNum = num;

int n = objNum;
```

- Puede tener efectos adversos en el rendimiento.

Ejercicios.

1. Crear subclases de cuentas en el Sistema Bancario.

2. Crear una colección heterogénea para representar las cuentas de los clientes en el Sistema Bancario.

3. Crear subclases de cuentas utilizando un método más sofisticado.

Repaso.

- Herencia, polimorfismo, sobrecarga y substitución de métodos.
- Modificadores de acceso private, public, protected y “friendly”.
- Colecciones heterogéneas.
- Construcción e inicialización de objetos, tomando en cuenta herencia y constructores.
- Métodos equals y toString.
- Clases Wrapper.

Módulo 7.

Conceptos Avanzados de Clases.

Objetivos.

- Crear variables, métodos e inicializadores *estáticos*.
- Entender el patrón de diseño Singleton.
- Describir variables, métodos y clases *finales*.
- Utilizar enumeraciones (*enumerated types*).
- Usar el postulado *import static*.
- Explicar el concepto y utilidad de las clases *abstractas*.
- Entender el concepto de las *interfaces* de Java.
- Utilizar clases anidadas.

La palabra clave *static*.

- Hay variables estáticas y métodos estáticos.
- Significa que el atributo o método está asociado con la clase y no con una instancia en particular de la clase.
- Por ese motivo se les llama también variables de clase o métodos de clase.
- También existen inicializadores estáticos, imports estáticos* y clases anidadas estáticas.

(*) nuevos en versión 1.5

Variables estáticas.

```
public class Count {
    private int serialNumber;
    public static int counter = 0;       // variable estática counter
    public Count() {
        counter++;
        serialNumber = counter;
    }
}

public class OtherClass {
    public void incrementNumber() {
        Count.counter++;                 // se puede accesar vía el nombre de la Clase
    }
}
```

Métodos estáticos.

- Son métodos que pueden ser invocados sin necesidad de crear un objeto de la clase a la que pertenecen.
- Se invocan usando la notación de punto, con el nombre de la clase, en vez del objeto.
- No pueden usar variables de instancia, sólo variables estáticas.
- Son equivalentes a las funciones de uso general en otros lenguajes.
- La clase Math contiene métodos estáticos muy útiles.

Métodos estáticos (2).

```
public class HypotV14 {
    public static void main(String args[ ]) {
       double side1, side2;
       double hypot;
       side1 = 3.0;
       side2 = 4.0;
       hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
       System.out.println("Given sides of lengths " + side1 + " and " +
                          side2 + "  the hypotenuse is " +  hypot);
  }
}
```

Métodos estáticos (3).

```
public class Count2 {
    private int serialNumber;
    private static int counter = 0;
    public static int getTotalCount() {
        return counter;
    }

    public Count2() {
        counter++;
        serialNumber = counter;
    }
}
```

Métodos estáticos (4).

```
public class TestCounter {

    public static void main(String[ ] args) {
        System.out.println("Number of counter is “  + Count2.getTotalCount());
        Count2 count1 = new Count2();
        System.out.println( "Number of counter is " + Count2.getTotalCount());
    }

 }
```

La salida debe ser:

Number of counter is 0

Number of counter is 1

Métodos estáticos (5).

```
public class Count3 {
        private int serialNumber;
        private static int counter = 0;
        public static int getSerialNumber() {        /*  *  */
            return serialNumber;
        }
 }

/* Error de compilación:
   non-static variable serialNumber cannot be referenced from a static context
*/
```

Métodos estáticos (6).

- No se puede substituir (override) un método estático, pero se puede esconder.
- Dos métodos estáticos con la misma firma en una jerarquía de clases, simplemente significa que son dos métodos distintos. Siempre se ejecuta el que se invoca con el nombre de la clase correspondiente.
- El método main() es estático porque la máquina virtual de Java no crea una instancia de la clase cuando ejecuta el método main.
- Lo anterior implica que no se pueden accesar directamente desde el método main() las variables de instancia no estáticas, es necesario crear un objeto para accesarlas.

Inicializadores estáticos.

- Son bloques de código que no pertenecen a ningún método.
- Se ejecutan sólo una vez, en el momento en que se carga la clase.
- Sólo pueden accesar variables estáticas.
- Se usan normalmente para inicializar variables estáticas.

Inicializadores estáticos (2).

```
public class Count4 {
        public static int counter;

        static {
          counter = Integer.getInteger("static.counter").intValue();
        }
}

public class TestStaticInit {
        public static void main(String[ ] args) {
          System.out.println("counter = " + Count4.counter);
        }
}
```

Nota. static.counter es una propiedad cuyo valor se pasa al programa en tiempo de ejecución como por ejemplo:

java -Dstatic.counter=47 TestStaticInit

El Patrón de diseño Singleton.

- Aplicación de variables y métodos estáticos.
- Resuelve el problema de diseño siguiente:

  ¿Cómo garantizar que sólo pueda existir una instancia de determinado objeto?

- Muy útil en diversas aplicaciones.
- La solución consiste en:
  - Hacer privado el constructor de la clase.
  - Declarar la instancia única como una variable estática.
  - Crear la instancia única en un inicializador estático.
  - Crear un método estático que regrese la instancia única, el cual substituye de hecho a la función new para esa clase.
  - New no se puede usar porque el constructor es privado.

Ejemplo de Singleton.

```
public class Bank {
      private static Bank instance;
      private String name;
      private Customer[ ] customers;
      static {
        instance = new Bank( );
      }
      public static Bank getBank() {
      return instance;
      }
      private Bank() {
        // constructor code
      }
      // other class code
}
```

Ejemplo de Singleton (2).

```
public class SavingsAccount  extends Account {

        public void withdraw(float amount) {

        Bank bank = Bank.getBank( );

        // use Bank object to retrieve customers, etc.

        }
}
```

La palabra clave *final*.

- Una clase *final* no admite subclases.

- Un método *final* no puede ser substituído (override).

- Una variable *final* es una constante.

- Variables finales en blanco ("blank final variables):

  - **Se declaran como final sin inicializar.**
  - **Se deben inicializar antes de utilizarse.**
  - **Una vez inicializadas no se puede cambiar su valor.**
  - **Si es variable de instancia, se debe inicializar en un constructor.**
  - **Si es variable local, se debe inicializar en el método donde se definió.**

Variables finales.

- Constantes:

```
public class Bank {
   private static final double DEFAULT_INTEREST_RATE=3.2;
   // more declarations
}
```

- **Blank Final Variables:**

```
public class Customer {
     private final long customerID;
     public Customer() {
        customerID = createID();
     }
     public long getID() {
        return customerID;
     }
      private long createID() {
        return ...        // generate new ID
     }
 }
```

Enumeraciones (enumerated types).

- Son conjuntos de nombres simbólicos que representan los valores de un atributo.
- Antes de la versión 1.5 se simulaban utilizando constantes independientes. (Era necesario validar tipos.)
- En versión 1.5 se introduce la palabra clave ***enum*** para crear enumeraciones.
- No confundir con la interfaz java.util.Enumeration, que es un tipo de colección.

Ejemplo de *enum*.

```
package enumPack;
public enum Suit {
   SPADES    ("Spades"),
   HEARTS    ("Hearts"),
   CLUBS     ("Clubs"),
   DIAMONDS  ("Diamonds");

   private final String name;

   private Suit(String name) {
      this.name = name;
   }
   public String getName() {
      return name;
   }
}
```

Ejemplo de *enum* (2).

```
package enumPack;
public class PlayingCard {
   private Suit suit;
   private int rank;
   public PlayingCard(Suit suit, int rank) {
      this.suit = suit;
      this.rank = rank;
   }
   public Suit getSuit() {
      return suit;
   }
   public int getRank() {
      return rank;
   }
}
```

Ejemplo de *enum* (3).

```
import enumPack.*;
public class TestPlayingCard {

   public static void main(String[ ] args) {

      PlayingCard card1 = new PlayingCard(Suit.SPADES, 2);
      System.out.println("card1 is the " + card1.getRank() +
         " of " + card1.getSuit().getName() );

      // Next statement will not compile...
      // PlayingCard card2 = new PlayingCard(47, 2);
   }
}
```

Import estático.

- Permite importar miembros estáticos de una clase, interfaz o *enum.*
- No es necesario calificarlos con el nombre de la clase, interfaz o *enum*.
- Hay que usarlo con cuidado, porque puede hacer el código poco claro, sobre todo si se usa con la opción *.

Import estático (2).

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

public class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;
    side1 = 3.0;
    side2 = 4.0;
    // Here, sqrt() and pow() can be called by themselves,
    // without their class name.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));
    System.out.println("Given sides of lengths " + side1 + " and " +
            side2 +  " the hypotenuse is " +  hypot);
  }
}
```

Import estático (3).

```
import enumPack.*;
import static enumPack.Suit.*;

public class TestPlayingCard2 {

    public static void main(String[ ] args) {

    PlayingCard card1 = new PlayingCard(SPADES, 2);
    System.out.println("card1 is the " + card1.getRank() +
                       " of " + card1.getSuit().getName() );
  }
}
```

Clases abstractas.

- Modelan una clase de objetos donde la implementación completa no se conoce, sino que es proporcionada por subclases concretas.
- No pueden ser instanciadas.
- Su finalidad es que existan subclases de ellas.
- Sirven para agrupar atributos comunes en superclases que no se deben instanciar.
- Pueden contener métodos abstractos y no abstractos.

Métodos abstractos.

- Métodos declarados con el modificador *abstract* en una clase abstracta.
- No contienen cuerpo, sólo definición.
- Deben ser implementados por una subclase de la clase abstracta.

Ejemplo de clase abstracta.

```
public abstract class Account {
    protected double balance;
    protected Account(double balance) {
        this.balance = balance;
    }
    public abstract boolean deposit(double amount);
    public abstract boolean withdraw(double amount);
    public double getBalance() {
       return balance;
    }
}
```

Interfaces.

- Una interfaz es una clase abstracta llevada al extremo.
- Permite establecer la forma de la clase: nombres de métodos, listas de argumentos y tipos de retorno, pero no cuerpos.
- Puede contener sólo métodos abstractos y variables estáticas finales.
- Una interfaz dice como deben ser las clases que la implementen, pero no dice como se deben implementar.

Interfaces (2).

- Se definen como clases pero con la palabra clave *interface* en vez de *class*.
- Las clases que implementan la interfaz utilizan *implements* en vez de extends.
- Una clase puede implementar varias interfaces, simulando herencia múltiple en Java.

Interfaces (3).

```
public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}

public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
        // lower landing gear
        // deccelerate and lower flaps until touch-down
        // apply breaks
    }
    public void fly() {
        // keep those engines running
    }
}
```

Interfaces (4).

```
public class Animal {
    public void eat( )    { // general eating implementation }
}

public class Bird extends Animal implements Flyer {
    public void takeOff() { /* take-off implementation */ }
    public void land() { /* landing implementation */ }
    public void fly() { /* fly implementation */ }
    public void buildNest( ) { /* nest building behavior */ }
    public void layEggs( ) { /* egg laying behavior */ }
    public void eat( ) { /* override eating behavior */ }
}
```

Usos de las interfaces.

- Declarar métodos que una o más clases deben implementar.
- Determinar la interfaz de programación de un objeto sin saber el cuerpo real de los métodos de la clase.
- Capturar similaridades entre clases poco relacionadas, sin forzar una relación de herencia entre clases.
- Simular la herencia múltiple declarando una clase que implementa varias interfaces.

Clases Anidadas.

- Permiten una definición de una clase dentro de otra.
- Agrupan clases que lógicamente deben ir juntas.
- Tienen acceso a las variables de la clase exterior.
- Utilizadas principalmente en el manejo de eventos de las GUI.

Clases Anidadas (2).

```
public class Outer {
   private int size;

   /* Declare an inner class called "Inner" */
   public class Inner {
      public void doStuff() {
         // The inner class has access to 'size' from Outer
         size++;
         System.out.println("size = " + size);
      }
   }

   public void testTheInner() {
      Inner i = new Inner();
      i.doStuff();
   }
}
```

Nota. Al compilar Outer.java se generan 2 archivos: Outer.class y Outer$Inner.class

Clases Anidadas (3).

```
public class TestInner {
   public static void main(String[ ] args) {
      Outer outer = new Outer();
      outer.testTheInner();
      Outer.Inner inner = outer.new Inner();
      inner.doStuff();
       }
}
```

Ejercicios.

1. Establecer otro paquete en el Sistema Bancario e implementar el patrón de diseño Singleton.

2. Trabajar con interfaces y clases abstractas.

Repaso.

- Variables, métodos e inicializadores estáticos.
- Clases, métodos y variables finales.
- El patrón de diseño Singleton.
- Clases y métodos abstractos.
- Interfaces.
- Clases Anidadas.

Módulo 8.

Excepciones y Aseveraciones.

Objetivos.

- Definir excepciones
- Usar los postulados *try*, *catch* y *finally*.
- Describir la jerarquía de las excepciones.
- Identificar excepciones comunes.
- Crear excepciones propias.
- Utilizar adecuadamente el mecanismo de aseveraciones.

Excepciones.

- Constituyen el mecanismo para describir que hacer cuando sucede algo inesperado en el programa.

- Hay dos clases principales.

  – Checked.
  – Unchecked.

- ***Checked exceptions*** son condiciones que es posible manejar en el programa; suceden debido a condiciones externas que pueden ocurrir en un programa en producción ya depurado, como por ejemplo un archivo no encontrado.

Excepciones (2).

- ***Unchecked exceptions*** son condiciones que representan *bugs* o situaciones difíciles que el programa no puede manejar razonablemente. Cuando representan *bugs* se les conoce como runtime exceptions.

- Condiciones excepcionales que resultan de cuestiones ambientales que son muy raras y normalmente irrecuperables, se llaman ***errors*** y no son manejadas por el programa. Por ejemplo, agotamiento de la memoria.

Ejemplo.

```
public class AddArguments {
    public static void main(String [ ] args) {
     int sum = 0;
     for (int i = 0; i < args.length; i++) {
        sum += Integer.parseInt(args[i]);
     }
     System.out.println("Sum = " + sum);
   }
}
```

java AddArguments 1 2 3 4
Sum = 10

java AddArguments 1 two 3 4
Exception in thread "main" java.lang.NumberFormatException: For input string: "two"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:447)
at java.lang.Integer.parseInt(Integer.java:497)
at AddArguments.main(AddArguments.java:5)

Postulados *try* y *catch*.

```
try {

    // código que podría provocar una excepción.

} catch (ExceptionType1 e) {

    // código que se ejecuta si se provoca una excepción de
    // tipo ExceptionType1.

} catch (Exception eg) {
    // código que se ejecuta si se provoca cualquier otra excepción.

}
```

Ejemplo con *try* y *catch* 1.

```
public class AddArguments2 {
   public static void main(String [] args) {
      try {
         int sum = 0;
         for (int i = 0; i < args.length; i++) {
            sum += Integer.parseInt(args[i]);
         }
         System.out.println("Sum = " + sum);
      } catch(NumberFormatException nfe) {
         System.err.println("One of the command-line arguments is not an
                                                                integer");
      }
   }
}
```

java AddArguments 1 two 3 4
One of the command-line arguments is not an integer

Ejemplo con *try* y *catch* 2.

```
public class AddArguments3 {
    public static void main(String [] args) {
        int sum = 0;
        for (int i = 0; i < args.length; i++) {
           try {
              sum += Integer.parseInt(args[i]);
           } catch(NumberFormatException nfe) {
              System.err.println("[" + args[i] + "] is not an integer" +
                 " and will not be included in the sum.");
           }
        }
        System.out.println("Sum = " + sum);
   }
}
```

java AddArguments 1 two 3 4
[two] is not an integer and will not be included in the sum.
Sum = 8

Mecanismo Call Stack.

- Si un postulado genera una excepción y ésta no es manejada en el método que se está ejecutando, la excepción se envía al método llamador en forma recursiva.

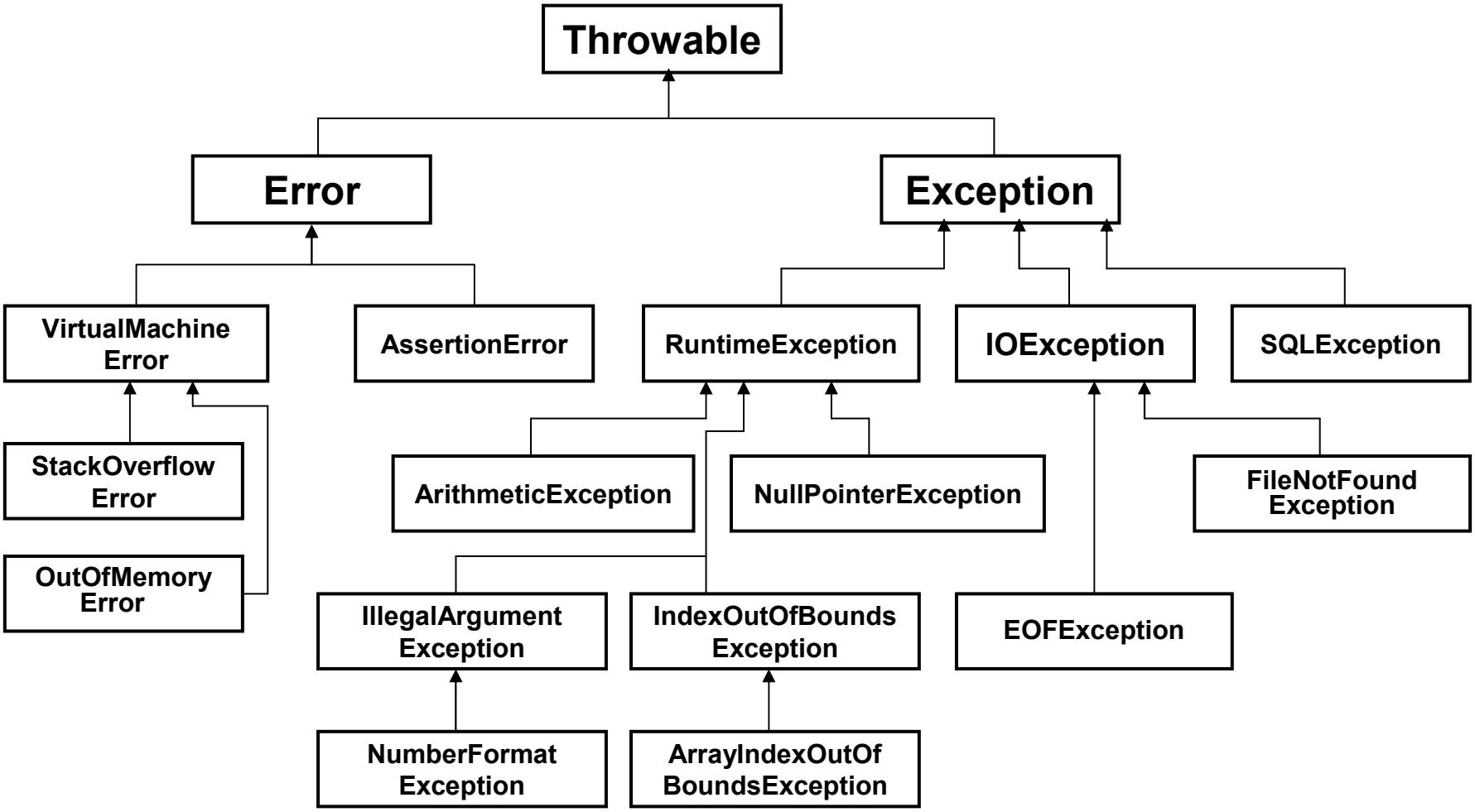
- Si la excepción llega hasta el método main sin ser manejada por ningún método, el programa termina anormalmente.

Postulado *finally*.

- El bloque de *finally* se ejecuta siempre, independientemente de que haya o no excepción.

```
try {
    startFaucet();
    waterLawn();
} catch (BrokenPipeException e) {
    logProblem(e);
} finally {
    stopFaucet();
}
```

Jerarquía de excepciones.



Manejar o Declarar.

Java requiere que si una excepción checked (es decir, que sea subclase de Exception, pero no subclase de RunTimeException) puede ocurrir, el método que contiene el postulado que pudiera provocar la excepción debe:

- Manejar la excepción utilizando los bloques de try, catch y finally.

o bien:

Manejar o Declarar (2)

- Declarar que el método puede provocar la excepción mediante la palabra clave *throws*.

- Provocar la excepción utilizando la palabra clave *throw.*

Nota. Un método puede declarar que puede provocar más de un tipo de excepción.

Declaración de excepciones.

```
public void readDatabaseFile(String file)
        throws FileNotFoundException, UTFDataFormatException {

          // open file stream; may cause FileNotFoundException
          FileInputStream fis = new FileInputStream(file);
          // read a string from fis may cause UTFDataFormatException...
}
```

Nota. No es necesario manejar o declarar runtime exceptions o errors.

Substitución de métodos y excepciones.

- El método de la subclase (overriding method) puede provocar excepciones que sean subclases de las excepciones que puede provocar el método de la superclase (overriden method).

Ejemplo.

```
public class TestA {
    public void methodA() throws java.io.IOException {
        // do some number crunching
    }
}
public class TestB1 extends TestA {
    public void methodA() throws java.io.EOFException {        // OK
        // do some number crunching
    }
}
public class TestB2 extends TestA {
    public void methodA() throws Exception {                   // Inválido
        // do some number crunching
    }
}
```

TestB2 no compila:

methodA() in TestB2 cannot override methodA() in TestA; overridden method does not throw java.lang.Exception

Excepciones de usuario.

```
public class ServerTimedOutException extends Exception {
     private int port;

     public ServerTimedOutException(String message, int port) {
        super(message);
        this.port = port;
     }

     public int getPort() {
        return port;
     }

      // El método getMessage() de la clase Exception se puede usar
      // para recuperar  el parámetro message.
}
```

Excepciones de usuario (2).

```
public void connectMe(String serverName)
                                    throws ServerTimedOutException {

    int success;
    int portToConnect = 80;

    success = open(serverName, portToConnect);

    if (success == -1) {
        throw new ServerTimedOutException("Could not connect",
                                                    portToConnect);
    }
}
```

Nota. El método open( ) es ficticio.

Excepciones de usuario (3).

```
public void findServer() {
    try {
        connectMe(defaultServer);
    } catch (ServerTimedOutException e) {
        System.out.println("Error: " + e.getMessage() +
                           " connecting to port " + e.getPort() );
    }
}
```

Aseveraciones (assertions).

- El mecanismo de aseveraciones permite probar ciertas suposiciones acerca de la lógica de un programa.
- Su característica más importante es que se pueden remover completamente del código cuando el programa ejecuta.
- Es posible habilitar las aseveraciones durante el desarrollo del programa, pero deshabilitarlas cuando el programa se envía a producción.

Sintaxis de las aseveraciones.

- assert **<expresión booleana> ;**
- assert **<expresión booleana> : <expresión-mensaje> ;**
- Si la expresión booleana es falsa se genera un error de tipo AssertionError y el programa termina anormalmente.
- La expresión-mensaje se convierte a String y se usa para complementar el mensaje que se imprime cuando ocurre el AssertionError.

Usos recomendados.

- Las aseveraciones se deben usar para verificar la lógica interna de un método o de un grupo de métodos relacionados.

- Los casos recomendados son:

  – Invariantes internas.
  – Invariantes de control de flujo.
  – Postcondiciones e invariantes de clase.

Invariantes internas.

- Existen cuando se sabe que una situación ocurre siempre o no ocurre nunca.

- Por ejemplo, si se sabe que x tiene que ser positivo o cero, forzosamente, se puede codificar:

```
if (x > 0) {
     System.out.println("x = " + x);
 } else {
      assert (x == 0):  "x no debe ser nunca negativo: " + x;
      System.out.println("x = " + x);
 }
```

Invariantes de control de flujo.

- Existen cuando se sabe que el programa no puede pasar por cierta parte del programa.
- Por ejemplo, si en un postulado switch se analizan todas las posibilidades, se puede codificar:

```
switch (conts) {
    case Africa:     System.out.println(conts); break;
    case America:    System.out.println(conts); break;
    case Asia:       System.out.println(conts); break;
    case Europa:     System.out.println(conts); break;
    case Oceanía:    System.out.println(conts); break;
    default: assert false : "Nunca debe entrar aquí";
}
enum Continentes {Africa, America, Asia, Europa, Oceanía}
```

Postcondiciones.

- Postcondiciones son suposiciones acerca del valor o relación de variables cuando se completa la ejecución de un método.
- Por ejemplo, cuando termina el método pop( ) de un stack se debe satisfacer la condición de que el stack tenga un elemento menos que al principio (a menos que estuviera vacío desde el inicio). Esto se puede codificar así:

```
public Object pop() {
 int antes = getElementCount();
 if (elems == 0) {
          throw new RuntimeException("Attempt to pop from empy stack");
        }
 Object result = unStack[--elems];
 int despues = getElementCount();
 assert (despues == --antes): "imposible";
 return result;
}
```

Invariantes de clase.

- Son postcondiciones que pueden ser probadas después de cada llamada a un método de una clase.
- En el ejemplo del stack una invariante de clase sería que el número de elementos no fuera nunca negativo.

Control de aseveraciones.

- Las aseveraciones están deshabilitadas por default.
- Para habilitarlas se usa:

  java -enableassertions MiPrograma

  java -ea MiPrograma

- Se puede controlar también a nivel de paquetes o jerarquía de paquetes.

Ejercicios.

1. Utilización de try y catch para manejar una excepción tipo runtime.

2. Crear excepciones de usuario para el Sistema Bancario.

Repaso.

- Concepto de excepciones y errores.
- Postulados try, catch y finally.
- Jerarquía y tipos de excepciones.
- Excepciones de usuario.
- Aseveraciones.

Módulo 9.

Entrada/Salida, Archivos, Colecciones y *Generics*.

Objetivos.

- Escribir programas que usen argumentos pasados en la línea de comandos.
- Entender las Propiedades del Sistema.
- Escribir programas que lean de la consola (*standard input)*
- Escribir programas que crean, escriban o lean archivos.
- Describir las colecciones básicas de Java.
- Escribir programas que usen sets y lists.
- Escribir programas que usen iteradores sobre colecciones.
- Escribir programas que usen colecciones genéricas.

Argumentos en la línea de comandos.

- Cualquier programa puede recibir argumentos tecleados en la línea de comandos.

  java TestArgs arg1 arg2 “arg num 3”

- Cada argumento se coloca en el arreglo de Strings mencionado en el método main.

  public static void main(String[ ] args)

Ejemplo.

```
public class TestArgs {
    public static void main(String[ ]  args) {
      for ( int i = 0; i < args.length; i++ ) {
          System.out.println("args[" + i + "] is '" + args[i] + "'");
      }
    }
}
```

java TestArgs arg1 arg2 "another arg"

```
args[0] is 'arg1'
args[1] is 'arg2'
args[2] is 'another arg'
```

Propiedades del Sistema.

- System properties es una característica similar a las variables de ambiente (que son dependientes de la plataforma).
- El método estático System.getProperties() regresa un objecto de clase Properties.
- El método getProperty(String ***prop***) de esta clase regresa un String con el valor de la propiedad cuyo nombre es ***prop***.
- La opción –D del comando Java permite incluir nuevas propiedades.

La clase Properties.

- Contiene un mapa de String a String que contiene nombres y valores de las propiedades.
- El método propertyNames() regresa una enumeración de todos los nombres de las propiedades.
- Se pueden leer y escribir propiedades en un archivo utilizando los métodos load() y store().

Ejemplo.

```
import java.util.Properties;
import java.util.Enumeration;
public class TestProperties {
    public static void main(String[ ] args) {
       Properties props = System.getProperties();
       Enumeration propNames = props.propertyNames();
       while (propNames.hasMoreElements() ) {
          String propName = (String) propNames.nextElement();
          String property = props.getProperty(propName);
          System.out.println("property '" + propName + "' is '" + property + "'");
       }
    }
}
```

java –DmyProp=TestProperties

I/O de la consola.

- **System.out** permite escribir a la salida standard.
- **out** es un objeto de clase PrintStream.
- **System.in** permite leer de la entrada standard.
- **in** es un objeto de clase InputStream.
- **System.err** permite escribir a la salida de error standard.
- **err** es un objecto de clase PrintStream.

Salida Standard.

- El método println( ) escribe el argumento a la salida standard y agrega newline newline (\n).
- El método print( ) imprime el argumento sin newline.
- Los métodos println( ) y print( ) están sobrecargados para aceptar primitivos (boolean, char, int, long, float, and double) , char[ ], String y Object.
- El argumento de los métodos println( ) y print( ) es convertido a String llamando al método toString.

Salida Formateada.

- Funcionalidad mediante los métodos printf( ) de la clase System y format( ) de la clase String.
- Misma sintaxis de C y C++.

```
System.out.printf("format string", args,…);
String s = String.format("format string", args,…);
```

- El "Format String" contiene los formatos de los argumentos args.

Sintaxis de los formatos.

%[argumentIndex$][flags][width][.precision]conversion

donde:

argumentlindex:	posición del argumento en la lista.
flags:	modificadores del formato.
width:	mínimo número de caracteres enviados a la salida.
precision	máximo número de caracteres enviados a la salida después del punto.
conversion:	código de formato.

Algunos códigos de formato.

%s	Formatea como String, llamando al método toString().
%d %o %x	Formatea entero decimal, octal o hexadecimal
%f %g	Formatea punto flotante normal o notación científica.
%n	Inserta newline
%%	Inserta el carácter %

Ejemplo de salida formateada.

```
public class TestPrintf {
    public static void main(String [] args) {
     int i = 14;
     float f = 3.1416F;
     double g = 3145678899.014;
     String usuario = "Juan Pérez";
     System.out.printf("Números con formato: %5d %10.4f %15g %s %n",
                                                    i, f, g, usuario);
     String s = String.format("Números con formato: %5d %10.4f %15g
                                                  %s   %n", i, f, g, usuario);
     System.out.println(s);
   }
}
```

Entrada Standard.

```
import java.io.*;
public class KeyboardInput {
  public static void main (String args[ ]) {
    String s;
    InputStreamReader ir = new InputStreamReader(System.in);
    BufferedReader in = new BufferedReader(ir);
    System.out.println("Type ctrl-z and then enter to exit.");
    try {
       while ((s = in.readLine()) != null) {
      System.out.println("Read: " + s);
      }
      in.close();
    } catch (IOException e) {.
      e.printStackTrace();
    }
  }
}
```

Entrada formateada.

```
import java.io.*;
import java.util.Scanner;
public class ScanTest {
   public static void main(String [ ] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Teclear un string: ");
      String s = sc.next();
      System.out.println("línea 1: " + s);
      System.out.print("Teclear un número entero: ");
      int n = sc.nextInt();
      System.out.println("línea 2: " + n);
      System.out.print("Teclear un número decimal: ");
      double d = sc.nextDouble();
      System.out.println("línea 3: " + d);
   }
}
```

Manejo de archivos.

- Basados en streams tipo Unix.
- Dos tipos de streams básicos:
  - Input y Output streams
  - Readers y writers
- Subclases de estos streams (decoradores o filtros) agregan métodos con mayor funcionalidad.
- La clase File representa la estructura del archivo, mientras que el stream representa los datos.

Creación de un objeto File.

- Creación del objeto File.

```
File myFile;

myFile = new File("myfile.txt");

myFile = new File("MyDocs", "myfile.txt");
```

- Los directorios se tratan de la misma manera que los archivos.

```
File myDir = new File("MyDocs");
myFile = new File(myDir, "myfile.txt");
```

Métodos para manejo de archivos.

- Nombres de archivos:
  - String getName( )
  - String getParent( )
  - boolean renameTo(File newName)
  - String getAbsolutePath( )
  - String getPath( )
- Condiciones:
  - boolean exists( )
  - boolean canWrite( )
  - boolean canRead( )
  - boolean isFile( )
  - boolean isDirectory( )
  - boolean isAbsolute( )

Métodos para manejo de archivos (2).

- Información del archivo:
  - long lastModified( )
  - long length( )
- Borrar el archivo.
  - boolean delete( )
- Manejo de directorios:
  - boolean mkdir( )
  - String[ ] list( )

File Streams principales.

- File input:
  - FileReader para leer caracteres.
  - BufferedReader por eficiencia y para usar el método readLine( )
- File output:
  - FileWriter para escribir caracteres.
  - PrintWriter para usar los métodos print( ) y println( ).

Ejemplo de lectura de archivo.

```
import java.io.*;
public class ReadFile {
    public static void main (String[ ] args) {
       File file = new File(args[0]);
       try {
          BufferedReader in = new BufferedReader(new FileReader(file));
          String s;
          s = in.readLine();
          while ( s != null ) {
               System.out.println("Read: " + s);
               s = in.readLine();
       }
```

Ejemplo de lectura de archivo (2).

```
            in.close();

            } catch (FileNotFoundException e1) {
                System.err.println("File not found: " + file);
            } catch (IOException e2) {
                e2.printStackTrace();
            }
        }
}
```

Ejemplo de escritura de archivo.

```
import java.io.*;
public class WriteFile {
   public static void main (String args[]) {
      File file = new File(args[0]);
      BufferedReader in = null;
      PrintWriter out = null;
      try {
         in = new BufferedReader(new InputStreamReader(System.in));
         out = new PrintWriter(new FileWriter(file));
         String s;
         System.out.print("Enter file text.  ");
         System.out.println("[Type cntl-z to stop.]");
         while ((s = in.readLine()) != null) {
            out.println(s);
         }
```

Ejemplo de escritura de archivo (2).

```
            in.close();
            out.close();
        }  catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

La clase Math.

- Se encuentra en el paquete java.lang
  (no confundir con el paquete java.math).
- Contiene muchos métodos matemáticos estáticos:
  - Truncamiento: ceil( ), floor( ), and round( )
  - Variantes de max( ), min( ) y abs( ).
  - Trigonometría: sin( ), cos( ), tan( ), asin( ), acos( ), atan( ), toDegrees( ), toRadians( ).
  - Logaritmos: log( ) and exp( ).
  - Otras: sqrt( ), pow( ), and random( ).
  - Constantes: PI and E

La clase String.

- Los objetos String son secuencias *inmutables* de caracteres Unicode.
- Operaciones para crear nuevos Strings: concat, replace, substring, toLowerCase, toUpperCase y trim.
- Operaciones de búsqueda: endsWith, startsWith, indexOf y lastIndexOf.
- Comparaciones: equals, equalsIgnoreCase y compareTo.
- Otros: charAt and length.

La clase StringBuffer.

- Los objetos StringBuffer son secuencias mutables de caracteres Unicode.

- Constructores:

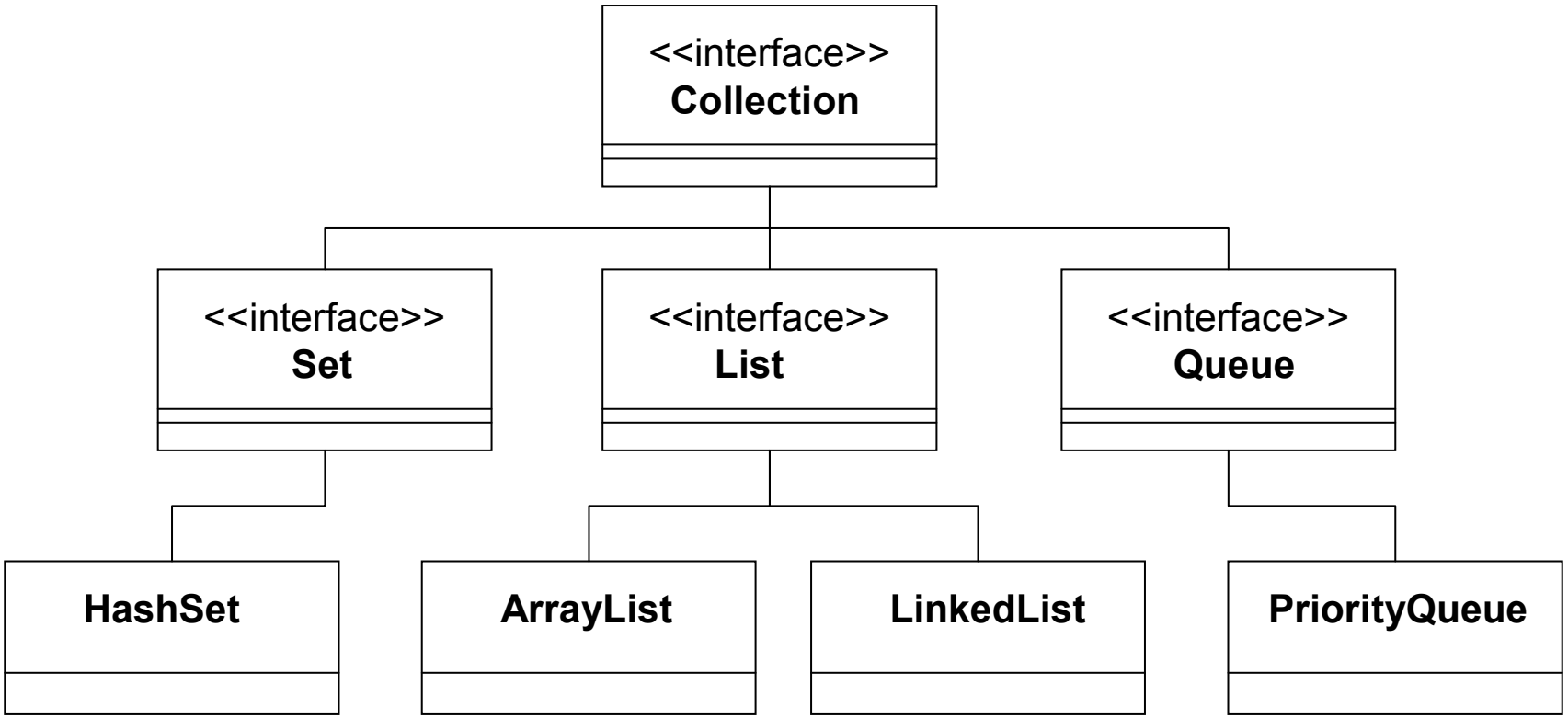
```
StringBuffer( )                      // crea un buffer vacío
StringBuffer(int capacity)           // crea un buffer con capacidad
StringBuffer(String initialString)   // crea un buffer inicializado
```

- Operaciones: append( ), insert( ), reverse( ), setCharAt( ) y setLength( ).

La API de colecciones.

- Una *colección* es un objeto que representa un grupo de objetos llamados sus elementos.
- Principales tipos de colecciones (interfaces).
  - Set un grupo de objetos sin orden específico, No se permiten elementos duplicados.
  - List un grupo de elementos ordenados, también llamado secuencia.
  - Queue colección diseñada para extracciones e inserciones.

La API de colecciones (2).



Nota. Existen muchas otras implementaciones de las colecciones.

Ejemplo de un set.

```
import java.util.*;
    public class SetExample {
       public static void main(String[ ] args) {
            Set set = new HashSet();
            set.add("one");
            set.add("second");
            set.add("3rd");
            set.add(new Integer(4));
            set.add(new Float(5.0F));
            set.add("second");          // duplicado, no se añade
            set.add(new Integer(4));    // duplicado, no se añade
            System.out.println(set);
    }
}
```

Ejemplo de una lista.

```
import java.util.*
public class ListExample {
    public static void main(String[ ] args) {
        List list = new ArrayList();
        list.add("one");
        list.add("second");
        list.add("3rd");
        list.add(new Integer(4));
        list.add(new Float(5.0F));
        list.add("second");              // duplicado, se añade
        list.add(new Integer(4));        // duplicado, se añade
        System.out.println(list);
    }
}
```

Iteradores.

- Iteración es el proceso de recuperar los elementos de una colección.
- La interfaz Iterator define la estructura de los iteradores.
- Existen iteradores para diferentes tipos de colecciones.
  - Los de Sets no permiten orden
  - Los de List pueden iterar hacia adelante (método next) o hacia atrás (método previous).

Ejemplo de Iterador.

```
import java.util.*;
public class ListExampleWithIte {
  public static void main(String[ ] args) {
    ArrayList list = new ArrayList();
    list.add("one");
    list.add("second");
    list.add("3rd");
    list.add(new Integer(4));
    list.add(new Float(5.0F));
    list.add("second");
    list.add(new Integer(4));
    System.out.println(list);
```

Ejemplo de Iterador (2).

```
System.out.println("\nUsando el Iterador");
    ListIterator iteList = list.listIterator();
    while (iteList.hasNext()) {
      System.out.println(iteList.next());
    }

    System.out.println("\nAhora al revés");
    while (iteList.hasPrevious()) {
        System.out.println(iteList.previous());
    }

  }
}
```

Colecciones de versiones anteriores.

- Vector implementa la interfaz List.
- Stack es una subclase de Vector y supporta los métodos push, pop y peek.
- Hashtable implementa la interfaz Map.
- Enumeration es una implementación de la interfaz Iterator.

Funcionalidad *Generics*.

- Nueva en versión 1.5.
- Las colecciones utilizan la clase Object para permitir cualquier tipo de objeto como elemento.
- Debido a esto, es necesario hacer casting al recuperar los elementos de la colección.
- Generics proporciona información al compilador acerca del tipo de colección utilizada.
- Elimina la necesidad del casting de tipo de datos.
- La combinación de esta funcionalidad con la de autoboxing, permite escribir código simple y más comprensible.

Comparación de código.

- Sin utilizar *Generics*:

```
ArrayList list = new ArrayList();
list.add(0, new Integer(42));
int total = ((Integer)list.get(0)).intValue();
```

- Utilizando *Generics*:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = list.get(0).intValue();
```

- Utilizando *Generics y autoboxing*:

```
ArrayList<Integer> list = new ArrayList<Integer> ();
list.add(0, 42);
int total = list.get(0);
```

Warnings del compilador.

- Cuando se usa el compilador de la versión 1.5 sin utilizar la sintaxis de Generics, aparecen advertencias especiales:

```
Note: GenericsExample0.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Finished
```

- Recompilando con javac -Xlint:unchecked aparece el mensaje completo:

```
GenericsExample0.java:6: warning: [unchecked] unchecked call to add(int,E) as a
member of the raw type java.util.ArrayList
list.add(0, new Integer(42));
     ^
```

- Se puede compilar con la opción -nowarn para evitar los mensajes de advetencia.

Clases y métodos descontinuados.

- *Deprecation* es el término usado para clases, atributos, métodos y constructores que han sido descontinuados.
- Se reemplazaron con elementos más eficientes o simplemente se les cambió de nombre.
- Se recomienda no usarlos, están marcados claramente en la documentación.
- Se puede compilar con la bandera –*deprecation* para tener más información:

javac -deprecation MyFile.java

Ejercicios.

1. Crear un archivo numerado a partir de datos tecleados en la consola.

2. Implementar los conjuntos de clientes y cuentas del Sistema Bancario como colecciones Java.

Repaso.

- Argumentos tecleados en la línea de comandos.
- Propiedades del Sistema.
- Lectura de la consola.
- Creación, lectura y escritura de archivos.
- Jerarquía básica de las colecciones de Java.
- Sets, Lists y Maps.
- Uso de Iteradores.
- Otros tipos de colecciones: Vector, Stack, Enumeration.
- Generics.
- Clases y métodos descontinuados y la bandera –deprecation.

Módulo 10.

Conceptos Básicos de GUIs.

Objetivos.

- Describir el paquete Abstract Windowing Toolkit (AWT) y sus componentes.
- Definir los términos *container*, *component* y *layout manager,* y entender como trabajan en la construcción de GUIs.
- Utilizar los tipos principales de Layout Managers: FlowLayout, BorderLayout y GridLayout.
- Aprender a añadir componentes a los contenedores.
- Utilizar los contenedores Frame y Panel adecuadamente.
- Describir como trabajan estructuras GUI más complejas mediante combinaciones de contenedores anidados.

El Paquete AWT.

- Proporciona los componentes básicos de las interfaces gráficas usadas en las aplicaciones y applets.
- El paquete Swing representa una versión mejorada de AWT.
- Las clases de AWT pueden ser incluídas directamente o pueden ser extendidas.
- Cada componente desplegado en la GUI es una subclase de Component o MenuComponent.
- Además, existe la clase Container, que es una subclase de Component, en donde se colocan los componentes.

Principales clases de AWT.

- BorderLayout
- CardLayout
- CheckboxGroup
- Event
- Font
- FlowLayout
- FontMetrics
- Graphics
- GridBagLayout
- GridLayout
- Image
- Point
- Polygon
- Rectangle

Principales clases de AWT (2).

- **MenuComponent**
  - **MenuBar**
  - **MenuItem**
    - **Menu**
      - **PopupMenu**
    - **CheckboxMenuItem**
- **Component**
  - **Button**
  - **Canvas**
  - **Checkbox**
  - **Choice**
  - **Label**
  - **List**
  - **Scrollbar**
  - **TextComponent**
    - **TextArea**
    - **TextField**

Principales clases de AWT (3).

- **Component** (continúa)
  - **Container**
    - **Panel**
      - » **Applet** **(en el paquete java.applet)**
    - **Window**
      - » **Dialog ➔ FileDialog**
      - » **Frame**
    - **ScrollPane**

- **Excepciones ➔ AWTException**

- **Errores ➔ AWTError**

Containers.

- Son componentes diseñados para incluir otros componentes, que pueden ser simples u otros containers.
- Los dos tipos principales son Window y Panel.
- Window es una ventana flotante sin título ni bordes. Normalmente se utiliza su subclase Frame que sí tiene estas características.
- Un Panel no puede existir por sí mismo sino en el contexto de otro contenedor como Frame.
- Se añaden componentes al Container con el método add( ).

Posicionamiento automático.

- Aunque es posible posicionar los componentes en forma manual, ésto no se recomienda.
- Los ***Layout Managers*** determinan automáticamente la posición y tamaño de los componentes.
- Los ***Layout Managers*** pueden deshabilitar y en ese caso usar los métodos:
  - setSize( )
  - setBounds( )

Frames.

- Subclase de Window.
- Tienen título, barra de menús, bordes y esquinas para modificar el tamaño.
- Por default son invisibles, se debe usar setVisible(true) una vez armado el frame para desplegarlo.
- Su *layout manager* de default es BorderLayout.
- Se usa el método setLayout( ) para usar otro *layout manager*. **(O setLayout(null) para no usar *layout manager*.)**

Ejemplo de Frame.

```
import java.awt.*;
public class FrameExample {
    private Frame f;
    public FrameExample() {
       f = new Frame("Hello Out There!");
    }
    public void launchFrame() {
    f.setSize(170,170);
    f.setBackground(Color.blue);
    f.setVisible(true);
    }
    public static void main(String args[]) {
       FrameExample guiWindow = new FrameExample();
       guiWindow.launchFrame( );
    }
}
```

Panel.

- Rectángulos sin título, bordes, menús.
- Proporcionan espacio para colocar componentes.
- Cada Panel puede tener su propio *Layout Manager*.
- El *Layout Manager* de default para panel es FlowLayout.

Ejemplo de Frame con Panel.

```
import java.awt.*;
public class FrameWithPanel {
      private Frame f;
      Panel pan;
      public FrameWithPanel(String title) {
          f = new Frame(title);
          pan = new Panel( );
      }
      public void launchFrame() {
          f.setSize(200,200);
          f.setBackground(Color.blue);
          f.setLayout(null);                    // Override default layout manager
          pan.setSize(100,100);
          pan.setBackground(Color.yellow);
          f.add(pan);
          f.setVisible(true);
      }
     public static void main(String args[ ]) {
          FrameWithPanel guiWindow = new FrameWithPanel("Frame with Panel");
          guiWindow.launchFrame( );
      }
}
```

Layout managers.

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

Nota. En swing existe también BoxLayout.

Default Layout managers.

- BorderLayout
  - Windows
  - Frame
  - Dialog
- FlowLayout
  - Panel
  - Applet

FlowLayout.

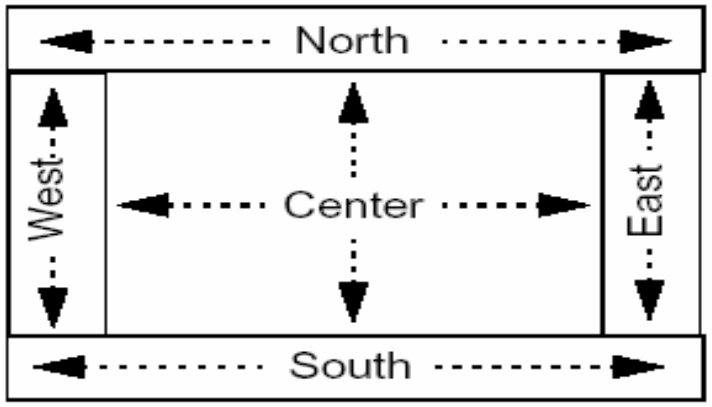
- Los componentes se añaden de izquierda a derecha hasta llenar una línea.
- Los components usan su tamaño preferido.
- Se alinenan centrados en cada línea.
- Con el constructor se puede afinar el comportamiento.
- Default de Panel y Applet.

Ejemplo de FlowLayout.

```
import java.awt.*;
public class FlowExample {
      private Frame f;
      private Button button1;
      private Button button2;
      private Button button3;
      public FlowExample( ) {
                f = new Frame("Flow Layout");
                button1 = new Button("Ok");
                button2 = new Button("Open");
                button3 = new Button("Close");
      }
      public void launchFrame( ) {
                f.setLayout(new FlowLayout());
                f.add(button1);
                f.add(button2);
                f.add(button3);
                f.setSize(100,100);
                f.setVisible(true);
      }
      public static void main(String args[ ]) {
      FlowExample guiWindow = new FlowExample( );
      guiWindow.launchFrame( );
      }
}
```

BorderLayout.

- Divide el container en 5 regiones.
- Los componentes se añaden a regiones específicas.
- Conducta de redimensionamiento:
  - Las regiones North, South y Center se ajustan horizontalmente.
  - Las regiones East, West y Center se ajustan verticalmente.
- Es el default para Frame.



Ejemplo de BorderLayout.

```
import java.awt.*;
public class BorderExample {
      private Frame f;
      private Button bn, bs, bw, be, bc;
      public BorderExample( ) {
            f = new Frame("Border Layout");
            bn = new Button("B1");
            bs = new Button("B2");
            bw = new Button("B3");
            be = new Button("B4");
            bc = new Button("B5");
      }
      public void launchFrame () {
            f.add(bn, BorderLayout.NORTH);
            f.add(bs, BorderLayout.SOUTH);
            f.add(bw, BorderLayout.WEST);
            f.add(be, BorderLayout.EAST);
            f.add(bc, BorderLayout.CENTER);
            f.setSize(200,200);
            f.setVisible(true);
      }
      public static void main(String args[ ]) {
            BorderExample guiWindow = new BorderExample();
            guiWindow.launchFrame( );
      }
}
```

GridLayout.

- Divide el contenedor en regiones de igual tamaño (grid).
- Añade componentes de izquierda a derecha y de arriba hacia abajo.
- El constructor especifica el número de filas y columnas.

Ejemplo de GridLayout.

```
import java.awt.*;
public class GridExample {
      private Frame f;
      private Button b1, b2, b3, b4, b5, b6;
      public GridExample( ) {
            f = new Frame("Grid Example");
            b1 = new Button("1");
            b2 = new Button("2");
            b3 = new Button("3");
            b4 = new Button("4");
            b5 = new Button("5");
            b6 = new Button("6");
      }
      public void launchFrame() {
            f.setLayout (new GridLayout(3,2));
            f.add(b1);  f.add(b2);
            f.add(b3);  f.add(b4);
            f.add(b5);  f.add(b6);
            f.pack( );
            f.setVisible(true);
      }
      public static void main(String args[ ]) {
            GridExample grid = new GridExample();
            grid.launchFrame();
      }
}
```

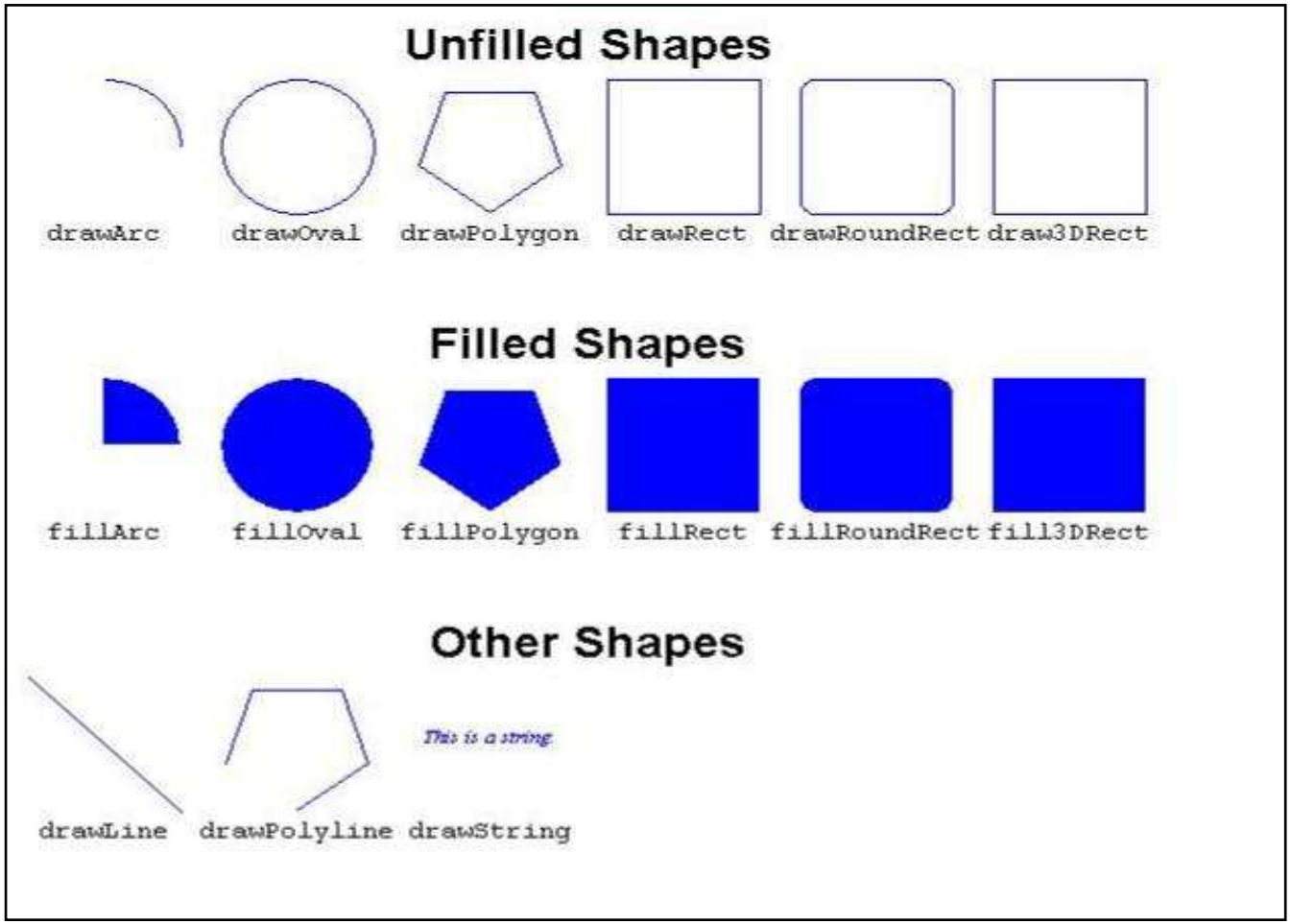
Ejemplo de combinación.

```
import java.awt.*;
public class ComplexLayoutExample {
      private Frame f;
      private Panel p;
      private Button bw, bc;
      private Button bfile, bhelp;
      public ComplexLayoutExample() {
            f = new Frame("GUI example 3");
            bw = new Button("West");
            bc = new Button("Work space region");
            bfile = new Button("File");
            bhelp = new Button("Help");
      }
      public void launchFrame() {
            f.add(bw, BorderLayout.WEST);
            f.add(bc, BorderLayout.CENTER);
            p = new Panel( );
            p.add(bfile);   p.add(bhelp);
            f.add(p, BorderLayout.NORTH);
            f.pack();     f.setVisible(true);
      }
      public static void main(String args[ ]) {
            ComplexLayoutExample gui = new ComplexLayoutExample();
            gui.launchFrame();
      }
}
```

Facilidades de dibujo.

- Se puede dibujar en cualquier componente, aunque se recomienda Canvas.
- Se crea una subclase de Canvas y se substituye el método paint.
- El método paint se llama cada vez que se muestra el componente.
- Cada componente tiene un objeto Graphics.
- La clase Graphics implementa muchos métodos para dibujar.

Ejemplo de dibujo con AWT.



Ejercicios.

1. Crear La GUI para una aplicación de un “chat room”.

2. Crear la GUI para implementar una calculadora.

Repaso.

- El Paquete AWT y sus componentes.
- Containers y componentes.
- Layout Managers y posicionamiento automático de componentes.
- FlowLayout, BorderLayout y GridLayout.
- Añadir componentes a containers.
- Frames y Panels.
- Combinaciones de contenedores y despliegues más complejos.
- Facilidades de dibujo en AWT.

Módulo 11.

Manejo de Eventos en interfaces gráficas.

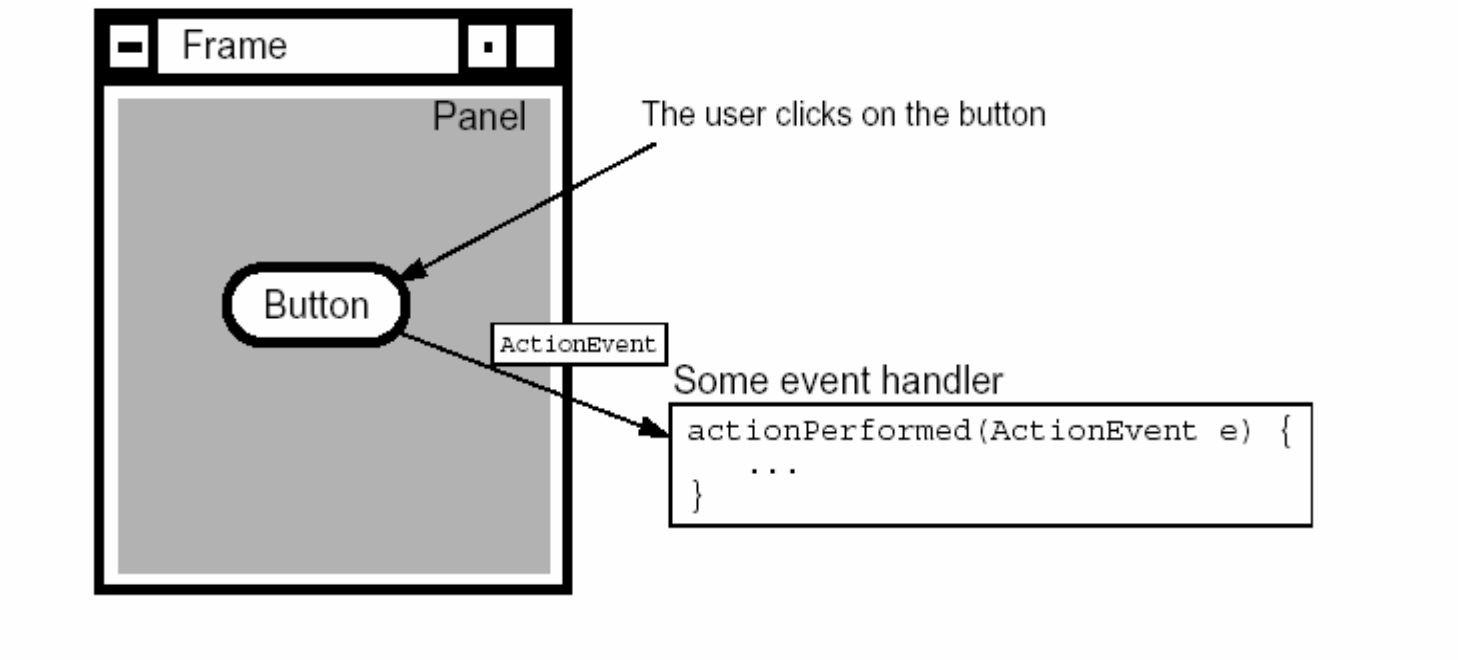
Objetivos.

- Definir eventos y manejo de eventos.
- Escribir código para manejar los eventos que ocurren asociados a la GUI.
- Describir las interfaces de manejo de eventos y los tipos de eventos.
- Describir las clases adaptadoras de eventos.
- Crear los manejadores de eventos apropiados para cada tipo de evento.
- Entender el uso de clases anidadas y anónimas en el manejo de eventos.

Concepto de evento.

- Objeto que describe que pasó cuando el usuario llevó a cabo alguna acción en la GUI.
- La fuente del evento es el componente que lo genera.
- El manejador del evento es el método que recibe el objeto *event* y reacciona ante la ocurrencia del mismo.

Concepto de evento (2).



Un click en el botón genera una instancia de ActionEvent, que contiene información de lo que pasó en los métodos:

- **getActionCommand()** -- entrega el nombre del comando asociado a la acción.
- **getModifiers()** -- entrega información de las teclas usadas para provocar el evento.

Modelo Delegacional.

- Un evento puede ser enviado a varios manejadores de eventos.
- Los manejadores de eventos se registran con los componentes cuando les interesa “escuchar” eventos generados por ese componente.
- Los componentes sólo activan los manejadores del tipo de evento que ocurrió.
- La mayoría de los componentes pueden generar varios tipos de eventos.

Ejemplo simple.

```
import java.awt.*;
public class TestButton {
    private Frame f;
    private Button b;
    public TestButton() {
        f = new Frame("Test");
        b = new Button("Press Me!");
        b.setActionCommand("ButtonPressed");
    }
    public void launchFrame() {
        b.addActionListener(new ButtonHandler());
        f.add(b,BorderLayout.CENTER);
        f.pack();
        f.setVisible(true);
    }
    public static void main(String args[]) {
        TestButton guiApp = new TestButton();
        guiApp.launchFrame();
    }
}
```

Ejemplo simple (2).

```
import java.awt.event.*;
public class ButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
       System.out.println("Action occurred");
       System.out.println("Button's command is: " + e.getActionCommand());
       System.out.println("Modifiers are: " + e.getModifiers());
    }
}
```

Tipos de eventos.

- ActionEvent
- AdjustmentEvent
- ComponentEvent
  - **FocusEvent**
  - **InputEvent**
    - **KeyEvent**
    - **MouseEvent**
  - **ContainerEvent**
  - **WindowEvent**
- ItemEvent
- TextEvent

Interfaces y métodos.

Categoría	Interfaz	Métodos
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	istemStateChanged(ItemEvent)
Mouse	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent)
Mouse Motion	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
Key	KeyListener	keyPressed(keyEvent) keyReleased(keyEvent) keyTyped(keyEvent)
Focus	FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
Adjustment	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)

Interfaces y métodos (2).

Categoría	Interfaz	Métodos
Component	ComponentListener	ComponentMoved(ComponentEvent) ComponentHidden(ComponentEvent) ComponentResized(ComponentEvent) ComponentShown(ComponentEvent)
Window	WindowListener	windowClosing(WindowEventEvent) windowOpened(WindowEventEvent) windowIconified(WindowEventEvent) windowClosed(WindowEventEvent) windowDeactivated(WindowEventEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

Ejemplo.

```
import java.awt.*;
import java.awt.event.*;
public class TwoListener implements MouseMotionListener, MouseListener {
    private Frame f;
    private TextField tf;
    public TwoListener() {
        f = new Frame("Two listeners example");
        tf = new TextField(30);
    }
    public void launchFrame() {
        Label label = new Label("Click and drag the mouse");
        f.add(label, BorderLayout.NORTH);
        f.add(tf, BorderLayout.SOUTH);
        f.addMouseMotionListener(this);
        f.addMouseListener(this);
        f.setSize(300, 200);
        f.setVisible(true);
    }
```

Ejemplo (2).

```
    public void mouseDragged(MouseEvent e) {
        String s = "Mouse dragging: X = " + e.getX() + " Y = " + e.getY();
        tf.setText(s);
    }
    public void mouseEntered(MouseEvent e) {
        String s = "The mouse entered";
        tf.setText(s);
    }
    public void mouseExited(MouseEvent e) {
        String s = "The mouse has left the building";
        tf.setText(s);
    }
    public void mouseMoved(MouseEvent e) { }
    public void mousePressed(MouseEvent e) { }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public static void main(String args[]) {
    TwoListener two = new TwoListener();
    two.launchFrame();
    }
}
```

Ejemplo (3).

```
    public static void main(String[ ] args) {
        TwoListener two = new TwoListener();
        two.launchFrame();
    }
}
```

Clases Adaptadoras.

- Son clases que implementan las interfaces de los eventos con métodos nulos.
- Ayudan a hacer más simple el programa.
- Problema de herencia múltiple.
- Se llaman igual que las interfaces pero con el sufijo *Adapter* en vez de *Listener.*

Ejemplo.

```
import java.awt.*;
import java.awt.event.*;
public class MouseClickHandler extends MouseAdapter {
// Sólo se requiere substituir el método que se va a usar
    public void mouseClicked(MouseEvent e) {
       // código para manejar mouse clicked...
    }
// los demás métodos de MouseListener están implementados en
    MouseAdapter
}
```

Eventos con clases anidadas.

```
import java.awt.*;
import java.awt.event.*;
public class TestInner {
      private Frame f;
      private TextField tf;
      public TestInner() {
          f = new Frame("Inner classes example");
          tf = new TextField(30);
      }
      public void launchFrame() {
          Label label = new Label("Click and drag the mouse");
          f.add(label, BorderLayout.NORTH);
          f.add(tf, BorderLayout.SOUTH);
          f.addMouseMotionListener(new MyMouseMotionListener());
          f.addMouseListener(new MouseClickHandler());
          f.setSize(300, 200);
          f.setVisible(true);
      }
```

Eventos con clases anidadas (2).

```
    class MyMouseMotionListener extends MouseMotionAdapter {
       public void mouseDragged(MouseEvent e) {
          String s = "Mouse dragging: X = "+ e.getX() + " Y = " + e.getY();
          tf.setText(s);
       }
    }  // fin clase anidada

    public static void main(String[ ] args) {
       TestInner obj = new TestInner();
       obj.launchFrame();
    }
}
```

Eventos con clases anónimas.

```
import java.awt.*;
import java.awt.event.*;
public class TestAnonymous {
      private Frame f;
      private TextField tf;
      public TestAnonymous() {
         f = new Frame("Anonymous classes example");
         tf = new TextField(30);
      }
      public void launchFrame() {
            Label label = new Label("Click and drag the mouse");
            f.add(label, BorderLayout.NORTH);
            f.add(tf, BorderLayout.SOUTH);
            f.addMouseMotionListener(new MouseMotionAdapter() {
                  public void mouseDragged(MouseEvent e) {
                     String s = "Mouse dragging: X = "+ e.getX() + " Y = " + e.getY();
                  t  f.setText(s);
                  }
            } ); // fin de la clase anónima (notar el paréntesis)
       f.addMouseListener(new MouseClickHandler());
```

Eventos con clases anónimas (2).

```
        f.setSize(300, 200);
        f.setVisible(true);
    }

    public static void main(String args[]) {
        TestAnonymous obj = new TestAnonymous();
        obj.launchFrame();
    }
}
```

Ejemplo de clase anónima.

```
// Código para cerrar el frame con click en x
    f.addWindowListener(new WindowAdapter() {
       public void windowClosing(WindowEvent e) {
       System.exit(0);
    }
} );
```

Ejercicios.

1. Añadir los manejadores de eventos básicos al "chat room".

2. Terminar la aplicación de la calculadora.

Repaso.

- Eventos y su manejo.
- Modelo Delegacional.
- Manejadores de Eventos.
- Tipos de eventos y sus interfaces.
- Clases Adptadoras.
- Clases anidadas como manejadores de Eventos.
- Clases anónimas como manejadores de Eventos.

Módulo 12.

Componentes y Menús en GUIs.

Objetivos.

- Identificar los componentes básicos del paquete AWT y los eventos que disparan.
- Describir como se construyen barras de menús, menús y menuitems.
- Entender como cambiar el color y el font de un componente.

Componentes.

Componente	Descripción
Button	Caja rectangular para recibir clicks del mouse
Canvas	Panel para dibujo
CheckBox	Caja para seleccionar algo.
CheckBoxMenuItem	Un checkbox dentro de un menú
Choice	Una lista pull-down de elementos para seleccionar.
Component	Superclase de los demás componentes (excepto los de menús)
Container	Superclase de los demás contenedores
Dialog	Una ventana flotante simple generalmente para decisiones
Frame	La clase base para las GUIs, contiene títulos, bordes, etc.
Label	Cadena de caracteres.

Componentes (2).

Componente	Descripción
List	Conjunto dinámico de elementos.
MenuBar	Componente principal de los menús, contiene menús.
Menu	Elemento contenido en la barra de menús, que contiene menuitems.
MenuItem	Un elemento dentro de un menú.
Panel	Contenedor básico para crear distribuciones de componentes complejas.
ScrollBar	Barra que permite al usuario seleccionar de un rango de valores.
ScrollPane	Contenedor con barras de deslizamiento horizontales y verticales.
TextArea	Zona de entrada para un block de texto de varias líneas.
TextField	Zona de entrada de una sola línea de texto.
Window	Superclase de Frame y Dialog, no usada frecuentemente.

Eventos por componentes.

Componente	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Txt	Win
Button	☺		☺		☺		☺	☺	☺		
Canvas			☺		☺		☺	☺	☺		
CheckBox			☺		☺	☺	☺	☺	☺		
CheckBoxMenuItem						☺					
Choice			☺		☺	☺	☺	☺	☺		
Component			☺		☺		☺	☺	☺		
Container			☺	☺	☺		☺	☺	☺		
Dialog			☺	☺	☺		☺	☺	☺		☺
Frame			☺	☺	☺		☺	☺	☺		☺
Label			☺		☺		☺	☺	☺		
List	☺		☺		☺	☺	☺	☺	☺		

Eventos por componentes.

Componente	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Txt	Win
Menu	☺										
MenuItem	☺										
Panel			☺	☺	☺		☺	☺	☺		
ScrollBar		☺	☺		☺		☺	☺	☺		
ScrollPane			☺	☺	☺		☺	☺	☺		
TextArea			☺		☺		☺	☺	☺	☺	
TextField	☺		☺		☺		☺	☺	☺	☺	
Window			☺	☺	☺		☺	☺	☺		☺

Nota. MenuBar no produce eventos.

Creación de Menús.

1. Crear el objeto MenuBar y colorcarlo en un container, normalmente frame.
2. Crear uno o más objetos Menu y añadirlos al MenuBar.
3. Crear uno o más objetos MenuItem y añadirlos al objeto Menu.

Creación de MenuBar.

```
Frame f = new Frame("MenuBar");

MenuBar mb = new MenuBar();

f.setMenuBar(mb);
```

Creación de Menu.

```
Frame f = new Frame("Menu");

MenuBar mb = new MenuBar();

Menu m1 = new Menu("File");

Menu m2 = new Menu("Edit");

Menu m3 = new Menu("Help");

mb.add(m1);

mb.add(m2);

mb.setHelpMenu(m3);

f.setMenuBar(mb);
```

Creación de MenuItem.

```
MenuItem mi1 = new MenuItem("New");
MenuItem mi2 = new MenuItem("Save");
MenuItem mi3 = new MenuItem("Load");
MenuItem mi4 = new MenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
```

Creación de CheckBoxMenuItem.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
.....
MenuItem mi2 = new MenuItem("Save");
mi2.addActionListener(this);
m1.add(mi2);
......
CheckboxMenuItem mi5 = new CheckboxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```

Fonts.

- Se incluyen de forma standard:
  - Dialog.
  - DialogInput.
  - Serif.
  - SansSerif.
- Se pueden usar otros fonts, pero hace el programa dependiente de la plataforma.
- Existe substitución de fonts.

Fonts (2).

- **El método setFont especifica el font usado para desplegar el texto.**
- **Ejemplo:**

```
Font font = new Font("TimesRoman", Font.PLAIN, 14);
```

- **Se puede usar la clase GraphicsEnvironment para recuperar el conjunto de fonts disponibles:**

```
GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
Font[ ] fonts = ge.getAllFonts();
for(Font f: fonts) {
    System.out.println(f.getFontName());
}
```

Colores.

- Métodos de los componentes:
  - setForeground
  - setBackground
- Clase Color.
  - Contiene constantes para los principales colores.
  - Se pueden crear otros colores usando esquema RGB o Color Spaces.

Colores (2).

- Ejemplo:

```
int r = 255;
int g = 200;
int b = 150;
Color c = new Color(r, g, b);
Button btn = new Button()
btn.setBackground(c);
```

Swing.

- Swing es una segunda generación de un framework para GUIs.
- Está basado en AWT, pero incluye componentes escritos totalmente en Java.
- Incluye nuevas características como Look and Feel programable, tool tips, botones con gráficas, etc.
- Hay varios nuevos componentes como JTable, JTree, y JComboBox.

Ejercicios.

1. Terminar la GUI del "chat room" añadiendo menús y otros componentes.

Repaso.

- Componentes de AWT y los eventos que provocan.
- Construcción de barras de menús, menús y menu items en las interfaces gráficas.
- Colores y fonts de los componentes.

Módulo 13.

Introducción a JDBC.

Objetivos.

- Describir JDBC.
- Explicar como usar JDBC para lograr portabilidad entre manejadores de Bases de Datos.
- Describir los 6 pasos necesarios para usar la interfaz de programación de JDBC.
- Conocer los requerimientos de un driver de JDBC y su relación con el JDBC driver manager.

La interfaz de JDBC.

- JDBC es una capa de abstracción que permite al usuario escoger el manejador de Bases de Datos.
- JDBC permite escribir código usando una API común.
- JDBC permite cambiar el manejador de Base de Datos transparentemente.
- JDBC suporta Bases de Datos compatibles con ANSI SQL-2, pero puede ser usado con otras bases de datos relacionales.

Componentes de JDBC.

- Una serie de interfaces y clases proporcionadas por Java para los desarrolladores de programas de bases de datos. Se distribuyen en el paquete java.sql.

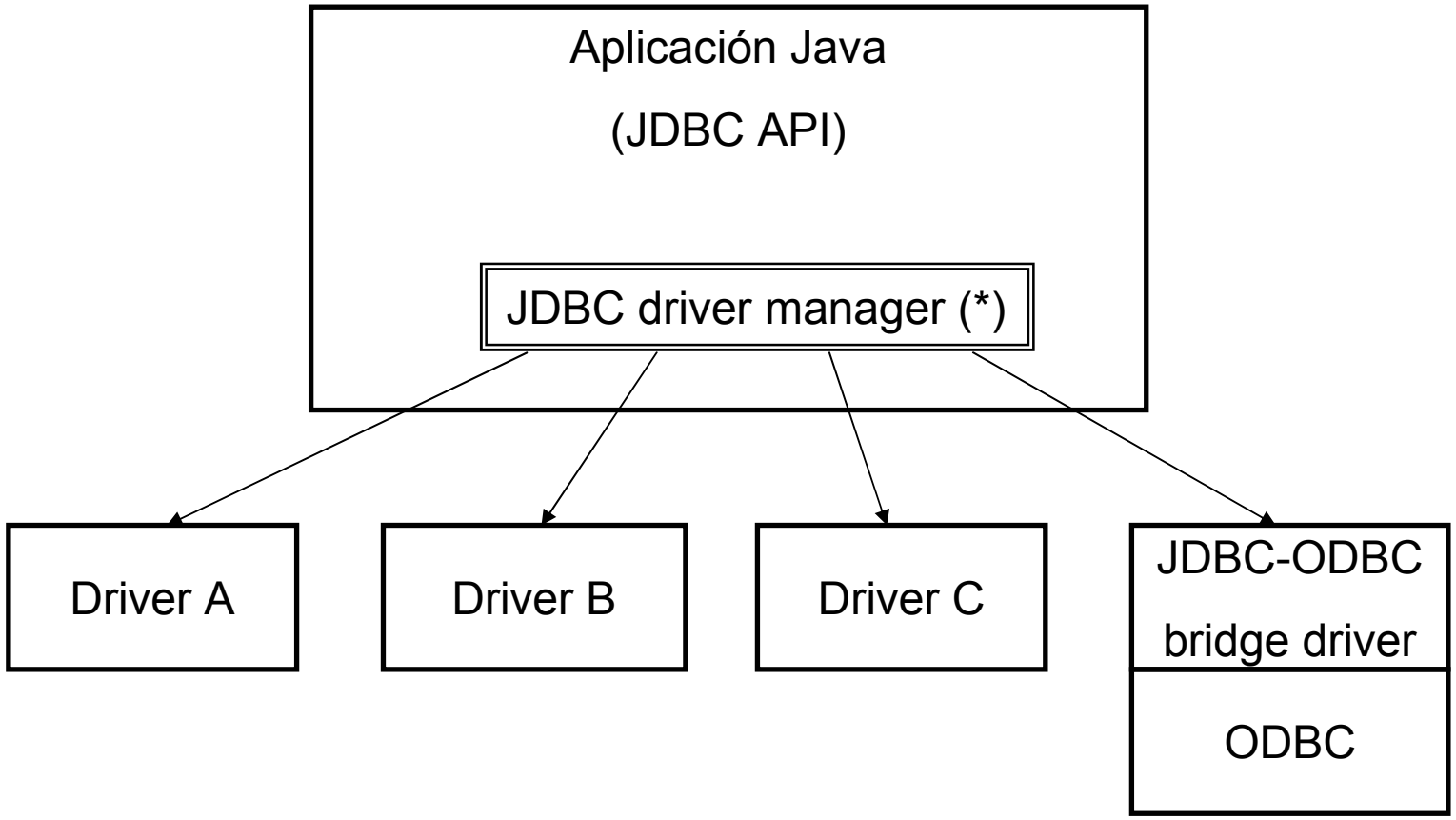
- Un conjunto de clases que implementan las interfaces del paquete java.sql, desarrolladas por el proveedor de la base de datos o algún desarrollador independiente.

- A este conjunto de clases se le conoce como el driver de JDBC.

JDBC Driver.

- El driver del DBMS debe proporcionar implementaciones, cuando menos, para las siguientes interfaces del paquete java.sql:
  - java.sql.Connection
  - java.sql.Statement
  - java.sql.PreparedStatement
  - java.sql.CallableStatement
  - java.sql.ResultSet
  - java.sql.Driver

JDBC Driver (2).



(*) Java.sql.DriverManager es una clase pública.

Pasos para usar JDBC.

1) Crear una instancia del JDBC driver .
2) Especificar la *url* de la base de datos.
3) Establecer una conexión usando el driver que crea el objeto *Connection*.
4) Crear un objeto *Statement,* usando *Connection.*
5) Armar el postulado SQL y enviarlo a ejecución usando el *Statement*.
6) Recibir los resultados en el objeto *ResultSet.*

1. Crear instancia.

```
import java.sql.*;
import paquete.driver.*;
…
new nombreDriver();
```

o bien:

```
Class.forName("paquete.driver.nombreDriver");
```

2. Especificar la *url* de la base de datos.

String url = “jdbc:subprotocolo:subname”

- *subprotocolo* es un nombre corto que identifica al driver.

- *subname* es dependiente del DBMS, normalmente contiene el nombre del servidor y el nombre de la base de datos cuando menos.

  e.g. String url = “jdbc:msql://servername:puerto/dbname”

3. Establecer una conexión.

Connection mcon = DriverManager.getConnection(url);

- El método estático getConnecion usa el valor de la url como argumento.
- Si se establece la conexión sin problema, regresa un objeto de clase Connection.
- Si hay problemas, se genera una SQLException.
- El objeto de clase Connection representa una sesión con una base de datos específica.

4. Crear un objeto *Statement.*

Statement stmt = mcon.createStatement()

- El método createStatement regresa un objeto de clase Statement.
- Si hay problemas, se genera una SQLException.
- El objeto clase Statement es usado para enviar postulados SQL a la Base de Datos.

5. Armar el postulado SQL y enviarlo a ejecución.

ResultSet rs = stmt.executeQuery("postulado Select SQL");
o bien:
int num = stmt.ExecuteUpdate("otro postulado SQL");

- El método executeQuery regresa un objeto de clase ResultSet que contiene los resultados del query SQL.
- El objeto clase ResultSt es usado para interpretar el resultado del query de SQL.
- El método executeUpdate regresa un entero que contiene el número de renglones de la tabla relacional afectados por "otro postulado SQL" que puede ser UPDATE, INSERT, DELETE u otro postulado de SQL.
- Si hay problemas, se genera una SQLException.

6. Recibir los resultados en el objeto *ResultSet.*

```
while(rs.next()) {
        System.out.println("Columna 1: " + rs.getString(1));
        System.out.println("Columna 2: " + rs.getString(2));
    }
```

- El conjunto de renglones, resultado del Query, se almacena en el objeto ResultSet.
- El objeto ResultSet inicialmente apunta antes de la primera fila.
- El método next( ) de ResultSet mueve a la siguiente fila.
- Los métodos getXXX( ) de ResultSet permiten acceso a las columnas de la fila apuntada.

Principales métodos getXXX.

Método	Tipo regresado	Método	Tipo regresado
getASCIIStream	Java.io.InputStream	getFloat	float
getBigDecimal	Java.math.BigDecimal	getInt	int
getBinaryStream	Java.io.InputStream	getLong	long
getBoolean	boolean	getObject	Object
getByte	byte	getShort	short
getBytes	byte[ ]	getString	java.lang.String
getDate	Java.sql.Date	getTime	java.sql.Time
getDouble	double	getTimestamp	Java.sql.Timestamp
getArray	Java.sql.Array	getBlob	java.sql.Blob

Métodos getXXX.

- El argumento de los métodos getXXX puede ser:
  - El número de la columna de la tabla empezando con 1
  - El nombre de la columna.
  - Se recomienda usar el número.

Programa Ejemplo.

- JDBCExampleXX.java
- XX para diferentes manejadores de bases de datos.

Prepared Statements.

- Para llamar al mismo postulado SQL repetidas veces se usa el objeto PreparedStatement.
- PreparedStatement es una subclase de Statement, que permite cambiar valores dentro del postulado.
- Se usa el carácter ? como placeholder para el parámetro.
- El valor del parámetro se especifica con los métodos setXXX.

Ejemplo.

```
Connection conn = DriverManager.getConnection(url);


PreparedStatement stmt = conn.prepareStatement(
                         "UPDATE table1 set m = ? WHERE x = ?");

stmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
  stmt.setInt(2, i);
  int j = stmt.executeUpdate();
  System.out.println(j  + " rows affected when i= " + i);
}
```

Principales métodos setXXX.

Método	Tipo SQL	Método	Tipo SQL
setArray	Array	setFloat	FLOAT
setASCIIStream	LONGVARCHAR	setInt	INTEGER
setBigDecimal	NUMERIC	setLong	LONG
setBinaryStream	LONGVARBINARY	setObject	Se convierte
setBoolean	BIT	setShort	SHORT
setByte	TINYINT	setString	VARCHAR o LONGVARCHAR
setBytes	VARBINARY	setTime	TIME
setDate	DATE	setTimestamp	TIMESTAMP
setDouble	DOUBLE	setBlob	BLOB

Callable Statements.

- Para ejecutar postulados No-SQL, como Stored Procedures contra la base de datos.

- CallableStatement es sublcase de PreparedStatement.

Ejemplo.

```
String planeID = "757";

CallableStatement querySeats = msqlConn.prepareCall("{call
                                        return_seats[?, ?, ?, ?]}");

querySeats.setString(1, planeID);
querySeats.registerOutParameter(2, java.sql.Types.INTEGER);
querySeats.registerOutParameter(3, java.sql.Types.INTEGER);
querySeats.registerOutParameter(4, java.sql.Types.INTEGER);
querySeats.execute();
int FCSeats = querySeats.getInt(2);
int BCSeats = querySeats.getInt(3);
int CCSeats = querySeats.getInt(4);
```

ResultSet MetaData.

- Objeto asociado al ResultSet que contiene información acerca de los datos.

```
rs = stmt.executeQuery(cmdSQL);
rsmd = rs.getMetaData();
```

Métodos de ResultSet MetaData.

- **getColumnCount()**
- **getColumnName(n)**
- **getColumnType(n)**
- **getColumnTypeName(n)**
- **getTableName()**

… y varios más.

JDBC 2.

- En el paquete javax.sql se incluyen nuevas clases que mejoran el rendimiento y funcionalidad de JDBC.
- La principal es el DataSource que susbstituye al driver manager.
- Utilizando DataSource es posible crear un pool de conexiones.
- En vez de utilizar DriverManager.getConnection( ) que **crea** una conexión se usa DataSource.getConnection( ) que **obtiene** una conexión del pool.
- JDBC 2 es la implementación utilizada por los Application Servers en aplicaciones web.

Ejercicio.

1. Escribir un programa de altas, bajas y cambios para una tabla de una Base de Datos.

Repaso.

- Concepto de JDBC.
- Drivers de JDBC.
- Los 6 pasos necesarios para usar la interfaz de programación de JDBC:
  1. Crear una instancia del JDBC driver .
  2. Especificar la *url* de la base de datos.
  3. Establecer una conexión.
  4. Crear un objeto *Statement.*
  5. Armar el postulado SQL
  6. Recibir los resultados..
- Métodos getXXX.
- Prepared Statements.
- Métodos setXXX
- Callable Statements.

Módulo 14.

Conceptos de Multithreading en Java.

Objetivos.

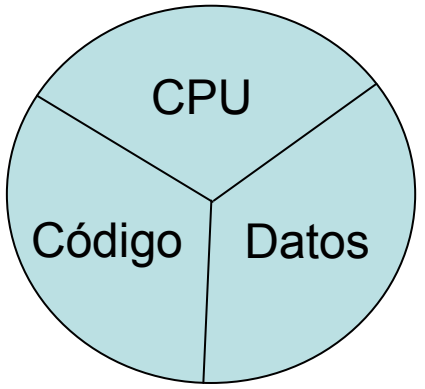
- Entender el concepto de *thread* y *multithreading.*
- Crear *threads* en un programa Java, controlando el código y los datos de cada *thread*.
- Controlar la ejecución de los *threads* y escribir código indpendiente de la plataforma usando *threads*.
- Describir las dificultades que se presentan cuando múltiples *threads* comparten datos.
- Usar *wait* y *notifiy* para manejar la comunicación entre threads.
- Utilizar la palabra clave *synchronized* para proteger datos compartidos entre threads.

Fundamentos de threads.

- El mecanismo de multithreading permite que una sola máquina virtual de Java parezca muchas corriendo al mismo tiempo.
- Un *thread*, también llamado *execution context* es un bloque de código de un programa ejecutándose independientemente.
- A veces se define como una CPU virtual.
- Hay *threads* del sistema siempre ejecutando concurrentemente con el programa del usuario.
- Existe siempre un *"Thread Scheduler"* en el sistema.

Fundamentos de threads (2).

- Un thread está compuesto de tres partes:
  - –Una CPU virtual.
  - –El código que ejecuta la CPU.
  - –Los datos con los que trabaja el código.
- Un proceso es un programa en ejecución.
- Uno o más threads constituyen un proceso.



Como se ejecuta un thread (1).

Método 1. Extendiendo clase Thread.

```
public class CounterThread extends Thread {
    public void run() {
        for(int i = 1; i<=10; i++) {
            System.out.println("Counting " + i);
        }
    }
}
```

En el programa principal:

```
CounterThread ct = new CounterThread();
ct.start();              // start() NO run();
```

Como se ejecuta un thread (2).

Método 2. Implementando la interfaz Runnable.

```
public class DownCounter implements Runnable {
    public void run() {
        for(int i = 1; i>=1; i--) {
            System.out.println("Counting Down " + i);
        }
    }
}
```

En el programa principal:

```
DownCounter dc = new DownCounter();
Thread t = new Thread(dc)
t.start();                                  // start() NO run();
```

Ventajas y desventajas.

Método 1, extendiendo Thread.

- Es más simple.

Método 2, implementando Runnable.

- Orientado a objetos.
- No hay problema de herencia múltiple.
- Consistencia.

Como termina un *thread*.

- El thread termina cuando el método run() termina.
- El thread sigue existiendo pero en estado *“dead”.*
- NO se puede reiniciar un thread *“dead”.*
- SÍ se pueden invocar los métodos de un thread *“dead”.*

Estados de un *thread*.

- Running — realmente ejecutando.
- Waiting states:
  - Sleeping — esperando voluntariamente.
  - Blocked — esperando involuntariamente.
  - Suspended — se ejecutó suspend( ) (descontinuado).
- Ready — listo para ejecutar.
- Monitor States — varios estados especiales.
- Dead — terminó run().

Prioridades de threads.

- Entero de 1 a 10.
- El Thread Scheduler usa la prioridad para despachar al siguiente thread.

```
int oldPriority = theThread.getPriority();
int newPriority = Math.min(oldPriority+1, Thread.MAX_PRIORITY);
theThread.setPriority(newPriority);
```

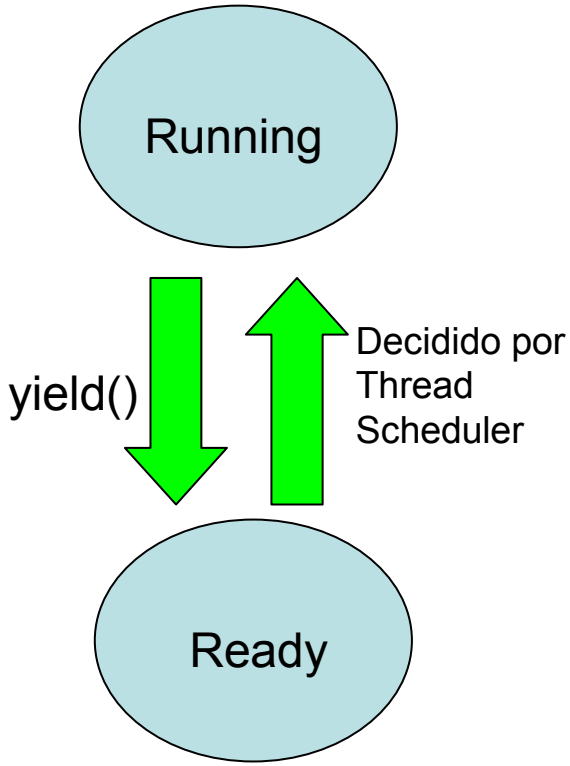
Control de threads.

- Mecanismo para mover threads de estado a estado.
- Principales movimientos para cambiar estado:
  - Yielding
  - Sleeping and waking up.
  - Blocking and then continuing.
  - Waiting and then being notified.
  - Suspending and then resuming (NO USAR métodos suspend() y resume(), están descontinuados).

Yielding.

- Ceder control voluntariamente.
- Ejemplo

```
private void traceRays() {
   for(int j=0; j<300; j++) {
      for(int(i=0;i<300; i++) {
         computeOnePixel(i, j);
         Thread.yield();
      }
   }
}
```

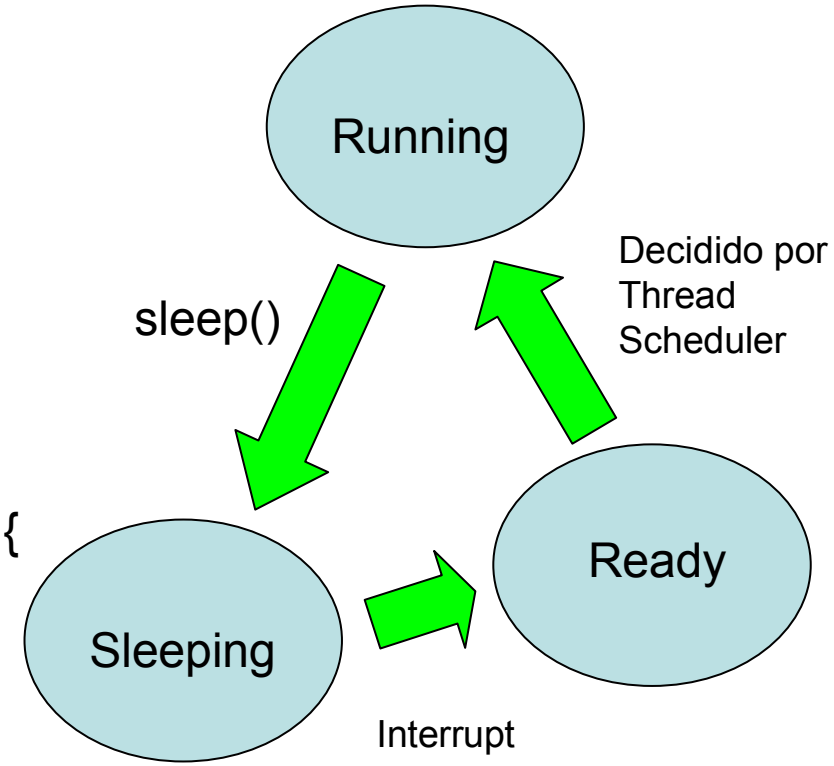


Sleeping and Waking Up.

- Ceder control voluntariamente por un tiempo.

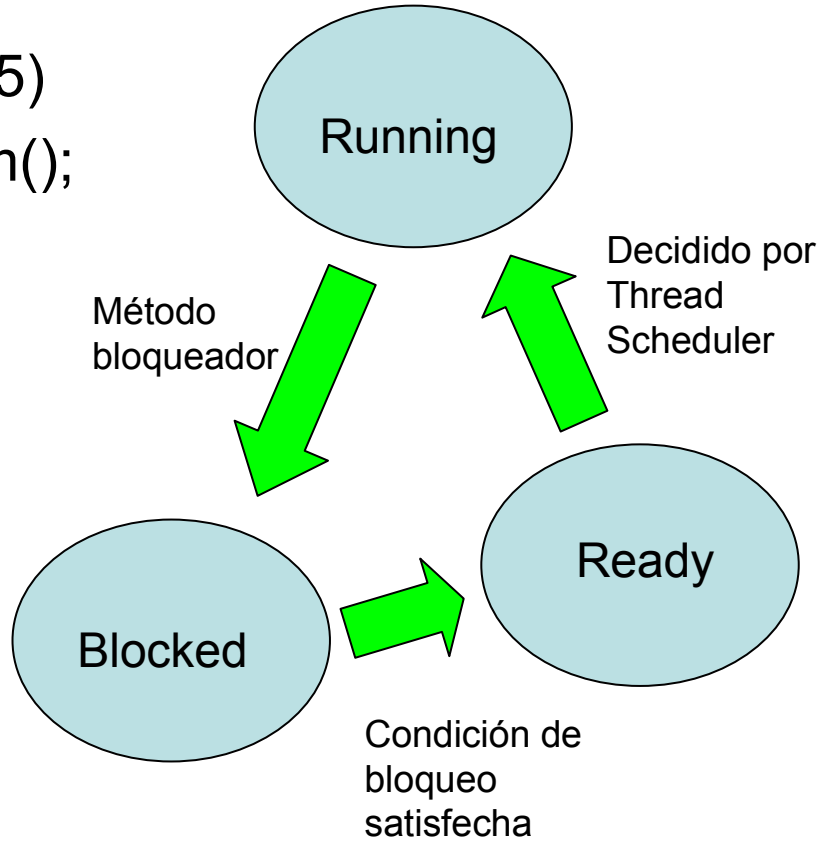
- Ejemplo:

```
private void traceRays() {
    for(int j=0; j<300;j++) {
       for(int(i=0;i<300;i++) {
          computeOnePixel(i, j);
           try {
            Thread.sleep(100);
          } catch (InterruptedException e)  {
                    // sleep interrupted
          }
       }
    }
}
```



Blocking.

```
try {
  Socket sk = new Socket("M1", 5505)
  InputStream is = sk.getInputStream();
  int b = is.read();
} catch (IOException e) {
   // …
}
```



Método join().

```
public static void main(String[ ] args) {
   Thread t = new Thread(new Runner());
    t.start();
   // …
   // Ejecución en paralelo con thread t
   // ...
   try {
      t.join();
   } catch (InterruptedException e) {
     // terminó thread t
   }
   // ...
   // Continúa thread principal
   // ...
}
```

Tipos de Scheduling.

- Preemptive (apropiativo).
  - Time-sliced.
  - Round-robin.
  - Otros algoritmos.
- Non-preemptive.
- La Java Virtual Machine usa el mecanismo del Sistema Operativo.
- UNIX generalmente es preemptive.
- Macinthos usa time-slicing.
- Windows usa combinaciones de ambos (?).

Soporte a Monitores.

- Un monitor es un objeto que puede bloquear y reactivar threads.
- El soporte de Java proporciona los siguientes recursos:
  - Un lock para cada objeto.
  - La palabra clave synchronized.
  - Los métodos:
    - wait()
    - notify()
    - notifyAll()

Locks y sincronización.

- Todos los objetos tienen un lock que controla el acceso a código sincronizado.
- En un momento dado, sólo un thread puede poseer el lock de un objeto.
- Si un thread quiere usar código sincronizado de un objeto, debe obtener el lock del objeto.
- Si el lock está disponible (es decir, si no lo tiene otro thread) se permite.
- Si el lock no está disponible, el thread entra al estado de monitor Seeking Lock.
- El thread pasa al estado Ready cuando el otro thread que tenía el lock lo libera.

Ejemplo sin Sincronización

(puede haber problemas).

```
public class MailConsumer extends Thread {
    private MailBox box ;
    public MailConsumer(MailBox box) {
       this.box = box;
    }
    public void run() {
       while(true) {
          if(box.request) {
             System.out.println(box.message);
             box.request = false;
          }
          try { Thread.sleep(50); } catch(InterruptedException e) {  }
          }
    }
class MailBox {
    public boolean request;
    public String message;
}
```

Ejemplo con Sincronización (1).

```
class MailBox {
    private boolean request;
    private String message;
    public synchronized void SetMessage(String message) {
        this.message = message;
        request = true;
    }
    public synchronized String getMessage() {
        request = false;        // notar orden de los postulados.
        return message;
    }
}
```

Nota. En este caso todo el cuerpo del método está sincronizado.

Ejemplo con Sincronización (2).

```
public class MailConsumer extends Thread {
    private MailBox box ;
    public MailConsumer(MailBox box) {
        this.box = box;
    }
    public void run() {
        while(true) {
          if(box.getRequest()) {
            System.out.println(box.getMessage());
          }
          try {
                Thread.sleep(50);
          }  catch(InterruptedException e) {
             // Espera por interrupción.
          }
 } } }
```

wait() and notify().

- Estos métodos proporcionan un mecanismo para que un monitor pueda detener un thread cuando el monitor no tenga el lock disponible y reactivarlo cuando se vuelva disponible.
- Están definidos en la clase Object.
- Los threads nunca tienen que checar el estado del objeto.
- wait() y notify() sólo pueden ser usados dentro de código sincronizado.
- Un thread que invoca wait() entra a estado waiting y libera el lock.

Ejemplo usando wait().

```
public synchronized String getMessage() {
    while(request == false) {
       try {
           wait();
       } catch(InterruptedException e) {
       }
       request = false;
       return message;
    }
```

El consumer ejecuta:

```
box.getMessage();
```

Ejemplo usando notify().

```
public synchronized void setMessage(String message) {
    this.message = message;
    request = true;
    notify();
}
```

El consumer ejecuta:

box.setMessage(“Texto del Mensaje”);

Deadlocks.

- Se presentan cuando dos threads están esperando cada uno por el lock del otro.
- No se detectan o evitan automáticamente.
- Se pueden evitar:
  - Decidiendo el orden en que se adquieren los locks.
  - Respetando el orden cuidadosamente.
  - Liberando los locks en el orden inverso precisamente.

Ejemplo completo: Producer.

```
public class Producer implements Runnable {
   private SyncStack theStack;
   private int num;
   private static int counter = 1;
   public Producer (SyncStack s) {
      theStack = s;
      num = counter++;
   }
   public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
       c = (char)(Math.random() * 26 +'A');
       theStack.push(c);
      System.out.println("Producer" + num + ": " + c);
      try { Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {  }
```

Ejemplo completo: Consumer.

```
public class Consumer implements Runnable {
   private SyncStack theStack;
   private int num;
   private static int counter = 1;
   public Consumer (SyncStack s) {
     theStack = s;
    num = counter++;  }
    public void run() {
       char c;
       for (int i = 0; i < 200; i++) {
       c = theStack.pop();
       System.out.println("Consumer" + num + ": " + c);
       try {  Thread.sleep((int)(Math.random() * 300));
       } catch (InterruptedException e) { }
   }
```

Ejemplo completo: Monitor.

```
import java.util.*;
public class SyncStack {
   private List buffer = new ArrayList(400);
   public synchronized char pop() {
      char c;
      while (buffer.size() == 0) {
        try { wait(); } catch (InterruptedException e) {  }
      }
     c = ((Character)buffer.remove(buffer.size()-1)).charValue();
     return c;
   }
   public synchronized void push(char c) {
     notify();
     Character charObj = new Character(c);
     buffer.add(charObj);
   }
```

Ejemplo completo: Probador.

```
public class SyncTest {
   public static void main(String[ ] args) {
      SyncStack stack = new SyncStack();
      Producer p1 = new Producer(stack);
     Thread prodT1 = new Thread (p1);
     prodT1.start();
     Producer p2 = new Producer(stack);
     Thread prodT2 = new Thread (p2);
     prodT2.start();
     Consumer c1 = new Consumer(stack);
     Thread consT1 = new Thread (c1);
     consT1.start();
     Consumer c2 = new Consumer(stack);
     Thread consT2 = new Thread (c2);
     consT2.start();
   }
```

Otra manera de sincronizar.

- Los siguientes segmentos de código son equivalentes:

```
public void push(char c) {
        synchronized(this) {
        // …
        // …
}
```

y

```
public synchronized void push(char c) {
        // …
        // …
}
```

- Es posible también sincronizar dependiendo de otro objeto.

Métodos descontinuados.

- Los siguientes métodos de la clase Thread están descontinuados y no se deben utilizar:
  - suspend()
  - resume()
  - stop()
- Ver artículo Why are Thread.stop, Thread.suspend and Thread.resume Deprecated? en la documentación de la API

Repaso.

- Concepto de thread y multithreading.
- Mecanismos de inicio y terminación de threads.
- Prioridades.
- Control de threads.
- Tipos de Scheduling.
- Soporte de Monitores.
- Locks y sincronización.

Módulo 15.

Manejo de entrada y salida,
I/O Streams.

Objetivos.

- Describir las principales características del paquete java.io.
- Construir y utilizar streams de tipos nodo y proceso apropiadamente.
- Distinguir readers y writers de streams, y seleccionar adecuadamente entre ellos.

Fundamentos de E/S.

- Un *stream* es un flujo de datos de una fuente *(source)* a un destino *(sink).*
- Un stream *source* inicia el flujo de datos y se conoce también como *input stream*.
- Un stream *sink* termina el flujo de datos y se conoce también como *output stream*.

Stream	**Byte Streams**	**Character Streams**
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer

Datos en Streams.

- La tecnología Java supporta dos tipos de streams:
  - byte streams y
  - character streams.
- La entrada y la salida de datos binarios la manejan los *input byte streams* y los *output byte streams.*
- La entrada y la salida de datos de texto la manejan los *character streams* llamados *readers* y *writers*.
- Normalmente, el término *stream* se refiere a *byte streams.*
- Los términos *reader* y *writers* se refieren a *character streams.*

Métodos de los InputStreams.

- Tres métodos básicos:
  - int read()
  - int read(byte[ ] buffer)
  - int read(byte[ ] buffer, int offset, int length)

- Varios métodos adicionales:
  - void close()
  - int available()
  - skip(long n)
  - boolean markSupported()
  - void mark(int readlimit)
  - void reset()

Métodos de los OutputStreams.

- Tres métodos básicos:
  - void write(int c)
  - void write(byte[ ] buffer)
  - void write(byte[ ] buffer, int offset, int length)
- Dos métodos adicionales:
  - void close()
  - void flush()

Métodos de los Readers.

- Tres métodos básicos:
  - int read()
  - int read(char[ ] cbuf)
  - int read(char[ ] cbuf, int offset, int length)

- Varios métodos adicionales:
  - void close()
  - boolean ready()
  - skip(long n)
  - boolean markSupported()
  - void mark(int readAheadLimit)
  - void reset()

Métodos de los Writers.

- Cinco métodos principales:
  - void write(int c)
  - void write(char[ ] cbuf)
  - void write(char[ ] cbuf, int offset, int length)
  - void write(String string)
  - void write(String string, int offset, int length)
- Métodos adicionales:
  - void close()
  - void flush()

Streams tipo Nodo.

Tipo	Character Streams	Byte Streams
File	FileReader FileWriter	FileInputStream FileOutputStream
Memory Array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memory String	StringReader StringWriter	No existen
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

Ejemplo Simple (1).

java TestNodeStreams file1 file2

```
1 import java.io.*;
2
3 public class TestNodeStreams {
4 public static void main(String[ ] args) {
5    try {
6        FileReader input = new FileReader(args[0]);
7        FileWriter output = new FileWriter(args[1]);
8        char[ ] buffer = new char[128];
9        int charsRead;
10
11       // read the first buffer
12       charsRead = input.read(buffer);
13
```

Ejemplo Simple (2).

```
14          while ( charsRead != -1 ) {
15              // write the buffer out to the output file
16              output.write(buffer, 0, charsRead);
17
18                  // read the next buffer
19                  charsRead = input.read(buffer);
20          }
21
22          input.close();
23          output.close();
24    } catch (IOException e) {
25        e.printStackTrace();
26    }
27 }
```

Ejemplo con Stream de Buffer (1).

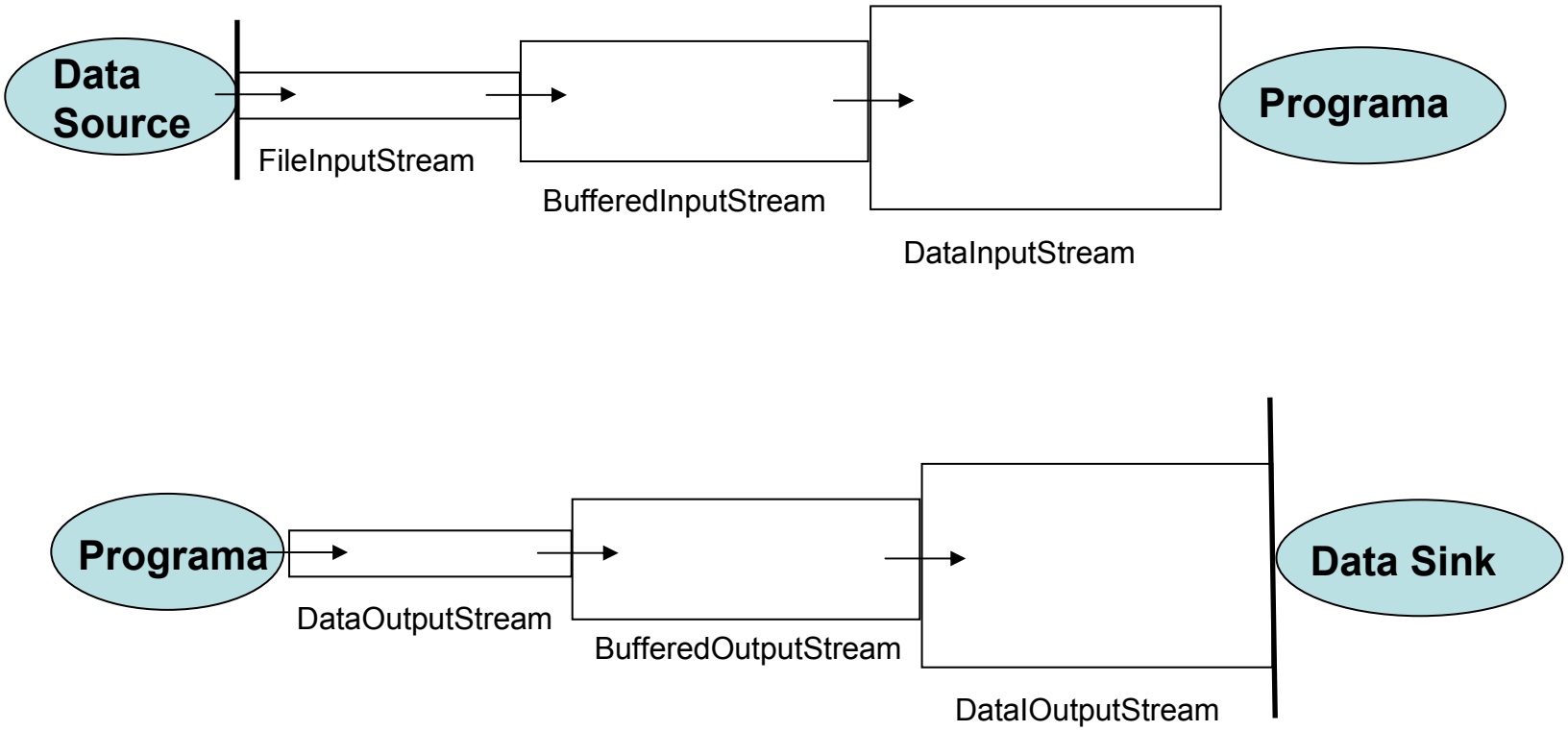
java TestBufferedStreams file1 file2

```
1 import java.io.*;
2
3 public class TestBufferedStreams {
4 public static void main(String[ ] args) {
5 try {
6     FileReader input = new FileReader(args[0]);
7     BufferedReader bufInput = new BufferedReader(input);
8     FileWriter output = new FileWriter(args[1]);
9     BufferedWriter bufOutput = new BufferedWriter(output);
10    String line;
11
12    // read the first line
13    line = bufInput.readLine();
14
```

Ejemplo con Stream de Buffer (2).

```
15      while ( line != null ) {
16            // write the line out to the output file
17            bufOutput.write(line, 0, line.length());
18            bufOutput.newLine();
19
20             // read the next line
21            line = bufInput.readLine();
22       }
23
24      bufInput.close();
25      bufOutput.close();
26      } catch (IOException e) {
27            e.printStackTrace();
28       }
29   }
30 }
```

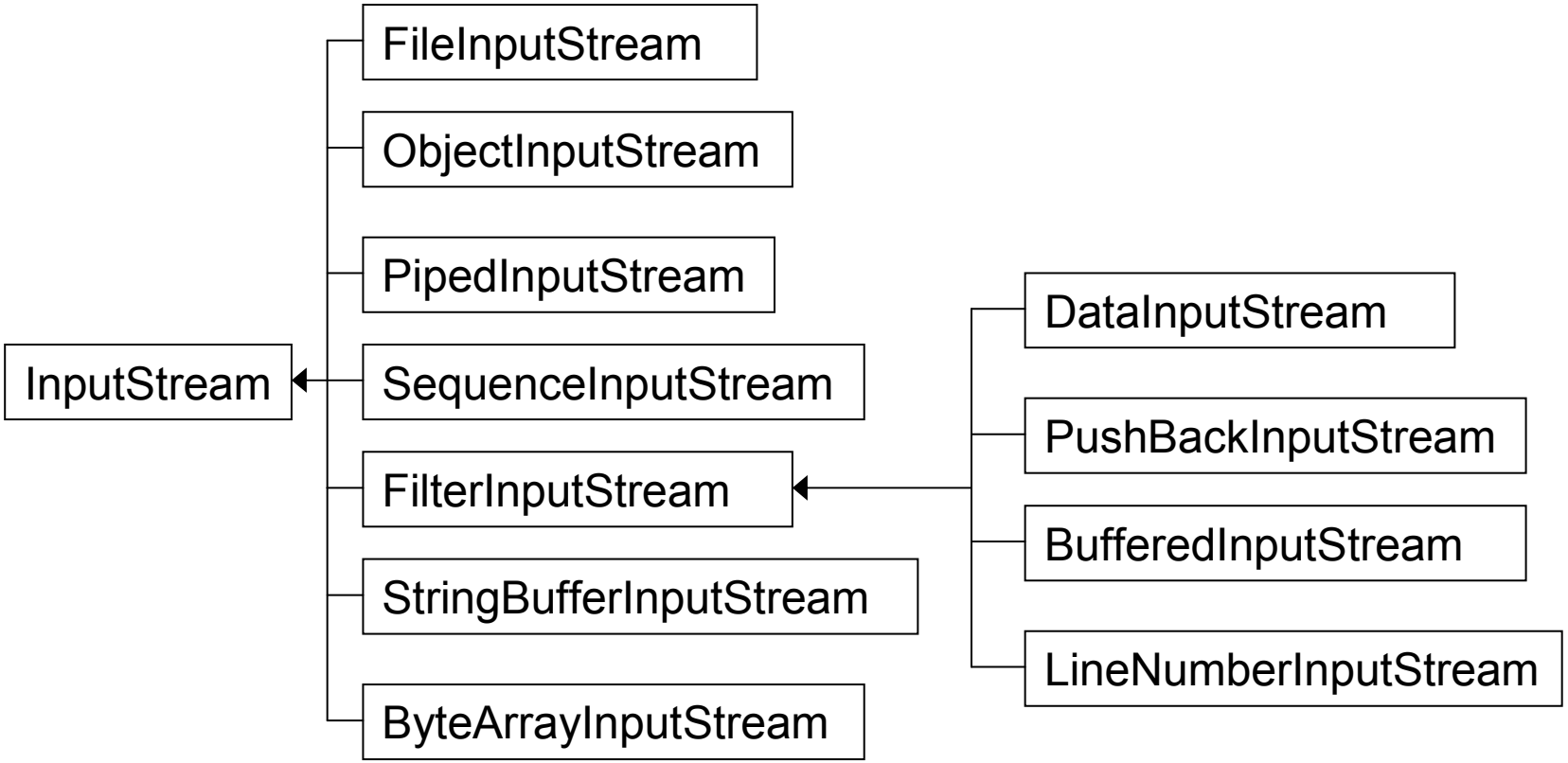
Encadenamiento de Streams.



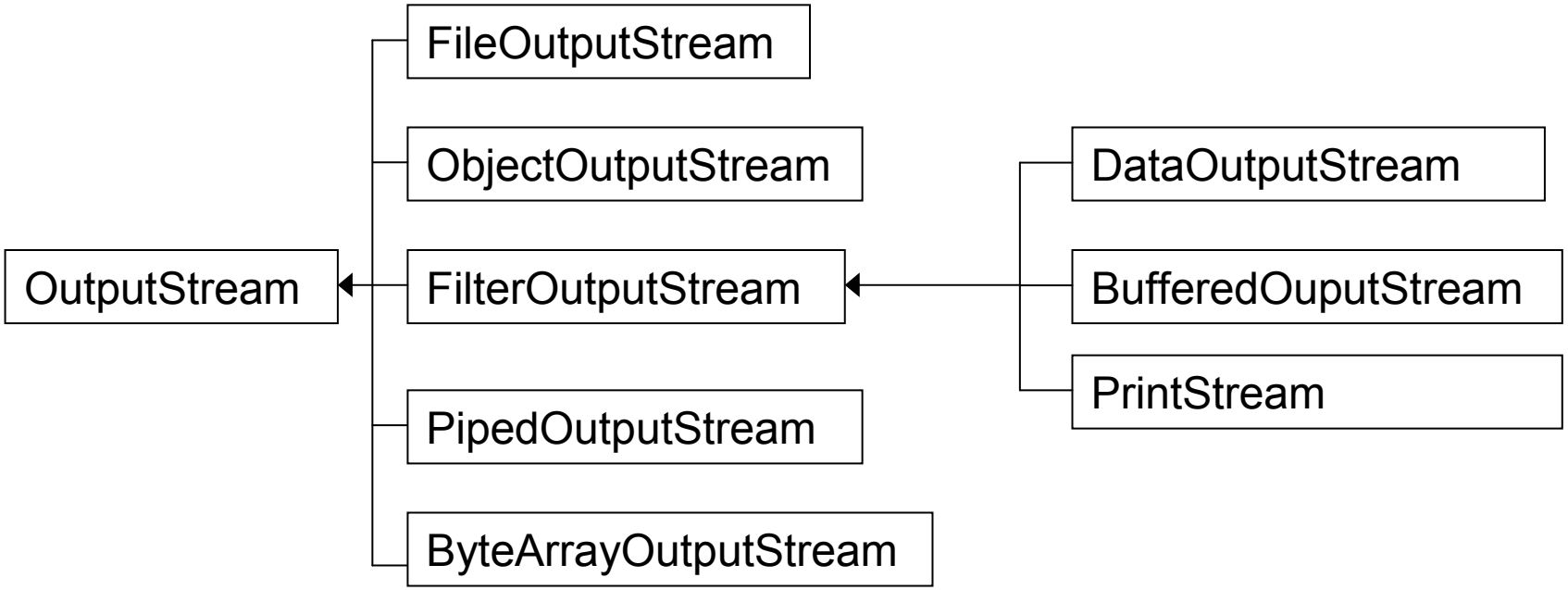
Streams tipo filtro.

Tipo	Character Streams	Byte Streams
Bufferización	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtro	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Conversión de bytes a caracteres	InputStreamReader OutputStreamWriter	No existen
Serialización de objetos	No existen	ObjectInputStream ObjectOutputStream
Conversión de datos	No existen	DataOuputStream DataInputStream
Enumeración	LineNumberReader	LineNumberInputStream
"Peeking"	PushBackReader	PushBackInputStream
Impresión	PrintWriter	PrintStream

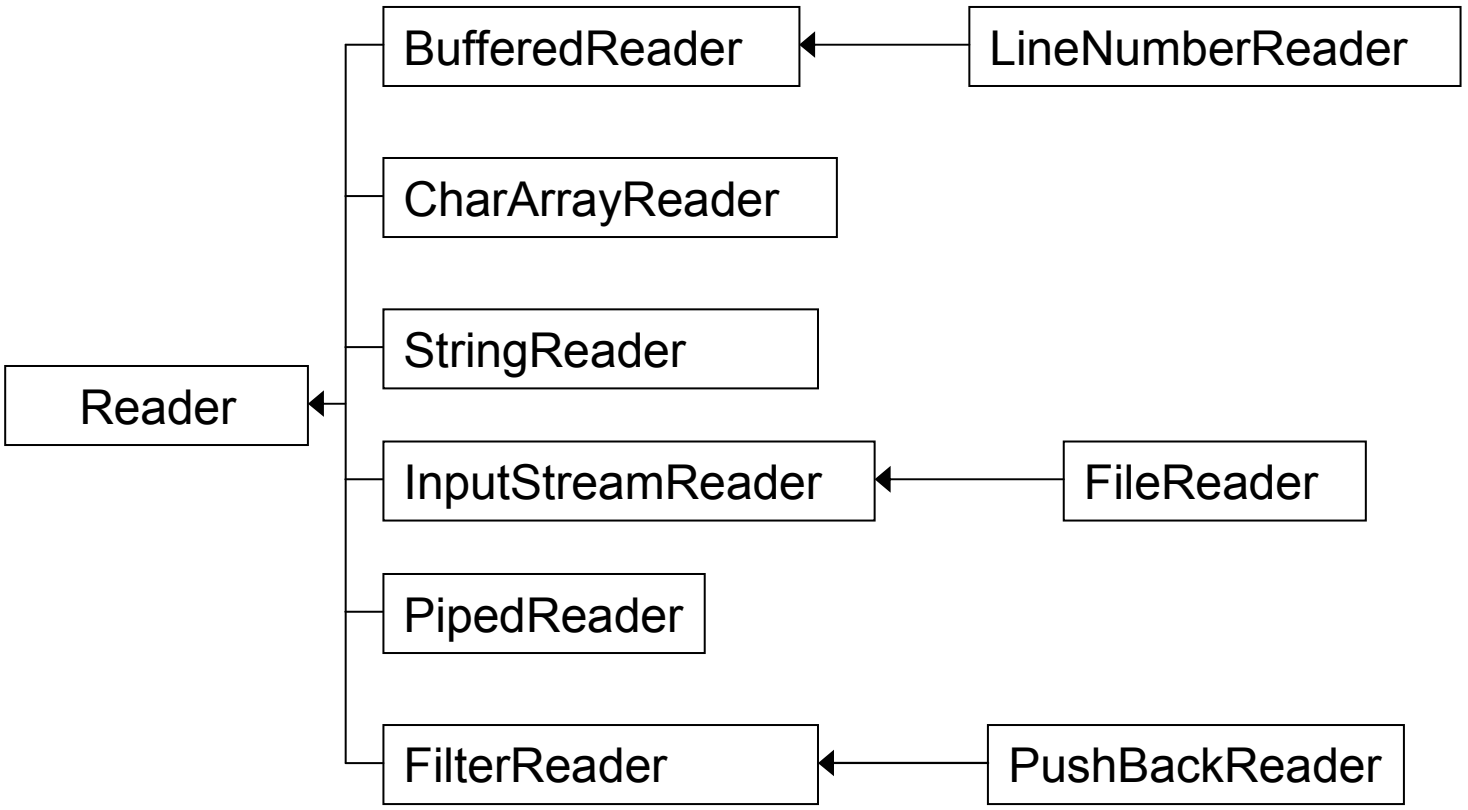
Jerarquía de InputStreams.



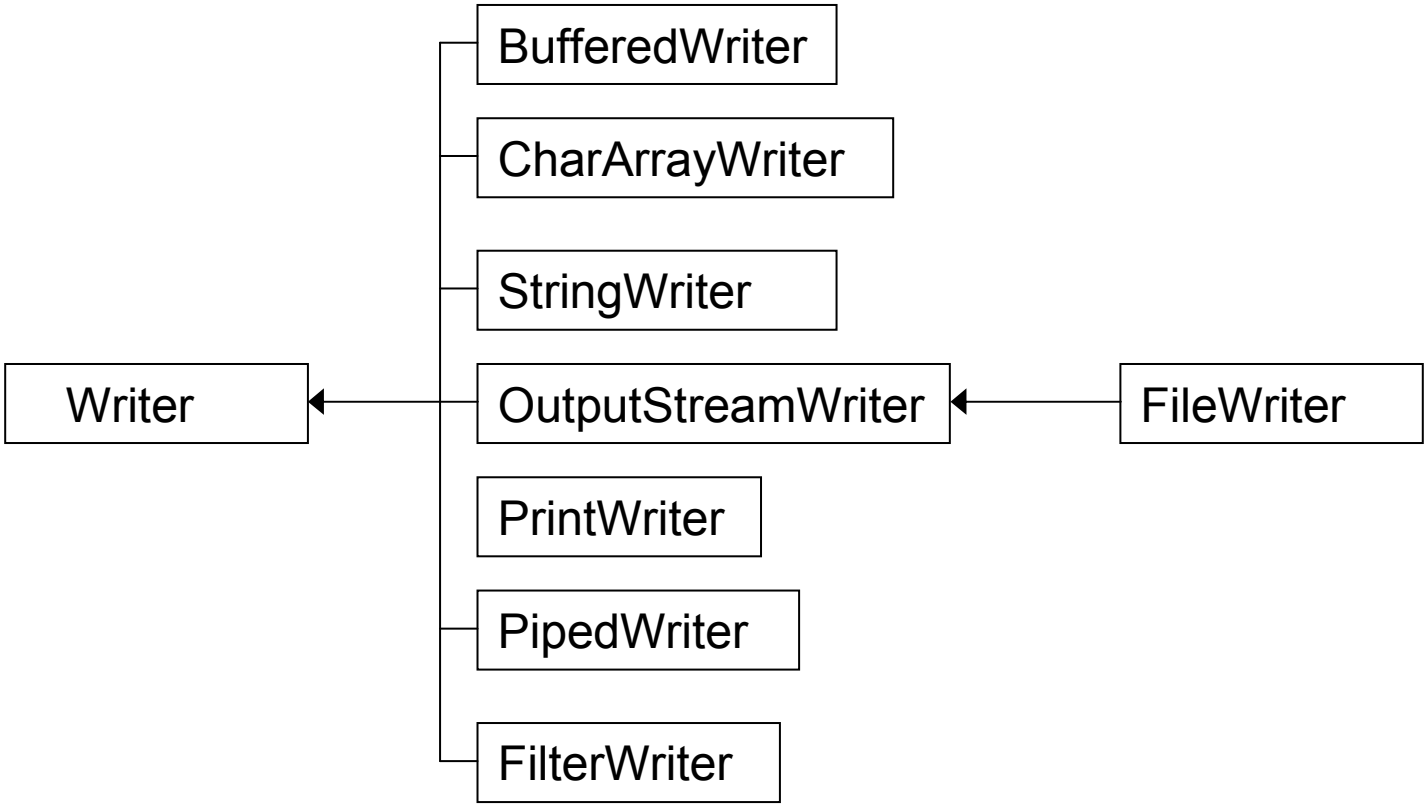
Jerarquía de OutputStreams.



Jerarquía de Readers.



Jerarquía de Writers.



Repaso.

- Características principales del paquete java.io.
- Construcción y uso adecuado de streams tipo nodo y tipo filtro.
- Distinción y selección adecuado de readers y writers y streams de bytes

Módulo 16.

Conceptos de Networking con Java.

Objetivos.

- Entender los conceptos básicos de networking en Java.
- Desarrollar código para establecer conexiones entre clientes y servidores utilizando las clases Socket y ServerSocket.
- Estudiar ejemplos de un cliente y un servidor simples programados en Java.
- Entender el mecanismo de Java RMI: Remote Method Invocation.

Protocolo.

- Basado en TCP/IP.
- Sockets.
- Puertos.

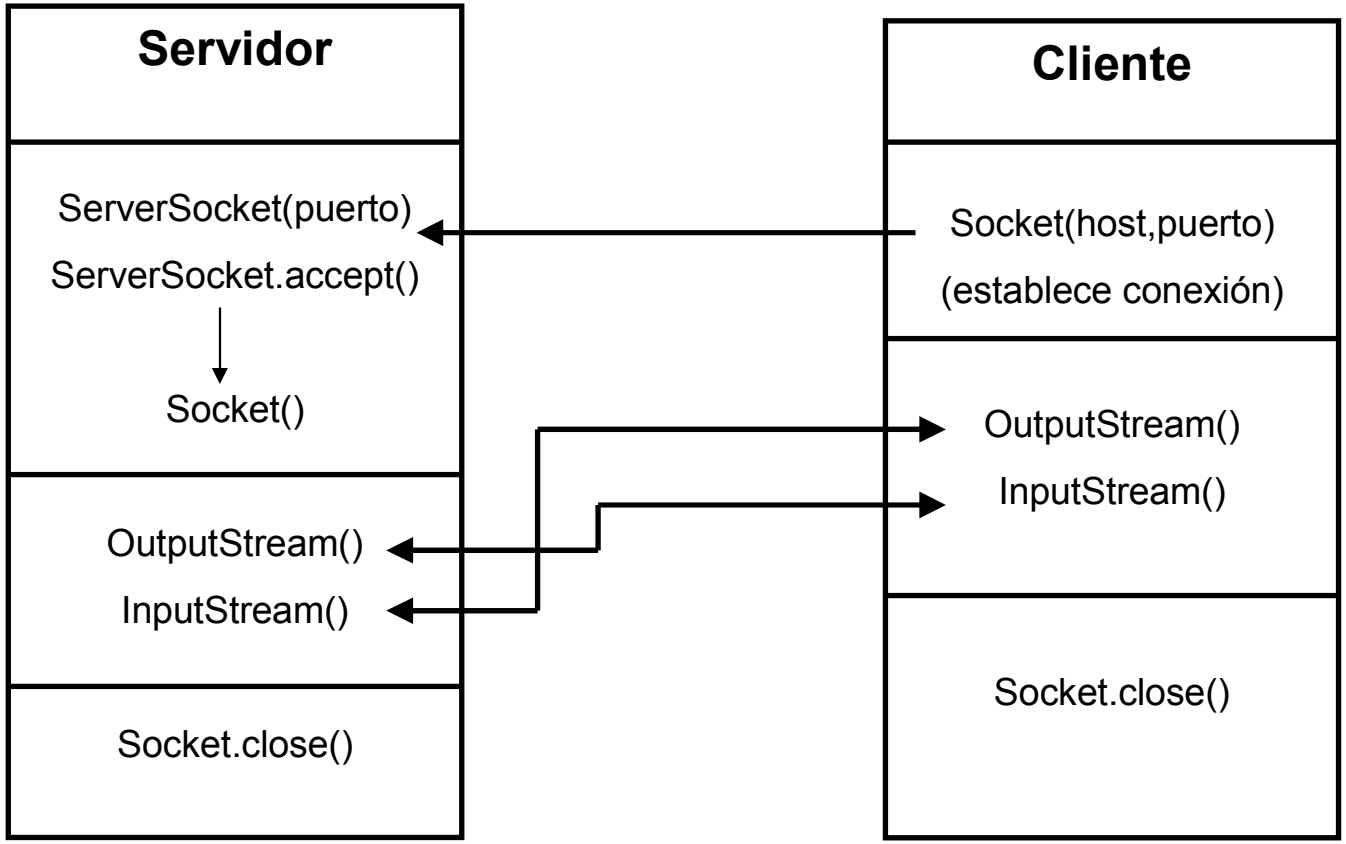
Sockets.

- Java usa el concepto de Sockets de TCP/IP para comunicar un cliente con un servidor.
- Un Socket contiene dos streams, un input stream y un output stream.
- El servidor crea un **ServerSocket** y espera por conexiones de los clientes.
- El cliente crea un **Socket** que se conecta con el servidor.
- Cuando se establece la conexión, se crea un **Socket** en el Servidor.

Puertos.

- Un servidor tiene varios puertos, que son los puntos de conexión de los clientes.
- Cada cliente se conecta al servidor usando un puerto determinado.
- Los puertos se identifican mediante números de 0 a 65535.
- Se recomienda no usar números pequeños porque éstos son usados para servicios standard.

Modelo de Networking de Java.



Ejemplo de Server mínimo.

```
import java.net.*;
import java.io.*;
public class SimpleServer {
    public static void main(String[ ] args) {
      ServerSocket s = null;
      try {
        s = new ServerSocket(5432);      // Registra el servicio en el
                                         // puerto 5432
      } catch (IOException e) {
        e.printStackTrace();
      }
```

Ejemplo de Server mínimo (2).

```
while (true) {      // Ciclo infinito para escuchar a los clientes.
  try {
    Socket s1 = s.accept();     //  Aquí espera por la conexión y
                                // recibe socket
    OutputStream os = s1.getOutputStream();
    DataOutputStream dos = new DataOutputStream(os);
    dos.writeUTF("Hello Net World!");   // envía mensaje al cliente.
    dos.close();                                // cierra output stream.
    s1.close();                  // cierra Socket, pero no ServerSocket
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Ejemplo de Cliente mínimo.

```
import java.net.*;
import java.io.*;
public class SimpleClient {
    public static void main(String args[]) {
    try {
       Socket s1 = new Socket("127.0.0.1", 5432);
       InputStream is = s1.getInputStream();
       DataInputStream dis = new DataInputStream(is);
       System.out.println(dis.readUTF());
       dis.close();
       s1.close();
```

Ejemplo de Cliente mínimo (2).

```
        } catch (ConnectException e1) {
          System.err.println("Could not connect to the server.");
          e.printStackTrace();
        } catch (IOException e) {
          e1.printStackTrace();
        }
    }
}
```

Java Remote Method Invocation.

- Java RMI permite ejecutar código Java en otra máquina conectada a la red.
- Alto grado de abstracción que simplifica enormemente la tarea.
- RMI visualiza objetos que residen en otra máquina y a los cuales se les envían mensajes *igual* que si fueran locales.
- RMI utiliza interfaces para enmascarar la implementación.

Interfaces Remotas.

- Deben ser public.
- Deben extender la interfaz java.rmi.Remote
- Cada método de la interfaz remota debe declarar java.rmi.RemoteException en su cláusula throws (además de otras excepciones que pueda usar).
- Un objeto remoto usado en un objeto local se declara como de tipo de la interfaz y no de la clase real.
- Ejemplo:

```
import java.rmi.*;
 public interface PerfectTimeInt extends Remote {
    public long getPerfectTime() throws RemoteException;
}
```

Implementación de las Interfaces Remotas.

- El server debe contener una clase que:
  - extienda UniCastRemoteObject.
  - implemente la Interfaz Remota.
- Puede tener métodos adicionales pero sólo los métodos definidos en la interfaz remota pueden ser vistos por el cliente.
- Debe incluir un constructor que lance RemoteException.
- El cliente obtiene solamente una referencia a la interfaz, no a la clase que la implementa.

RMI Security Manager

- Es necesario crear y activar un Security Manager para RMI, cuya función es validar los requerimientos de los clientes remotos.
- El que viene con la distribución de Java se llama RMISecurity Manager.
- El usuario puede crear otros.
- Usar el método setSecurityManager de la clase System en el servidor:

```
System.setSecurityManager(new RMISecurityManager());
```

RMI Registry.

- El RMI registry contiene los nombres de las máquinas, puertos y servicios que pueden participar en la aplicación.
- Para arrancar el registry en Windows:

  start rmiregistry [puerto]

- El puerto de default es 1099.

RMI Binding.

- Para registrar una aplicación en el RMI registry se usa el método estático bind() de la clase Naming:

  Naming.bind("//host:port/Service", remoteObj);

  donde Service es un nombre arbitrario para el servicio.

- O usando el puerto de default:

  Naming.bind("//host/Service", remoteObj);

- O usando el sistema local:

  Naming.bind("Service", remoteObj);

- También se puede usar el método rebind()

Ejemplo de Servidor RMI.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;
public class PerfectTime extends UnicastRemoteObject
                                                    implements PerfectTimeInt {
    public long getPerfectTime() throws RemoteException {
        return System.currentTimeMillis();
    }
    public PerfectTime() throws RemoteException { }
    public static void main(String[ ] args) throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        PerfectTime pt = new PerfectTime();
        Naming.bind("PerfectTime", pt);
        System.out.println("Ready to serve time…");
    }
}
```

Stubs y Skeletons.

- Son clases que se encargan de la transmisión real de los objetos del cliente al servidor y viceversa.
- Los stubs se agregan del lado del cliente.
- Los skeletons se agregan del lado del servidor.
- Se generan automáticamente por medio del compilador de RMI (que se encuentra en el directorio bin del jdk):

```
rmic  ServerName
```

Ejemplo de cliente RMI.

```
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
   public static void main(String[ ] args) throws Exception {
        System.setSecurityManager(new MiSecurityManager());
        PerfectTimeInt  t  =
                    (PerfectTimeInt)Naming.lookup(“PerfectTime");
        for (int i=0; i<10; i++) {
           System.out.println("Perfect Time: " + t.getPerfectTime());
        }
    }
}
```

Pasos a seguir para ejecutar el ejemplo de RMI.

1. **En una ventana de comandos arrancar el RMIRegistry:**

   start rmiregistry

   (se despliega una ventana donde corre el registry, la cual se debe minimizar y dejar corriendo).

2. **En la ventana de comandos donde se arrancó el registry (no en la que está corriendo el registry), teclear:**

   javac DisplayPerfectTime.java
   rmic PerfectTime
   java PerfectTime

3. **En otra ventana de comandos, correr el cliente:**

   java DisplayPerfectTime

Ejercicio.

1. Terminar la aplicación del “chat room” estableciendo la conexión con el servidor.

Repaso.

- Modelo de Networking de Java basado en TCP/IP.
- Clases ServerSocket y Socket para implementar servidores y clientes.
- Java RMI permite ejecutar métodos en servidores remotos de una manera similar a la ejecución de métodos locales.