Servicios de Formación
Módulo 1. Programación Java Básico

www.iconotc.com
info@iconotc.com
Tfno. +34 912 986 176

Instructor: Rodolfo Campos

Índice

- Introducción
- Programación orientada a objetos
- Identificadores, palabras clave y tipos
- Expresiones y control de flujo
- Matrices
- Diseño de clases
- Funciones de clases avanzadas
- Excepciones y aserciones
- API Collections y Generics
- E/S de consola y E/S de archivos
- Creación de interfaces Java con la API de Swing

Índice

- Manejo de eventos generados por la interfaz gráfica
- Aplicaciones basadas en la interfaz gráfica
- Threads
- Redes

¿Qué es Java?

Java es una plataforma tecnológica, compuesta de:

- Un lenguaje de programación. Podemos utilizar este lenguaje de programación para crear una gran variedad de aplicaciones.
- Un ambiente de desarrollo, proporciona una amplia gama de herramientas: un compilador, un interpretador, un generador de documentación, herramientas de empaquetamiento de clases.
- Un ambiente de aplicación, o JRE (Java Runtime Environment), ambiente de ejecución de Java, una máquina virtual que permite ejecutar programas escritos en Java.

¿Qué es Java?

- Un ambiente de despliegue, los elementos y herramientas necesarias para desplegar y hacer disponibles los programas Java dentro de las computadoras que tengan la maquina virtual.

La Máquina Virtual de Java (JVM)

- Es una máquina, o programa, que proporciona el comportamiento genérico a las aplicaciones desarrolladas en Java, es decir, que sean independientes de las plataformas de hardware o software donde se ejecutan tales aplicaciones.
- Una especificación, es quien provee las definiciones concretas para la implementación de dicha máquina virtual, de manera que, todo código Java, al ser compilado, pueda ser ejecutado exitosamente en cualquier plataforma.
- El diseño de la JVM permite la creación de implementaciones para múltiples ambientes operativos. Por ejemplo, Sun Microsystems proporciona implementaciones de la JVM para los sistemas operativos Solaris OS, Linux y Microsoft Windows

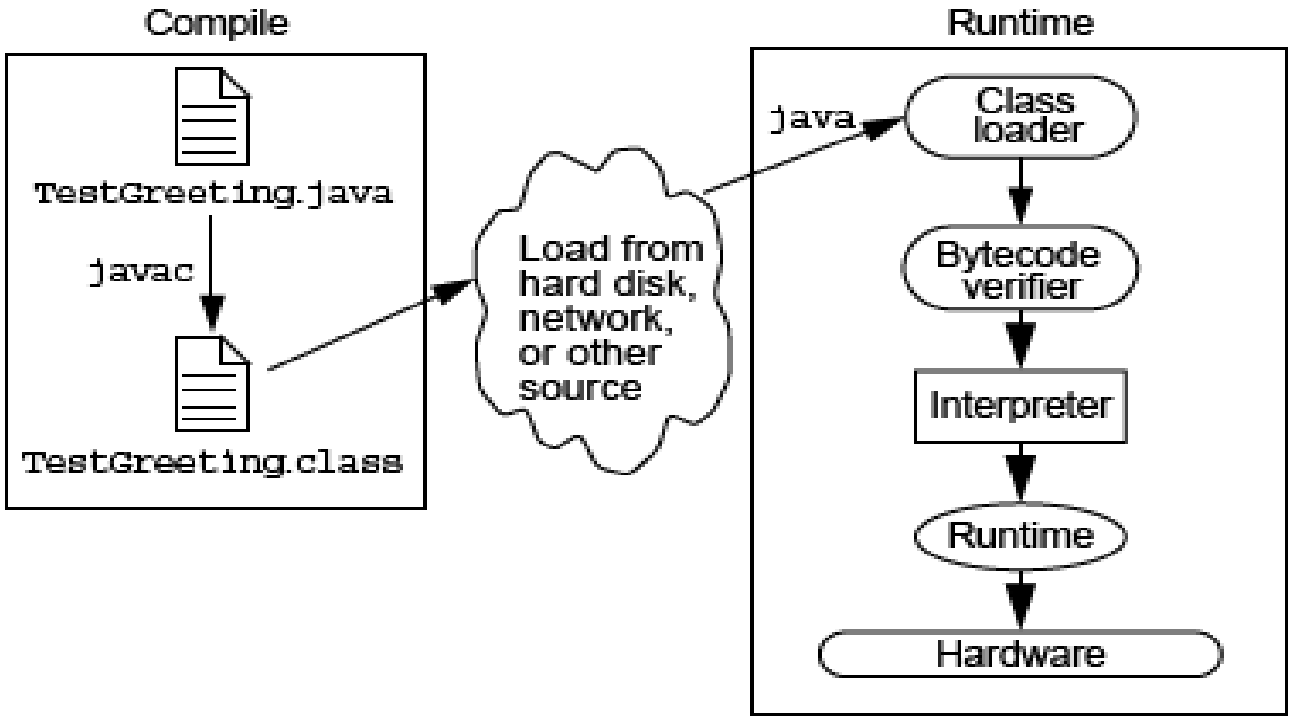
La Recolección de Basura

- Muchos lenguajes de programación permiten que la memoria sea asignada dinámicamente, en tiempo de ejecución. Haciendo uso de apuntador a direcciones de bloques de memoria. Los bloques de memoria no requeridos, deberían ser liberados, ya sea por el programa o el ambiente de ejecución.
- Generalmente, la responsabilidad de esa liberación, recae en el desarrollador. Esta liberación puede ser difícil de llevar a cabo, usualmente no se sabe, a futuro, cuando un espacio de memoria dejará de ser usado.
- Si no se liberan los bloques de memoria, los programas eventualmente terminan fallando, la memoria para asignar se agota (memory leaks).

La Recolección de Basura

- Java, proporciona un mecanismo de rastreo y vigilancia de cada asignación de memoria, liberando al desarrollador de la responsabilidad de tal proceso. La JVM chequea y libera cualquier espacio de memoria que pueda ser liberado.
- Este proceso es llamado, Recolección de Basura, y se ejecuta automáticamente durante la vida de un programa Java.

El Ambiente de Ejecución



Una aplicación sencilla

Aplicación TestGreeting.java

```
//
// Ejemplo de una aplicación "Hello World"
//
public class TestGreeting {
     public static void main (String[] args) {
          Greeting hola = new Greeting();
          hola.saludo();
     }
}
```

Una aplicación sencilla

Clase Greeting.java

```
public class Greeting {
     public void saludo () {
          System.out.println("¡hola mundo!");
     }
}
```

Compilación y ejecución

Luego de creado el código fuente `TestGreeting.java`, se compila con el siguiente comando:

javac TestGreeting.java

Si el compilador no retorna algún mensaje, el nuevo archivo `TestGreeting.class` se almacenará en el mismo directorio que el código fuente, al menos que se especifique de otra manera. El archivo `Greeting.java` ha sido compilado en `Greeting.class`. Esto es realizado automáticamente por el compilador.

Para ejecutar la aplicación `TestGreeting`, se hace uso del interpretador de Java.

java TestGreeting

Ejercicio

Programación Orientada a Objetos

- Permite expresar un programa como un conjunto de objetos, que colaboran entre ellos para realizar tareas específicas. Esto permite hacer las aplicaciones y sus componentes internos, más fáciles de escribir, mantener y reutilizar.
- Abstracción
- Permite expresar las características esenciales de un objeto, que permitan distinguir dicho objeto de los demás.
- Mediante una clase, se define el conjunto de datos de los elementos (atributos) que definen al objeto, así como el conjunto de comportamientos o funciones (métodos) que permiten manipular al objeto o realizar interacciones entre objetos relacionados.

Declaración de una clase Java

```
<modificador> class <nombre_clase> {
      <declaracion_atributo>*
      <declaracion_constructor>*
      <declaracion_método>*
}
```

Hay varias posibilidades para `<modificador>`, por ahora, solo usaremos `public`. Este modificador declara que la clase es accesible al universo. El cuerpo de la clase declara el conjunto de atributos para los datos, los constructores, y los métodos asociados con ella.

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

Declaración de Atributos

```
<modificador>* <tipo_dato> <nombre_atributo> [ =
<valor_inicial> ];
```

Hay varias posibilidades para `<modificador>`, por ahora, usaremos `public` o `private`. El modificador `private` declara que el atributo es solo accesible para los métodos dentro de la clase. El `<tipo_dato>` del atributo puede ser cualquier tipo primitivo (int, float, etc) u otra clase.

```
public class Foo {
      private int x;
      private float y = 10000.0F;
      private String name = "Bates Motel";
}
```

Declaración de Métodos

```
<modificador>* <tipo_retorno> <nombre_método>
( <argumento>* ) {
      <instrucción>*
}
```

`<modificador>` puede ser `public`, protected o `private`. El modificador `public` indica que puede ser llamado desde otra clase. Los métodos `private` pueden ser llamados solamente por otros métodos de esa misma clase. El modificador de acceso `protected` se describirá luego.

`<tipo_retorno>` indica el tipo de valor retornado por el método. Si el método no retorna un valor, debe ser declarado void.

La lista de `<argumento>` permite indicar los valores de los argumentos de entrada del método. Los elementos de esta lista están separados por coma, mientras que cada elemento consiste en un tipo de datos y un identificador.

Acceso a los miembros de un Objeto

```
public class TestDog {
      public static void main(String[] args) {
            Dog d = new Dog();
            System.out.println("Peso del perro d es " +
                                          d.getWeight());

            d.setWeight(42);
            System.out.println("Peso del perro d es " +
                                          d.getWeight());

            d.setWeight(-42);
            System.out.println("Peso del perro d es " +
                                          d.getWeight());
      }
}
```

Encapsulamiento

Supongamos que tenemos una clase llamada `MiFecha` que incluye estos atributos: `día`, `mes` y `anio`.

```
public class MiFecha {
      public int dia;
      public int mes;
      public int anio;
}
```

Luego, se puede acceder a los atributos directamente y cometer errores, por ejemplo (suponiendo una variable d del tipo `MiFecha`):

```
d.dia = 32;                          // día inválido
d.mes = 2; d.dia = 30;               // posible pero erróneo
d.dia = d.dia + 1;                   // no hay validación
```

Encapsulamiento

Para resolver este problema, se esconde los atributos de datos, haciéndolos privados y proporcionando método de recuperación (getters) y métodos de almacenamiento (setters).

Estos métodos permiten a la clase modificar los valores internos, pero más importante, verificar que los cambios solicitados, son válidos. Por ejemplo:

```
d.setDia(32);                    //día inválido, retorna falso

d.setMes(2); d.setDia(30)  // posible pero erróneo,
                                 // setDia retorna false

d.setDia(d.getDia() + 1);  // retornará false si el
chequeo
```

Constructores

Un constructor es un conjunto de instrucciones diseñadas para inicializar una instancia. Los parámetros son pasados al constructor en la misma forma que a un método.

```
[<modificador>] <nombre_clase> ( <argumento>* ) {
        <instrucción>*
}
```

El nombre del constructor debe ser siempre el mismo que el nombre de la clase. Si se utiliza un `<modificador>` los valores válidos son `public`, `protected` y `private`.

Constructores

```
public class Perro {
      private int peso;

      public Perro() {
            perro = 42;
      }
}
```

Los constructores no son métodos, ellos no retornan valores y no son heredados. Toda clase tiene al menos un constructor, el llamado Constructor por Defecto, no recibe argumentos y no tiene instrucciones, es proporcionado por Java.

Distribución del Código

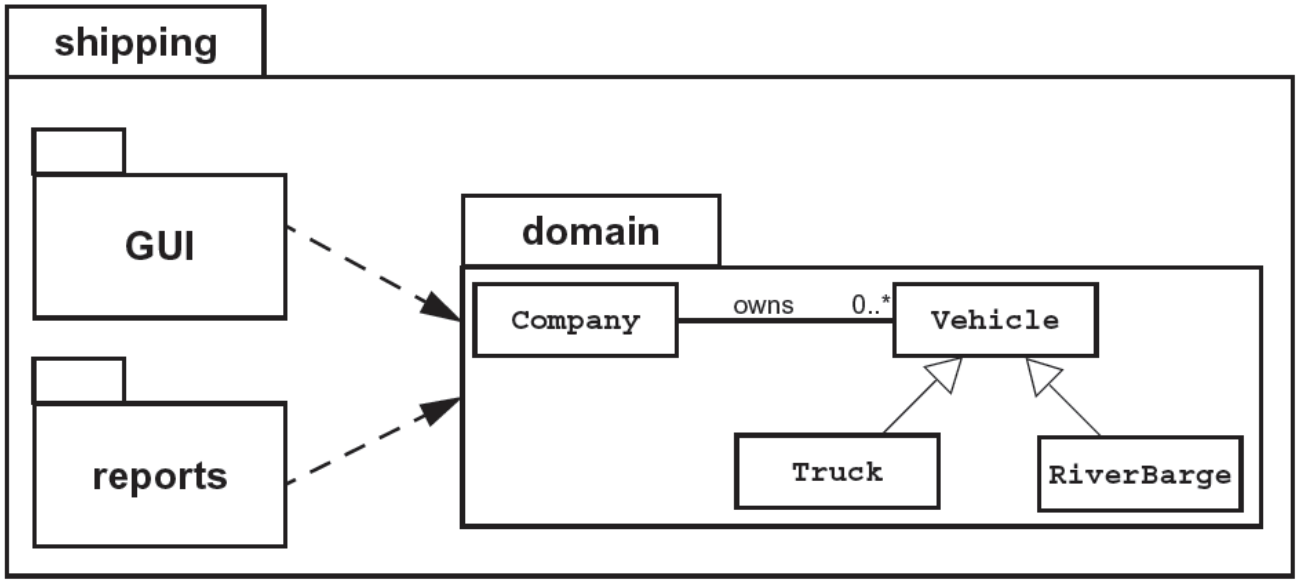
Un archivo de código fuente de Java toma la siguiente forma.

```
[<declaracion_paquete>]
<declaracion_import>*
<declaracion_clase>+
```

Archivo VehicleCapacityReport.java

```
package shipping.reports;
import shipping.domain.*;
import java.util.List;
import java.io.*;
public class VehicleCapacityReport {
        private List vehicles;
        public void generateReport(Writer output) {
                // código para generar el reporte
        }
}
```

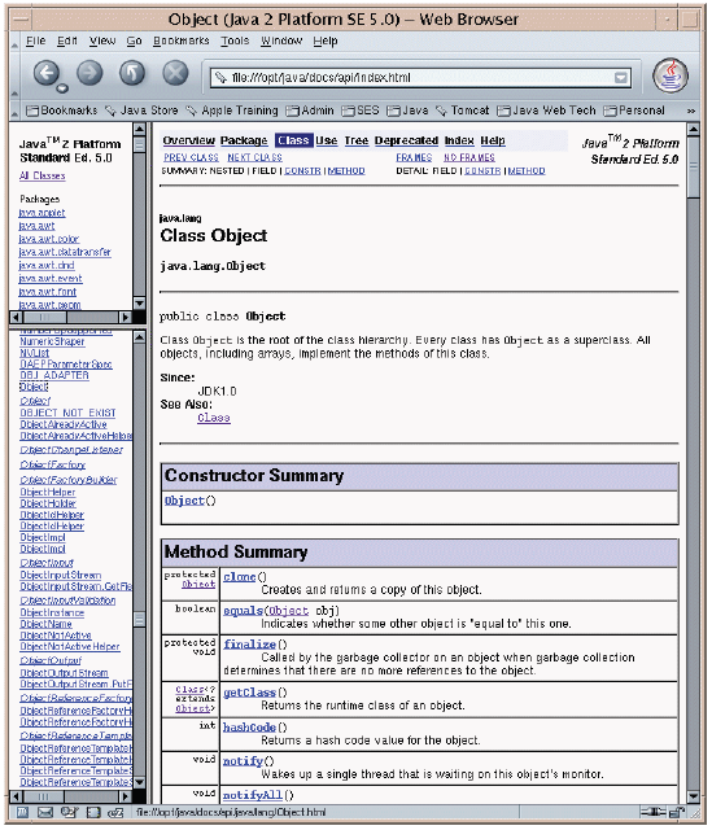
Empaquetamiento de programas



Repaso de términos

- **Clase:** una manera de definir nuevos tipos de objeto. Puede ser considerado como un esquema que modela un objeto que estemos describiendo.
- **Objeto:** una instancia actual de una clase. Un objeto es lo que obtenemos cada vez que instanciamos una clase usando la instrucción new. Un objeto también es conocido como instancia.
- **Atributo:** elemento de datos de un objeto. Almacena información de un objeto. Puede ser conocido como: miembro de datos, variable de instancia o campo de datos.
- **Método:** elemento funcional de un objeto. Es conocido también como: función o procedimiento.
- **Constructor:** pieza parecida a un método, utilizada para inicializar o construir un nuevo objeto. Los constructores tienen el mismo nombre que la clase.
- **Paquete:** un grupo de clases, de sub-paquetes, o ambos.

La API de Java



Ejercicio

Identificadores, keywords y tipos

Comentarios

Son 3, los estilos de comentarios, permitidos, dentro de un programa Java

```
// comentario en una línea

/* comentario en una
 * o más líneas
 */

/** comentario para documentación
 * que también puede escribirse
 * en una o más líneas
 */
```

Bloques y punto y comas

En Java, toda instrucción debe terminar con un punto y coma (;) sin importar si está distribuida en una o varias líneas.

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

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

Un bloque, a menudo llamado, instrucción compuesta, es un grupo de instrucciones limitadas por llaves (`{ }`)

Bloques y punto y comas

```
// un bloque de instrucciones
{
      x = y + 1;y=x+1;
}
```

```
// definición de una clase
public class MyDate {
      private int day;
      private int month;
      private int year;
}
```

```
// un bloque de instrucciones
// anidado en otro bloque
while (i < large) {
      a = a + i;
      // bloque anidado
      if (a == max) {
            b = b + a;
            a = 0;
      }
      i=i+1;
}
```

Identificadores y keywords

- Un identificador, es el nombre dado a una variable, una clase, o un método. Empiezan con una letra, underscore (_), o signo de dólar ($), aunque está permitido, estas dos últimas alternativas de nombre, son inusuales.
- Las palabras reservadas de Java, tienen un significado especial para el compilador, identifican un tipo de datos o constructores de objetos, entre otros.

Identificadores y keywords

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

Tipos básicos de datos

- Java presenta muchos tipos de datos integrados. Estos caen en 2 categorías: los tipo de clases y los tipos primitivos. Los tipos primitivos son simples valores, no son objetos. Los tipos de clases son usados para tipos de datos complejos, incluyendo los tipos o clases que declaramos.
- Tipos primitivos
  - Lógico: boolean, tiene como valores posibles: true y false.
  - Textual: char, caracteres sencillos: ‘a’, ‘\t’.
  - Integrales, correspondientes a números enteros
    - byte: 8 bits
    - short: 16 bits
    - int: 32 bits
    - long: 64 bits
  - Punto Flotante, correspondientes a números decimales
    - double: 32 bits
    - float: 64 bits

Tipos básicos de datos

```
public class Assign {
        public static void main (String args[]) {
                // declarar variables enteros
                int x, y;
                // declarar y asignar punto flotante
                float z = 3.414f;
                // declarar y asignar un double
                double w = 3.1415;
                // declarar y asignar un booleano
                boolean truth = true;
                // declarar una variable char
                char c;
                // asignar valor a una variable char
                c = 'A';
                // asignar valor a las variables int
                x = 6;
                y = 1000;
        }
}
```

Tipos de referencia

Hemos visto como existen 8 tipos de datos primitivos: `boolean`, `char`, `byte`, `short`, `int`, `long`, `double` y `float`. Los demás tipos corresponden a objetos. Las variables que se refieren a objetos son llamadas variables de referencia.

```
public class MyDate {
        private int day = 1;
        private int month = 1;
        private int year = 2000;
        public MyDate(int day, int month, int year) { ... }
        public String toString() { ... }
}

public class TestMyDate {
        public static void main(String args[]) {
                MyDate today = new MyDate(22, 7, 1964);
        }
}
```

Construcción e inicialización de objetos

Usando la instrucción `new Xyz()` podemos especificar un espacio de almacenamiento para un nuevo objeto.

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

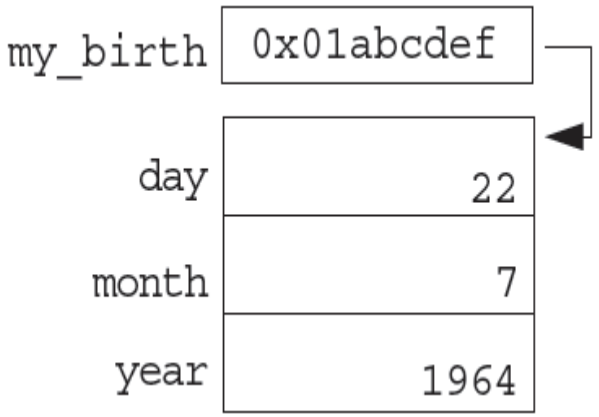
El uso de la palabra reservada `new` provoca lo siguiente:

Primero, el espacio para el nuevo objeto es reservado e inicializado a 0 o nulo. Esto impide que un objeto tenga valores al azar en él.
Segundo, cualquier inicialización explicita es realizada.
Tercero, el constructor es ejecutado.
Finalmente, el valor de retorno de la operación `new` es una referencia al nuevo objeto en la memoria. Esta referencia es almacenada en la variable de referencia.

Asignación de Variable

La asignación de la variable entonces se asigna a la variable de referencia

```
MyDate my_birth = new MyDate(22, 7, 1964);
```

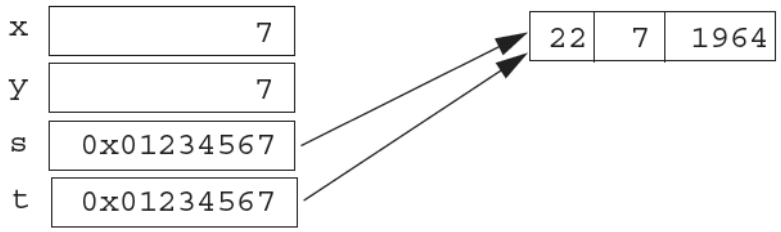


Asignación de Referencias

Una variable declarada con un tipo de clase, es referida como un tipo de referencia debido a que se refiere a un tipo no primitivo.

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
```

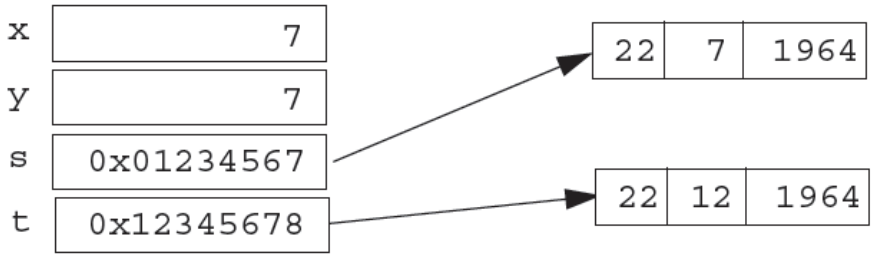
Cuatro variables son creadas, 2 primitivas de tipo `int`, y 2 de referencia del tipo `MyDate`.



Asignación de Referencias

Con la siguiente reasignación, se crea una nueva variable de referencia

```
t = new MyDate(22, 7, 1964);
```



La referencia *this*

```
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;
        }
        public MyDate addDays(int moreDays) {
                MyDate newDate = new MyDate(this);
                newDate.day = newDate.day + moreDays;
                return newDate;
        }
    }
```

Convenciones de codificación

Paquetes: los nombres deben ser en minúsculas

```
package shipping.objects
```

Clases: los nombres debe ser sustantivos, la primera letra en mayúscula.

```
class AccountBank
```

Métodos: deben ser verbos, con la primera letra en minúscula.

```
balanceAccount()
```

Variables: cualquier nombre que sea significativo, la primera letra en minúscula.

```
currentCustomer
```

Constantes: deben ser en mayúsculas, con palabras separadas por underscore (_)

```
HEAD_COUNT
```

Convenciones de codificación

Estructuras de control: utilizar llaves ({ }) alrededor de todas las instrucciones.

```
if ( condición ) {
  instrucción
}
```

Espaciado: colocar una sola instrucción por línea, y utilizar tabulación de 2 o 4 espacios para legibilidad.

Comentarios: use comentarios para explicar los segmentos de código que no sean obvios.

Ejercicio

Expresiones y flujo de control

- Las variables y su alcance
- Existen dos formas de describir variables: de tipo primitivo o de referencia. Se pueden declarar dentro de un método o fuera de el, pero dentro de la clase.
- Las variables locales, son aquellas definidas dentro de un método. Deben ser inicializadas explícitamente para poder ser utilizadas. Los parámetros de entrada de un método o de un constructor también son variables locales pero son inicializadas por el código que los invoca.
- Las variables definidas fuera de un método, son creadas cuando el objeto es construido usando la palabra reservada new. Existen 2 tipos posibles para estas variables. El primero, es la variable de clase, que es declarada por la palabra reservada static. El segundo, es la variable de instancia, son aquellas que se declaran sin la palabra reservada static.

Operadores

Variable	Value
byte	0
short	0
int	0
long	0L
float	0.0F
double	0.0D
char	'\u0000'
boolean	false
All reference types	null

Operadores

Operators
++ -- + - ~ ! (<*data_type*>)
* / %
+ -
<< >> >>>
< > <= >= instanceof
== !=
&
^
|
&&
||
<*boolean_expr*> ? <*expr1*> : <*expr2*>
= *= /= %= += -= <<= >>= >>>= &= ^= |=

Operadores lógicos

Los operadores lógicos y relacionales retornan un resultado `boolean`.

Los operadores de tipo `boolean` que son soportados por Java, se tienen: `!`, `&`, `^` y `|` para las operaciones algebraicas booleanas NOT, AND, XOR y OR respectivamente. Cada uno de estos operadores retornan un resultado `boolean`.

Los operadores `&&` y `||` son los operadores de corto circuito equivalentes a los operadores `&` y `|`.

```
MyDate d = reservation.getDepartureDate() {
        if (d != null) && (d.day > 31) {
                // alguna instrucción sobre d
        }
}
```

Concatenación de caracteres con +

El operador `+` permite la concatenación de objetos de tipo `String`, la clase que nos permite representar caracteres, a diferencia del tipo primitivo `char`, que se utiliza para representar un único carácter.

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

El resultado de la última línea es

```
Dr. Peter Seymour
```

Si alguno de los argumentos del operador `+` es un objeto `String`, entonces el otro argumento es convertido a un objeto `String`.

Casting (conversión)

El casting permite asignar un valor de un tipo a una variable de otro tipo. Si los dos tipos de datos son compatibles, la plataforma Java realiza una conversión automática. Por ejemplo, un valor `int` siempre puede ser asignado a una variable `long`.

Si en la conversión se puede perder información, se requiere que se confirme dicha asignación con un casting. Por ejemplo, reducir un valor `long` en una variable `int`:

```
long bigValue = 99L;
int squashed = bigValue;   // Error, casting necesario
int squashed = (int) bigValue;          // Correcto
```

Las variables pueden ser promovidas automáticamente a una forma más grande (por ejemplo, de `int` a `long`) cuando no hay riesgo de perdida de información.

Instrucciones de rama

Las instrucciones condicionales, permiten la ejecución selectiva de porciones del código, de acuerdo a algunas expresiones. Java soporta las instrucciones if y switch para ramas de 2 vías o múltiples vías, respectivamente

Instrucciones simples de `if`, `else`

La sintaxis básica de la instrucción `if` es:

```
if ( <expresion_booleana> )
       <instruccion_o_bloque>
```

Por ejemplo

```
if (x<10) {
       System.out.println("¿Hemos terminado?");
}
```

Instrucciones de rama

Instrucciones complejas de if, `else`

Si se requiere una clausula `else`, entonces se utiliza la instrucción `if-else`:

```
if ( <expresion_booleana> )
        <instruccion_o_bloque>
else
        <instruccion_o_bloque>
```

Por ejemplo

```
if (x<10) {
        System.out.println("¿Hemos terminado?");
} else {
        System.out.println("Seguimos trabajando...");
}
```

Instrucciones de rama

Instrucciones complejas de `if`, `else`

Si se requiere de una serie de validaciones condicionales, entonces se puede encadenar una secuencia de instrucciones `if-else`:

```
if ( <expresion_booleana> )
       <instruccion_o_bloque>
else if ( <expresion_booleana> )
       <instruccion_o_bloque>
else
       <instruccion_o_bloque>
```

Instrucciones de rama

La instrucción switch, su sintaxis es como sigue a continuación:

```
switch ( <expresion> ) {
        case <constante1>:
                <instruccion_o_bloque>*
                [break;]
        case <constante2>:
                <instruccion_o_bloque>*
                [break;]
        default:
                <instruccion_o_bloque>*
                [break;]
}
```

Se tiene la restricción de que `<expresion>` debe ser compatible con un tipo `int`.

Instrucciones de rama

En el `switch` La etiqueta opcional `default` especifica el segmento de código que será ejecutado cuando el valor de la variable o expresión no hace match con cualquiera de los valores `case`. Si no hay una instrucción de `break` como última instrucción en el segmento de código de un `case`, la ejecución continua en el segmento de código del siguiente `case` sin validar su expresión correspondiente.

Instrucciones de rama

```
switch ( carModel ) {
        case DELUXE:

addAirConditioning();
                addRadio();
                addWheels();
                addEngine();
                break;
        case STANDARD:
                addRadio();
                addWheels();
                addEngine();
                break;
        default:
                addWheels();
                addEngine();
}
```

```
switch ( carModel ) {
        case DELUXE:

addAirConditioning();
        case STANDARD:
                addRadio();
        default:
                addWheels();
                addEngine();
}
```

Instrucciones de ciclo

Las instrucciones de ciclo, permiten la ejecución de bloques de instrucciones, de forma repetitiva. Java soporta tres tipos de instrucciones de ciclo: `for`, `while` y `do`. Los ciclos de `for` y `while` verifican las condiciones antes de ejecutar el bloque de instrucciones, mientras que el `do` verifica las condiciones luego de ejecutar el bloque de instrucciones.

El ciclo `for`:

La sintaxis de la instrucción de cliclo `for` es:

```
for (<expresion_inicial>; <expresion_prueba>;
<expresion_alteracion>)
        <instruccion_o_bloque>
```

Instrucciones de ciclo

El ciclo `while`:

La sintáxis de la instrucción de cliclo `while` es:

```
while ( <expresion_prueba> )
       <instruccion_o_bloque>
```

Hay que asegurarse que las variables a utilizar estén inicializadas antes de la ejecución. Y así mismo, actualizar la variable de control de forma apropiada, para prevenir un ciclo infinito.

Instrucciones de ciclo

El ciclo `do / while`:

La sintáxis de la instrucción de cliclo `do / while` es:

```
do
        <instruccion_o_bloque>
while ( <expresion_prueba> )
```

Al igual que en la instrucción anterior, hay que asegurarse que las variables a utilizar estén inicializadas antes de la ejecución.

Instrucciones de ciclo

`Break` se utiliza para salir prematuramente de instrucciones `switch`, de ciclo: `for`, `while` o `do`.

```
do {
   instruccion;
   if ( condicion ) {
          break;
   }
   instruccion;
} while ( expresion_prueba );
```

Instrucciones de ciclo

`Continue` se utiliza para saltar e ir al final del cuerpo del bloque de instrucciones a repetir, y entonces retornar a la instrucción de ciclo.

```
do {
  instruccion;
  if ( condicion ) {
        continue;
  }
  instruccion;
} while ( expresion_prueba );
```

Ejercicio

Declaración de arrays

Los arrays son usados generalmente para agrupar objeto del mismo tipo. Podemos declarar arreglos de cualquier tipo, ya sea primitivos o de una clase:

```
char[] s;
Point[] p;    // donde Point es una clase
```

En Java, un array es un objeto, aún cuando este arreglo este compuesto de tipos de datos primitivos, y así como sucede con otras clases, la declaración no crea al objeto en si mismo. La declaración crea la referencia del objeto. La memoria a utilizar será asignada dinámicamente usando `new` o un inicializador de arreglo.

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

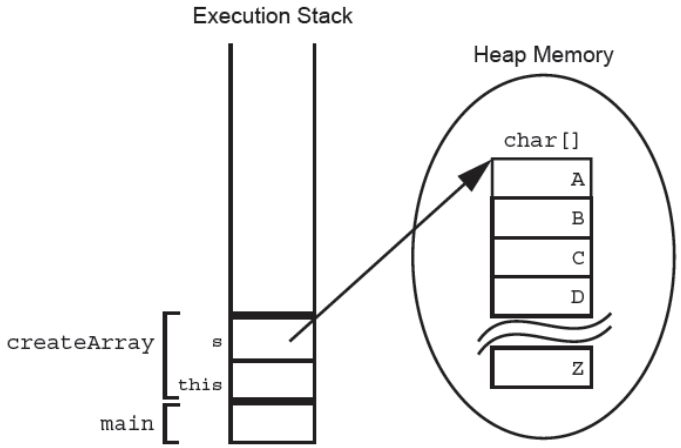
Creación arrays

Se pueden crear arreglo, como cualquier objeto, usando la palabra reservada `new`

```
s = new char[26];
```

Se debe rellenar el arreglo para que sea útil.

```
public char[] createArray() {
        char[] s;

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

        return s;
}
```



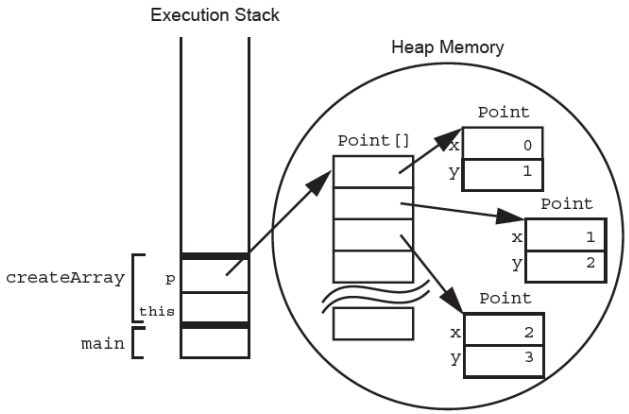
Creación de arrays de referencia

De igual forma se pueden crear arreglos de objetos:

```
p = new Point[10];
```

Su inicialización sería como sigue a continuación.

```
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;
}
```



Inicialización de arrays

Cuando se crea un arreglo, cada elemento es inicializado. En el caso del arreglo de char llamado s, cada valor es inicializado al carácter nulo (‘\u0000’). En el caso del arreglo de tipo Point llamado p, cada valor es inicializado a null, indicando que cada una de esas referencias aún no apuntan a un objeto Point.

Java permite un atajo en la creación de arreglos, con valores iniciales:

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

Este código es equivalente a:

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

Arrays multidimensionales

Debido a que se puede declarar arreglos que tengan cualquier tipo, se pueden crear arreglos de arreglos (y arreglos de arreglos de arreglos, y así sucesivamente).

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

El objeto creado por la primera llamada a new es un arreglo que contiene tres elementos. Cada elemento es una referencia `null` a un elemento del tipo `array` de `int` y debe ser inicializado cada elemento de forma separada, así cada elemento apunta a su arreglo.

Arrays multidimensionales

Se pueden crear arreglos de arreglos, no rectangulares:

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

Este tipo de inicialización es tedioso y los arreglos de arreglos, rectangulares, son de uso comun

```
int[][] twoDim = new int[3][5];
```

Límites de los arrays

En Java, el índice de todos los arreglos empiezan en 0. El número de elementos es un arreglo es almacenado como parte del objeto como tal, en un atributo llamado length.

```
public void printElements(int[] list) {
	for (int i = 0; i < list.length; i++) {
		System.out.println(list[i]);
	}
}
```

Límites de los arrays

Iterar sobre un arreglo es una tarea muy común. Para ello se puede hacer uso del ciclo `for` mejorado

```
public void printElements(int[] list) {
        for (int element : list) {
                System.out.println(element);
        }
}
```

La versión de este ciclo `for` puede ser leído como para cada elemento en la lista hacer tal instrucción.

Redimensionamiento de un array

Luego que un arreglo es creado, un arreglo no puede ser redimensionado. Sin embargo, se puede utilizar la misma variable de referencia para referirse a un nuevo arreglo.

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

En este caso, el primer arreglo efectivamente es perdido, al menos que otra referencia a él se mantenga en algún lado.

Copia de arrays

Java proporciona un método especial en la clase `System`, para copiar arreglos: `arraycopy()`

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

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

// copiar todos los elementos del arreglo myArray
// al arreglo hold, empezando por el índice 0
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

En este punto, el arreglo `hold` tiene los siguientes elementos:

```
1, 2, 3, 4, 5, 6, 4, 3, 2, 1
```

Copia de arrays

Java proporciona un método especial en la clase `System`, para copiar arreglos: `arraycopy()`

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

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

// copiar todos los elementos del arreglo myArray
// al arreglo hold, empezando por el índice 0
System.arraycopy(myArray, 0, hold, 0, myArray.length);
```

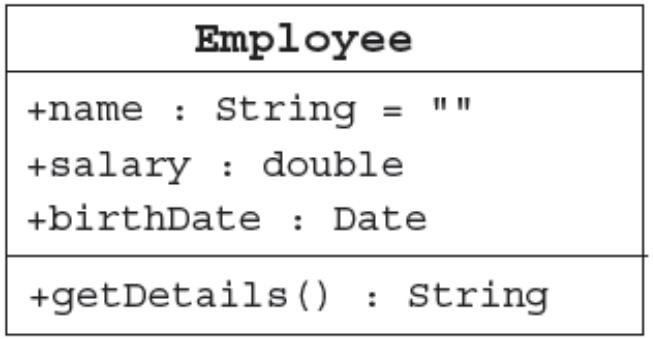
En este punto, el arreglo `hold` tiene los siguientes elementos:

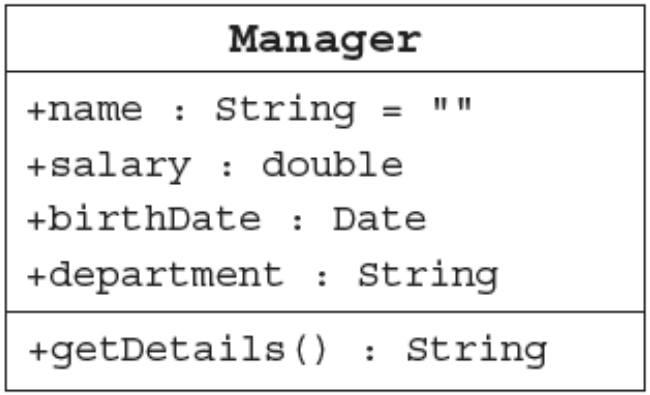
```
1, 2, 3, 4, 5, 6, 4, 3, 2, 1
```

Ejercicio

Diseño de clases

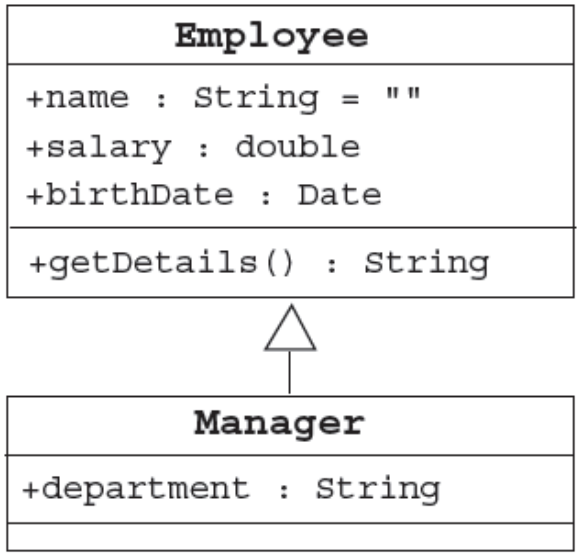
A menudo se crea un modelo de algo (por ejemplo, un empleado), luego se necesita una versión más especializada del modelo original. Por ejemplo, se desea modelar un gerente. Un gerente es un empleado, pero con características adicionales.



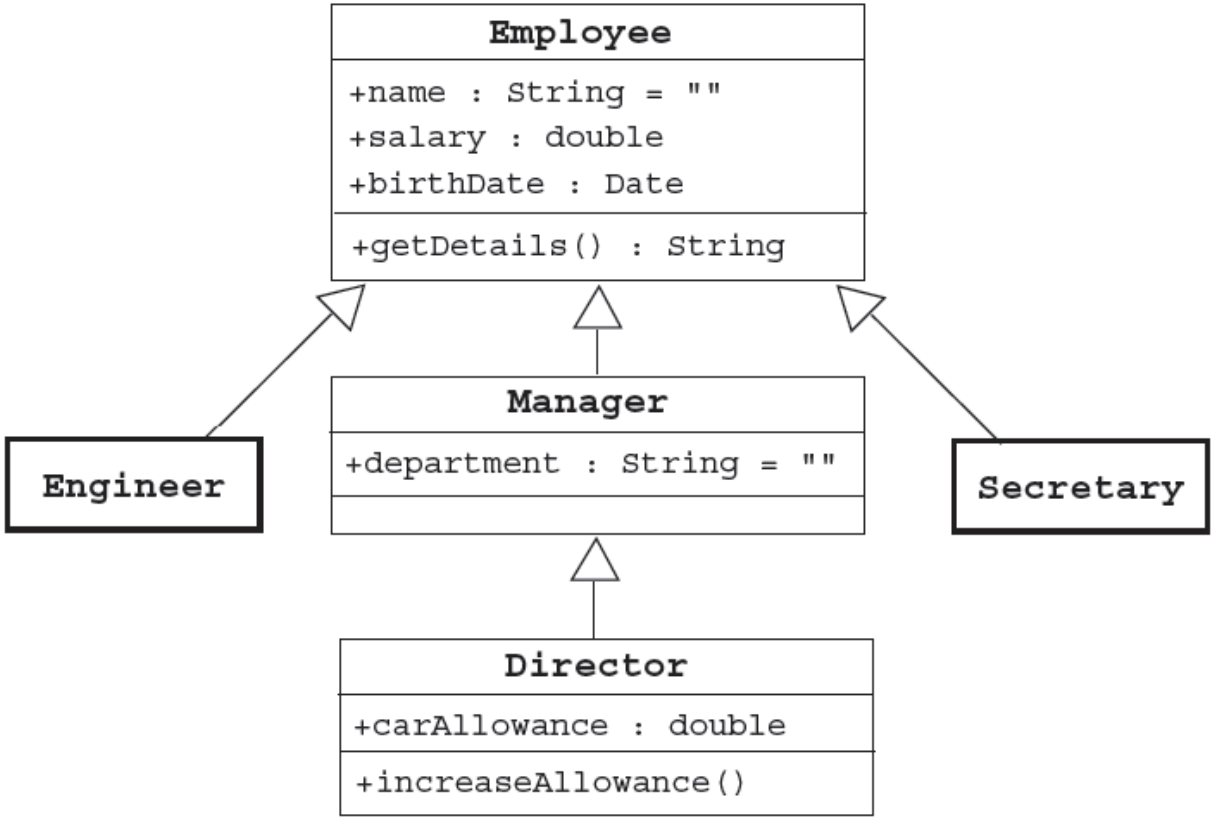


Subclases

Visto lo anterior, se tiene duplicación de data entre la clase `Manager` y la clase `Employee`. Adicionalmente, puede haber un número de métodos aplicables tanto a `Employee` como a `Manager`. Una subclase, es lo que nos permite crear una nueva clase a partir de un ya existente.



Subclases


Employee
+name : String = ""
+salary : double
+birthDate : Date
+getDetails() : String
Manager
+department : String = ""
Engineer
Secretary
Director
+carAllowance : double
+increaseAllowance()

Control de acceso

Modifier	Same Class	Same Package	Subclass	Universe
`private`	Yes			
default	Yes	Yes		
`protected`	Yes	Yes	Yes	
`public`	Yes	Yes	Yes	Yes

Sobrescritura de métodos

```
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;
        }
}
```

Sobrescritura de métodos

El nombre del método, el tipo de retorno y la lista de argumentos de un método hijo, deben ser idénticos al método en la clase padre para que, ese método pueda sobrescribir la versión del padre. Además, un método que sobrescribe no puede tener menor accesibilidad que el método original.

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

public class Child extends Parent() {
        private void doSomething () {}    // ilegal
}
```

Sobrescritura de métodos

```
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

Describir un `Manager` como un `Employee` no es solo una forma conveniente de describir la relación entre estas dos clases. `Manager` tiene todos los miembros de la clase padre `Employee`. Esto significa que cualquier operación que es legitima en `Employee` también lo es en `Manager`. Podría parecer poco realista crear un `Manager` y asignar deliberadamente su referencia a una variable de tipo `Employee`.

Un objeto solo tiene una forma (aquella que le viene dada cuando es instanciado). Sin embargo, una variable es polimórfica debido a que puede referirse a objetos de diferentes formas. Por lo que se puede:

```
Employee e = new Manager();

e.department = "Sales";    // intento ilegal de asignar
atributos
                           // de la clase Manager
```

Métodos virtuales

Dado este escenario

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

Si se ejecuta `e.getDetails()` y `m.getDetails()`, se invocan diferentes comportamientos. El objeto `Employee` ejecuta si versión del método, e igual sucede con el objeto `Manager`. En cambio, si se ejecutase

```
Employee e = new Manager();
e.getDetails();
```

Métodos virtuales

Se obtiene el comportamiento asociado con el objeto al cual la variable se refiere en tiempo de ejecución. El comportamiento no es determinado por el tipo de la variable en tiempo de compilación. Este es un aspecto del polimorfismo, y es una característica importante de los lenguajes orientados a objetos. Este comportamiento a menudo es llamado como Invocación de métodos virtuales.

Por lo que, en el ejemplo previo, la llamada `e.getDetails()` es ejecutada desde el tipo real del objeto, `Manager`.

Colecciones heterogéneas

Se pueden crear colecciones de objetos que tengan una clase común, es lo que se llaman colecciones homogéneas

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

Java proporciona la clase `Object`, por lo que se pueden hacer colecciones de todos los tipos de elementos debido a que todas las clases extienden de `Object`. Estas son las llamadas colecciones heterogéneas.

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

El operador instanceof

```
public class Employee extends Object
public class Manager extends Employee
public class Engineer extends Employee

public void doSomething(Employee e) {
       if ( e instanceof Manager ) {
              // Procesar como Manager
       } else if ( e instanceof Engineer ) {
              // Procesar como Engineer
       } else {
              // Procesar como cualquier tipo de
Employee
       }
}
```

Casting de objetos

En circunstancias donde se recibe la referencia a un clase padre, y se determina que el objeto, de hecho, es una subclase particular usado el operador `instanceof`, se puede acceder a las funcionalidades completas del objeto, convirtiendo la referencia mediante un casting.

```
public void doSomething(Employee e) {
        if ( e instanceof Manager ) {
                Manager m = (Manager) e;
                System.out.println("Este es el gerente de "
                                + m.getDepartment());
        }
        // resto de operaciones
}
```

Sobrecarga de métodos

En algunas circunstancias, se desea escribir muchos métodos en alguna clase que hagan el mismo trabajo básico con diferentes argumentos. Java permite reusar el nombre de un método. Se debe cumplir con las restricciones que permitan distinguir esos métodos.

Por ejemplo

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

Dos reglas aplican para métodos sobrecargados

La lista de argumentos debe diferir
Los tipos de retorno deben ser diferentes

Sobrecarga de constructores

```
public class Employee {
        private static final double BASE_SALARY = 15000.00;
        private String name;
        private double salary;
        private 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);
        }
}
```

La clase Object

La clase Object es la raíz de todas las clases en Java. Si una clase es declarada sin la clausula `extends`, entonces el compilador añade implícitamente el código `extends Object` a la declaración

```
public class Employee {
	// código de la clase
}
```

Es equivalente a

```
public class Employee extends Object {
	// código de la clase
}
```

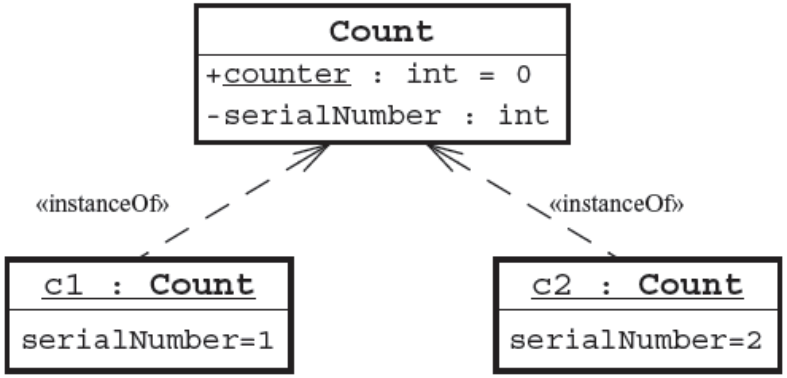
Esto permite sobrescribir muchos métodos heredados de la clase `Object`, como por ejemplo: `equals`, `toString`, etc.

Clases contenedoras

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

Ejercicio

Características de clases avanzadas




```
public class Count {
        private int serialNumber;
        public static int counter = 0;

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

```
public class OtherClass {
        public void incrementNumber() {
                Count.counter++;
        }
}
```

La palabra reservada static

```
public class Count2 {
        private int serialNumber;
        private static int counter = 0;

        public Count2() {
                counter++;
                serialNumber = counter;
        }
        public static int getTotalCount() {
                return counter;
        }
}

public class TestCounter {
        public static void main(String[] args) {
                System.out.println(Count2.getTotalCount());
        }
}
```

La palabra reservada static

Debido a que los métodos estáticos se pueden invocar sin instanciar la clase, no se tiene acceso al valor `this`.

```
public class Count3 {
	private int serialNumber;
	private static int counter = 0;

	public static int getTotalCount() {
		return this.serialNumber;		// ERROR
DEL COMPILADOR!!!
	}
}
```

No se pueden sobrescribir métodos estáticos.

La palabra reservada final

Clases finales, son aquellas clases de las cuales no se puede extender, es decir, no se puede heredar.

```
public final class NoExtendsClass {
          ...
}
```

Métodos finales, estos son métodos que no pueden ser sobrescritos, para garantizar que su implementación no pueda ser cambiada.

```
public final void finalPrint() {
         ...
}
```

Variables finales, es la manera en que Java permite la declaración de constantes. Su valor no puede ser cambiado.

```
public final int TOP_SALARY = 10000;
```

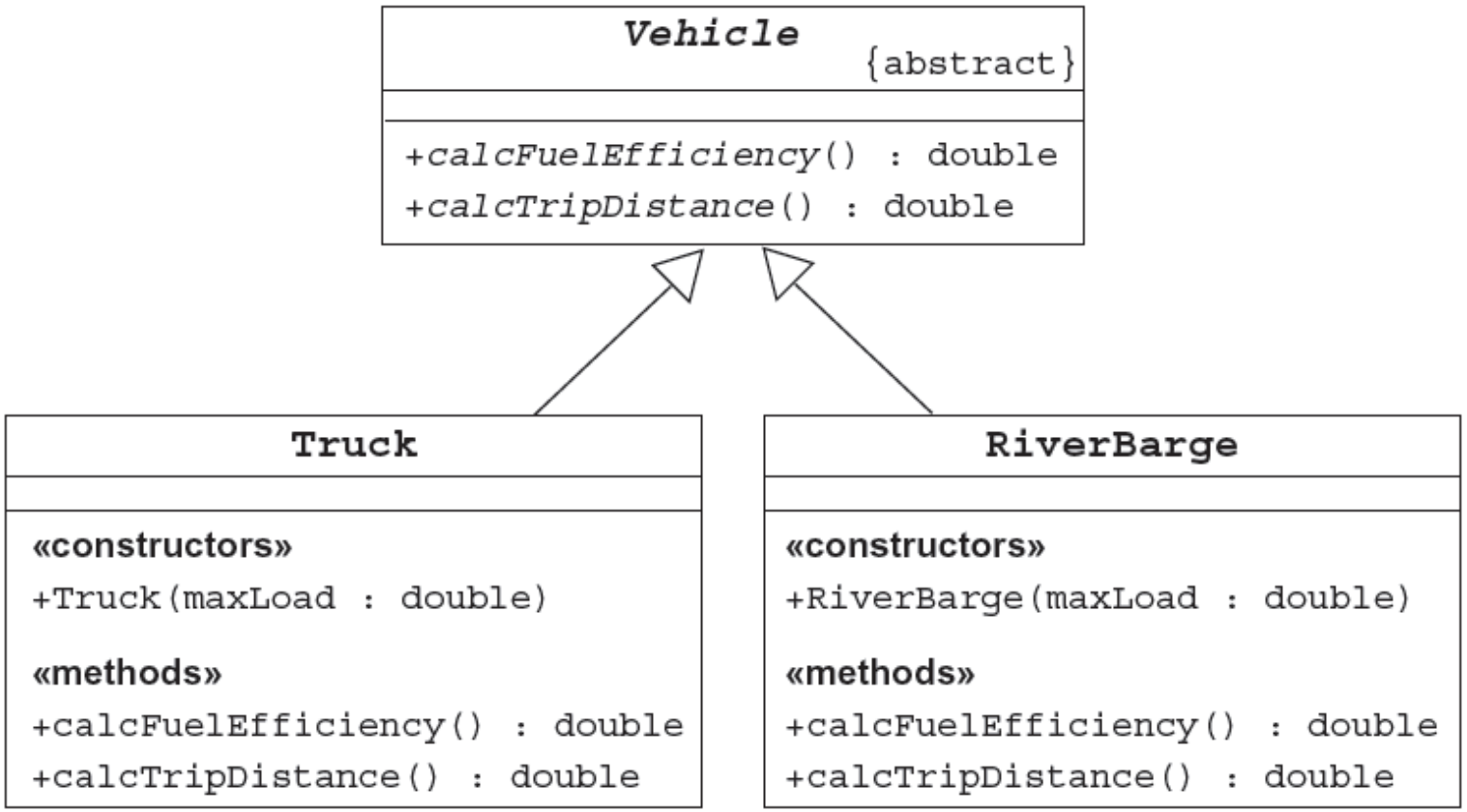
Tipos enumerados

```
public class JuegoCarta {
        public static final int ESPADAS = 0;
        public static final int BASTOS = 1;
        public static final int OROS = 2;
        public static final int COPAS = 3;

        private int pinta;
        private int rango;

        public JuegoCarta(int pinta, int rango) {
                this.pinta = pinta;
                this.rango = rango;
        }
        public int getPinta() { return pinta; }
}
```

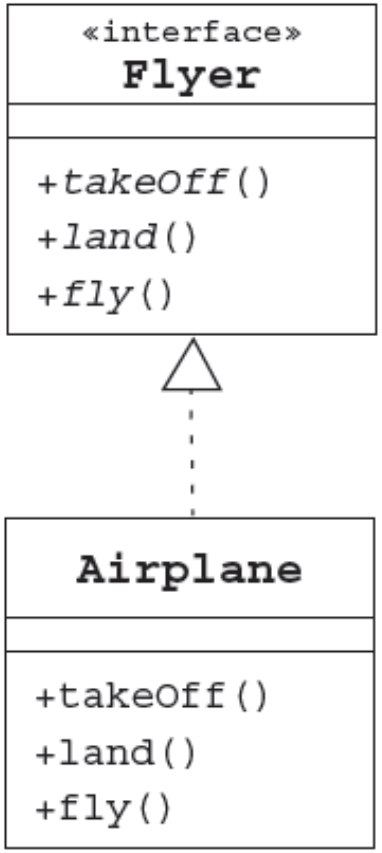
El nuevo tipo enumerado

```
public enum Carta {
        ESPADAS,
        BASTOS,
        OROS,
        COPAS
}

public class JuegoCarta {
        private Carta pinta;
        private int rango;

        public JuegoCarta(Carta pinta, int rango) {
                this.pinta = pinta;
                this.rango = rango;
        }
        public Carta getPinta() { return pinta; }
        ...
}
```

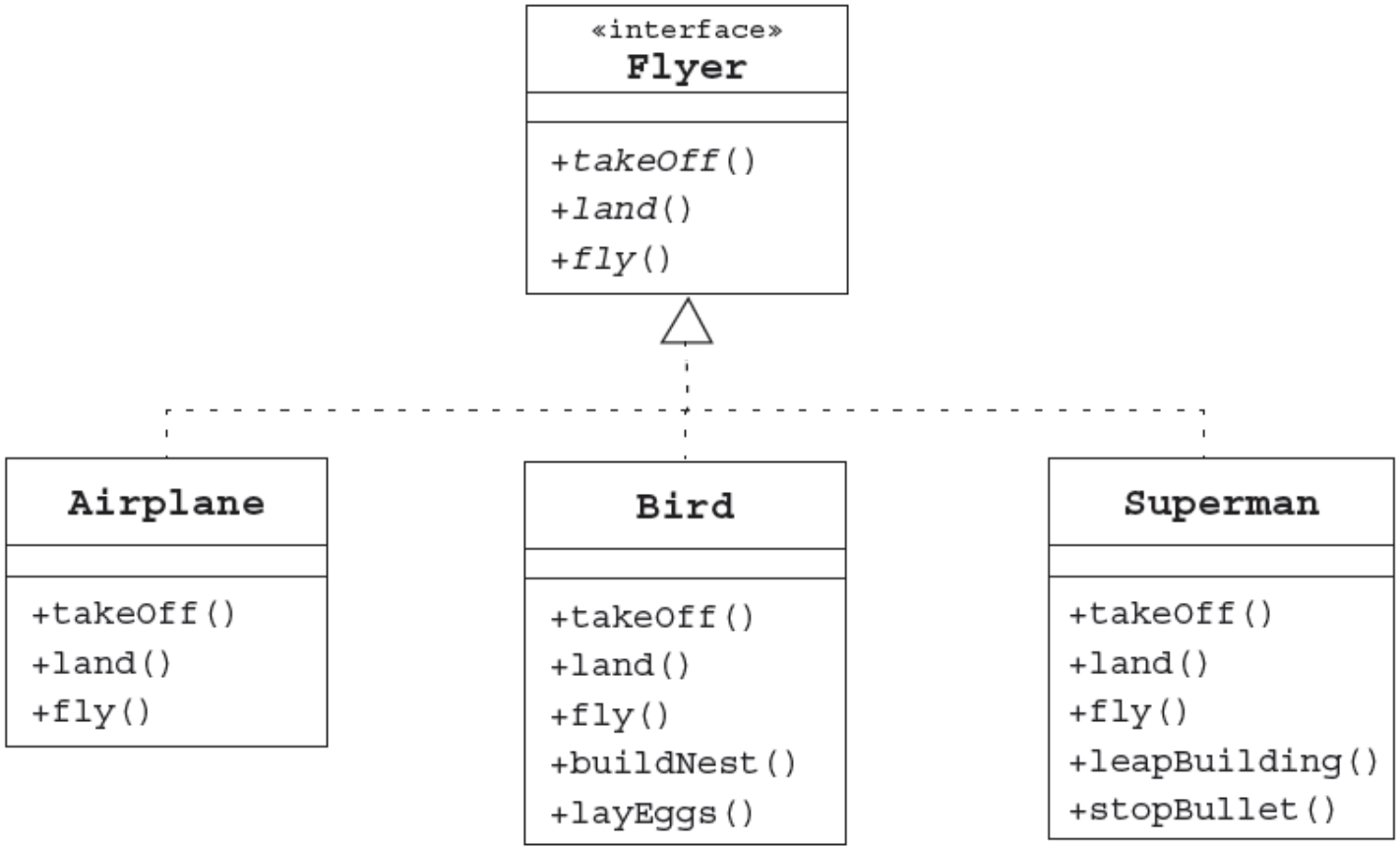
Clases abstractas



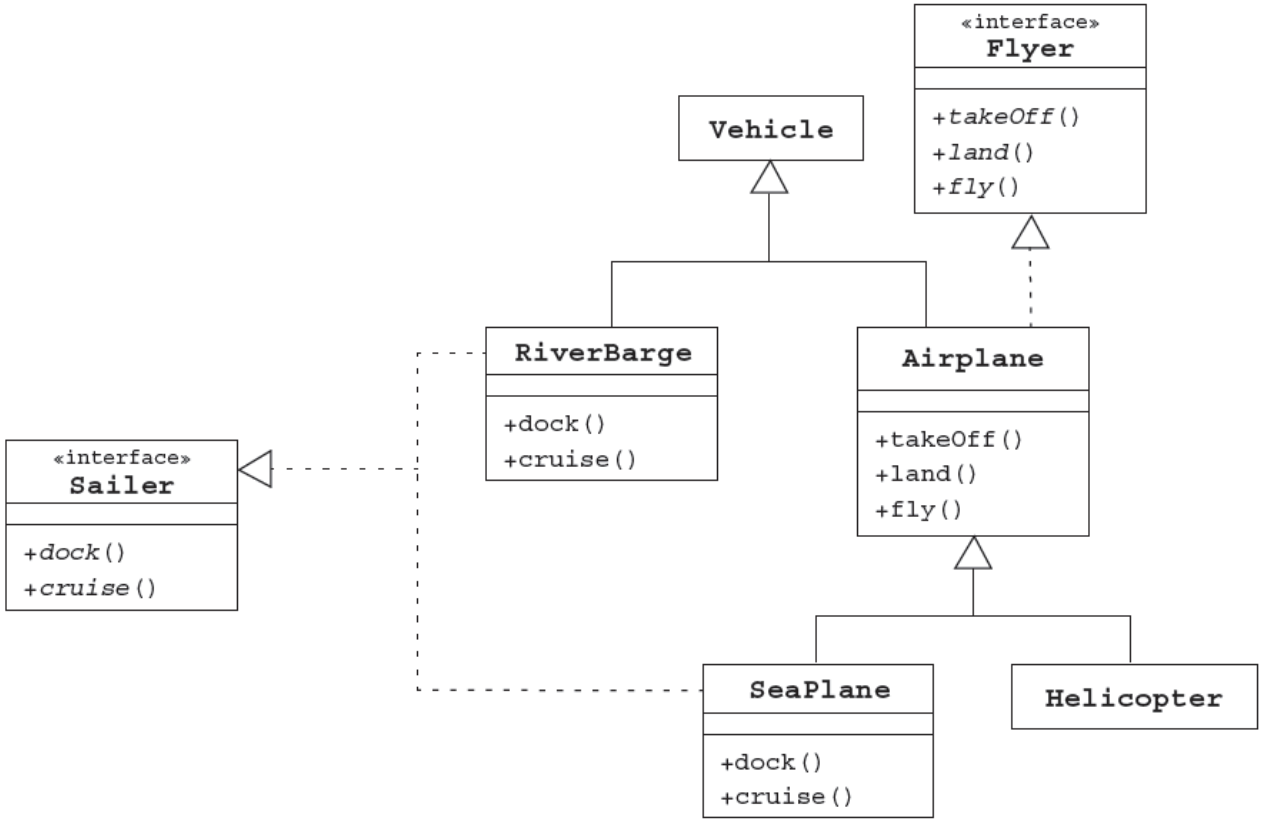
Interfaces


«interface»
Flyer
+takeOff()
+land()
+fly()
Airplane
+takeOff()
+land()
+fly()

Interfaces


«interface»
Flyer
+takeOff()
+land()
+fly()
Airplane
+takeOff()
+land()
+fly()
Bird
+takeOff()
+land()
+fly()
+buildNest()
+layEggs()
Superman
+takeOff()
+land()
+fly()
+leapBuilding()
+stopBullet()

Implementando varias interfaces



Ejercicio

Excepciones y aserciones

Las excepciones son mecanismos usados por los lenguajes de programación para describir que hacer cuando algo inesperado ocurre. Generalmente, un error.

Las aserciones son una manera de probar ciertas suposiciones sobre la lógica de un programa. Estas características, forman parte del proceso de desarrollo, y por lo tanto pueden ser removidas en el código ejecutable del programa.

Las excepciones se clasifican en dos grandes categorías: las chequeadas y las no chequeadas.

Las excepciones chequeadas, son aquellos errores que se esperan que puedan suceder durante la ejecución de un programa.

Las excepciones no chequeadas surgen de condiciones que pueden representar bugs, o situaciones que se consideran difíciles de ocurrir.

Excepciones

Cuando una excepción ocurre en un programa, el método que encuentra el error, puede manejar la excepción, o lanzar la excepción al método que lo invocó en primer lugar.

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

La instrucción try-catch

Java proporciona un mecanismo para capturar excepciones y así poder manejar dichas situaciones donde puedan surgir estas.

```
public class AddArguments2 {
        public static void main(String args[]) {
                try {
                        int sum = 0;
                        for(int i = 0; i < args.lenght; i++) {
                                sum += Integer.parseInt(args[i]);
                        }
                        System.out.println("Sum = " + sum);
                } catch (NumberFormatException nfe) {
                        System.out.println("Uno de los argumentos"
                        +" de la línea de comando, no es un
entero");
                }
        }
}
```

Múltiples cláusulas catch

Se puede tener múltiples bloques catch luego de un try, cada uno para manejar un tipo diferente de excepción.

```
try {
        // código que puede lanzar uno o más excepciones
} catch (MiExcepcion e1) {
        // código a ejecutar cuando ocurre MiExcepcion
} catch (MiOtraExcepcion e2) {
        // código a ejecutar cuando ocurre MiOtraExcepcion
} catch (Exception e3) {
        // código a ejecutar cuando ocurre cualquier
excepción
}
```

El orden de las clausulas importan, ya que una excepción lanzada por el código dentro del `try` será manejada por el primer `catch` que se encuentre.

La instrucción finally

Esta clausula define un bloque de código que siempre se ejecutará sin importar si la excepción ha ocurrido o ha sido manejada

```
try {
        iniciarProceso();
} catch (Exception e) {
        reportarProblema();
} finally {
        terminarProceso();
}
```

La instrucción throws

Luego de la palabra reservada throws se puede listar la o las excepciones que este método puede lanzar a método que lo invocó inicialmente. Finalmente, un desarrollador puede crear sus propias excepciones, implementando clases que extiendan de `Exception`

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

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

Puede ser lanzada así

```
throw new ServerTimedOutException("Sin conexión", 80);
```

Aserciones

¿Qué sucede si x es negativo? La suposición asumida es invalidada. Para verificar esto, se usan las aserciones.

```
if (x > 0) {
        // hacer esto
} else {
        assert( x == 0);
        // hacer aquello
}
```

Con la adición de la instrucción de aserción, se refuerza la validación de la suposición esperada. Si un cambio es realizado sobre el programa y modifica las condiciones iniciales que, provoca la invalidación de la suposición, al usar el `assert` se puede verificar esta situación.

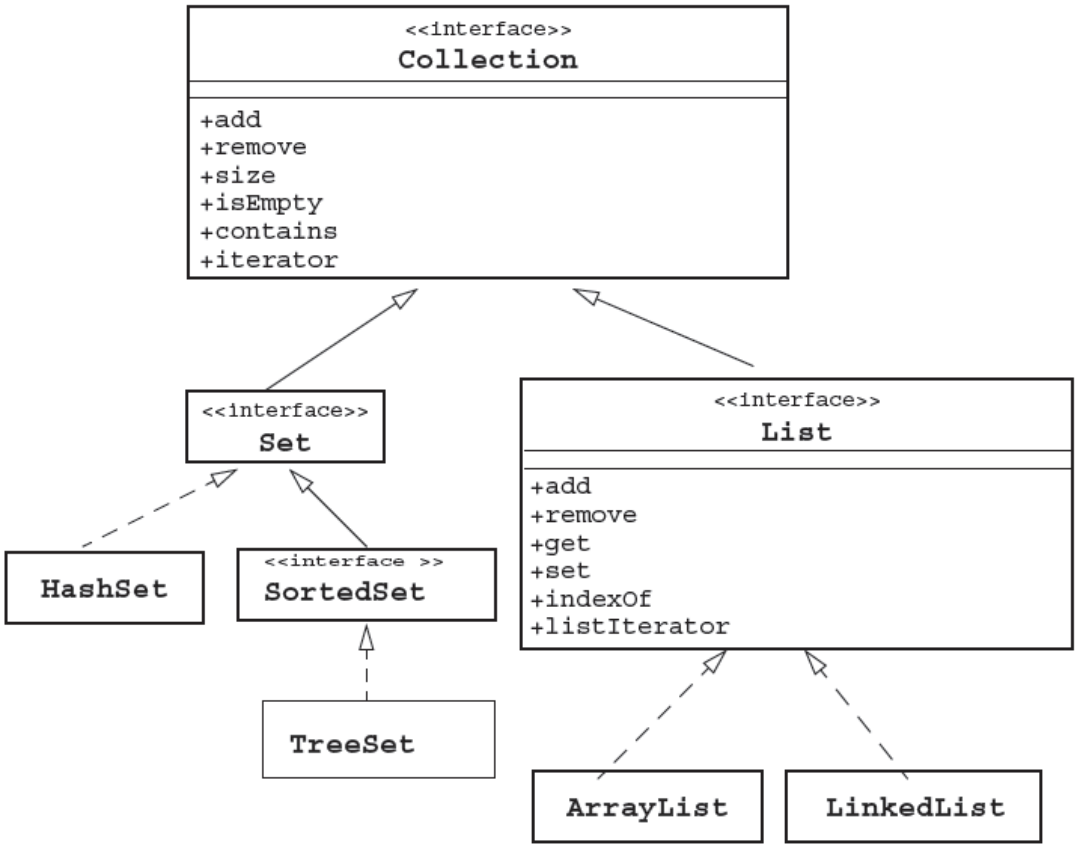
Ejercicio

Colecciones y genéricos

Una colección es un objeto sencillo que maneja un grupo de objetos. Java proporciona una librería para tales objetos: `Collections`, la cual contienes interfaces que agrupan objetos tales como:

- `Collection` – un grupo de objetos conocidos como elementos; las implementaciones determinan si se tiene un ordenamiento específico o si los duplicados se permiten.
- `Set` – una colección no ordenada, y los duplicados no están permitidos.
- `List` – una colección ordenada, y los duplicados están permitidos.

Colecciones y genéricos



Implementaciones de colecciones

	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

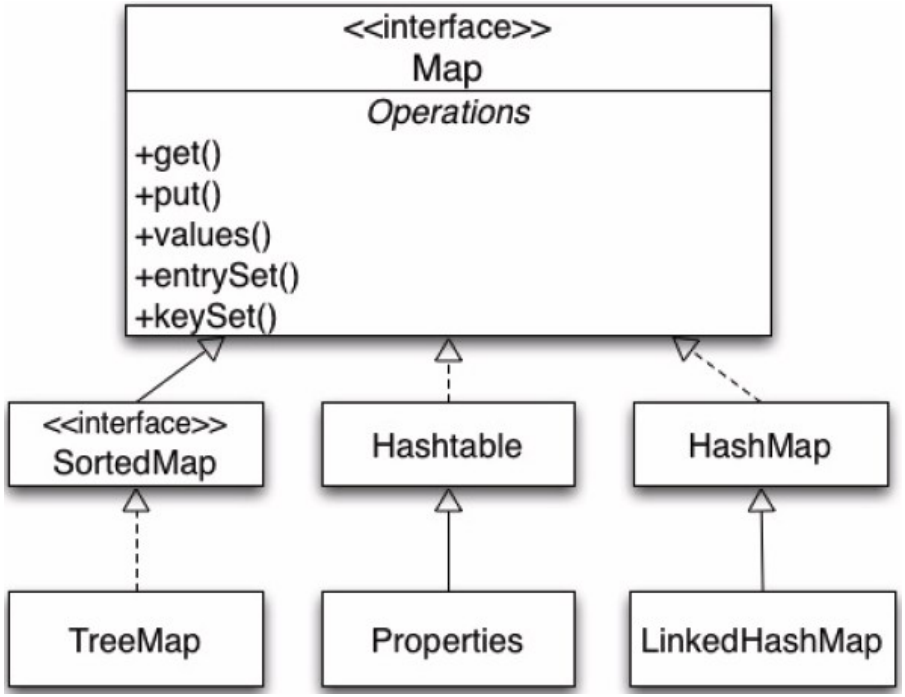
Ejemplo de Set

```
import java.util.*;
public class SetExample {
       public static void main(Strig args[]) {
              Set set = new HashSet();
              set.add("uno");
              set.add("segundo");
              set.add("3ero");
              set.add(new Integer(4));
              set.add(new Float(5.0F));
              set.add("segundo");                          //
duplicado, no agregado
              set.add(new Integer(4));              //
duplicado, no agregado
              System.out.println(set);
       }
}
```

Ejemplo de List

```
import java.util.*;
public class ListExample {
	public static void main(Strig args[]) {
		List list = new ArrayList();
		list.add("uno");
		list.add("segundo");
		list.add("3ero");
		list.add(new Integer(4));
		list.add(new Float(5.0F));
		list.add("segundo");			//
duplicado, es agregado
		list.add(new Integer(4));  // duplicado, es
agregado
		System.out.println(set);
	}
}
```

La interfaz Map



Ejemplo de Map

```
import java.util.*;
public class MapExample {
        public static void main(Strig args[]) {
                Map map = new HashMap();
                map.put("uno", "1ero");
                map.put("segundo", new Integer(2));
                map.put("tercero", "3ero");
                // sobrescritura de la asignación previa
                map.put("tercero", "III");
                // retorno del conjunto de claves
                Set set1 = map.keySet();
                // retorna la colección de valores
                Collection collection = map.values();
                // retorna el conjunto de correspondencia entre
                // claves y valores
                Set set2 = map.entrySet();
                System.out.println(set1 + collection + set2);
        }
}
```

Ordenamiento de colecciones

```
class Student implements Comparable {
        String firstName, lastName;
        int studentID = 0;
        double GPA = 0.0;
        public Student(String firstName, String lastName,
                                int studentID, double GPA) {
                if (firstName == null || lastName == null ||
studentID == 0
                                || GPA == 0.0) {
                                throw new IllegalArgumentException();
                }
                this.firstName = firstName;
                this.lastName = lastName;
                this.studentID = studentID;
                this.GPA = GPA;
        } // continua
```

Ordenamiento de colecciones

```
        // continua...
        public String firstName() { return firstName; }
        public String lastName() { return lastName; }
        public int studentID() { return studentID; }
        public GPA() { return GPA; }

        // implementación del método compareTo
        public int compareTo(Object o) {
                double f = GPA - ((Student)o).GPA();
                if (f == 0.0) return 0; // 0 significa iguales
                else if (f < 0.0) return -1; // negativo
significa menor que
                else return 1; // positivo significa mayor qu

        }
}
```

Ordenamiento de colecciones

```
public static void main(String[] args) {
	TreeSet studentSet = new TreeSet();
		studentSet.add(new Student("Miguel", "Perez",
101, 4.0);
	studentSet.add(new Student("Juan", "Diaz", 102, 2.8);
	studentSet.add(new Student("Jaime", "Garcia", 103,
3.6);
	studentSet.add(new Student("Maria", "Lopez", 104,
2.3);
	Object[] studentArray = studentSet.toArray();
	for(Object obj : studentArray) {
		Student s = (Student) obj;
		System.out.println("Nombre = %s %s ID = %d GPA
= %.1f\n",
		s.firstName(), s.lastName(), s.studentID(),
s.GPA());
	}
}
```

Genéricos

Las colecciones usan el tipo `Object` para los datos a registrar. Es necesario un casting explicito a la clase especifica. No hay seguridad de tipo.

Aunque se tienen colecciones homogéneas, no se tiene un mecanismo para prevenir que otras tipos de objetos se puedan insertar en la colección.

La solución a este problema, está en utilizar la funcionalidad `generics`

Esto elimina la conversión explicita de tipos de datos a ser usados en la colección.

Por ejemplo:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(new Integer(42));
```

Ejemplo de un Set genérico

```
import java.util.*;

public class GenSetExample {
	public static void main(String args[]) {
		Set<String> set = new HashSet<String>();
		set.add("uno");
		set.add("segundo");
		set.add("3ero");
		// esta línea falla en compilación
		set.add(new Integer(4));
		// duplicado, no se agrega
		set.add("segundo");
		System.out.println(set);
	}
}
```

Iteradores

```
Iterator<Student> elements = list.iterator();
while(elements.hasNext()) {
   System.out.println(element.next());
}

for (Iterator<Student> elements = list.iterator();
   elements.hasNext(); ) {
   System.out.println(element.next());
}


for (Student s : list) {
   System.out.println(s);
}
```

Ejercicio

Fundamentos de E/S

Un `stream` es un flujo de datos que van desde un origen (`source`) a un destino (`sink`). Generalmente, un programa es un extremo de ese flujo, y algún otro nodo (por ejemplo, un archivo) es el otro extremo.

Los orígenes y destinos son llamados flujos de entrada (`input stream`) y flujos de salida (`output stream`). Se puede leer de un `input stream` pero no se puede escribir a él. De igual forma, se puede escribir en un `output stream`, pero no se puede leer de él.

Java soporta dos tipos de datos en los `streams`: bytes crudos o caracteres Unicode. Generalmente, el termino `stream` aplica a flujo de bytes y los terminos `reader` y `writer` refieren a flujos de caracteres.

Fundamentos de E/S

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

Flujos de bytes

Métodos de la clase `InputStream`

Los siguientes tres métodos proveen acceso a la data del flujo de entrada:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

Cuando se ha terminado de trabajar con un flujo, este se debe cerrar.

```
void close()
```

Para saber el número de bytes que están disponibles para leer desde el flujo.

```
int available()
```

El método para descartar un número especificado de bytes del flujo de datos:

```
long skip(long n)
```

Flujos de bytes

Métodos de la clase `OutputStream`

Los siguientes tres métodos proveen acceso a la data del flujo de entrada:

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

Cuando se ha terminado de trabajar con un flujo, este se debe cerrar.

```
void close()
```

Algunas veces un flujo de salida acumula escrituras antes de enviarlas, el siguiente método permite forzar el envío:

```
void flush()
```

Flujos de caracteres

Métodos de la clase `Reader`

Los siguientes tres métodos proveen acceso a la data del flujo de entrada:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

Cuando se ha terminado de trabajar con un flujo, este se debe cerrar.

```
void close()
```

Para verificar si hay caracteres disponibles para leer desde el flujo.

```
boolean ready()
```

El método para descartar un número especificado de caracteres del flujo de datos:

```
long skip(long n)
```

Flujos de caracteres

Métodos de la clase `Writer`

Los siguientes tres métodos proveen acceso a la data del flujo de entrada:

```
void write()
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

Cuando se ha terminado de trabajar con un flujo, este se debe cerrar.

```
void close()
```

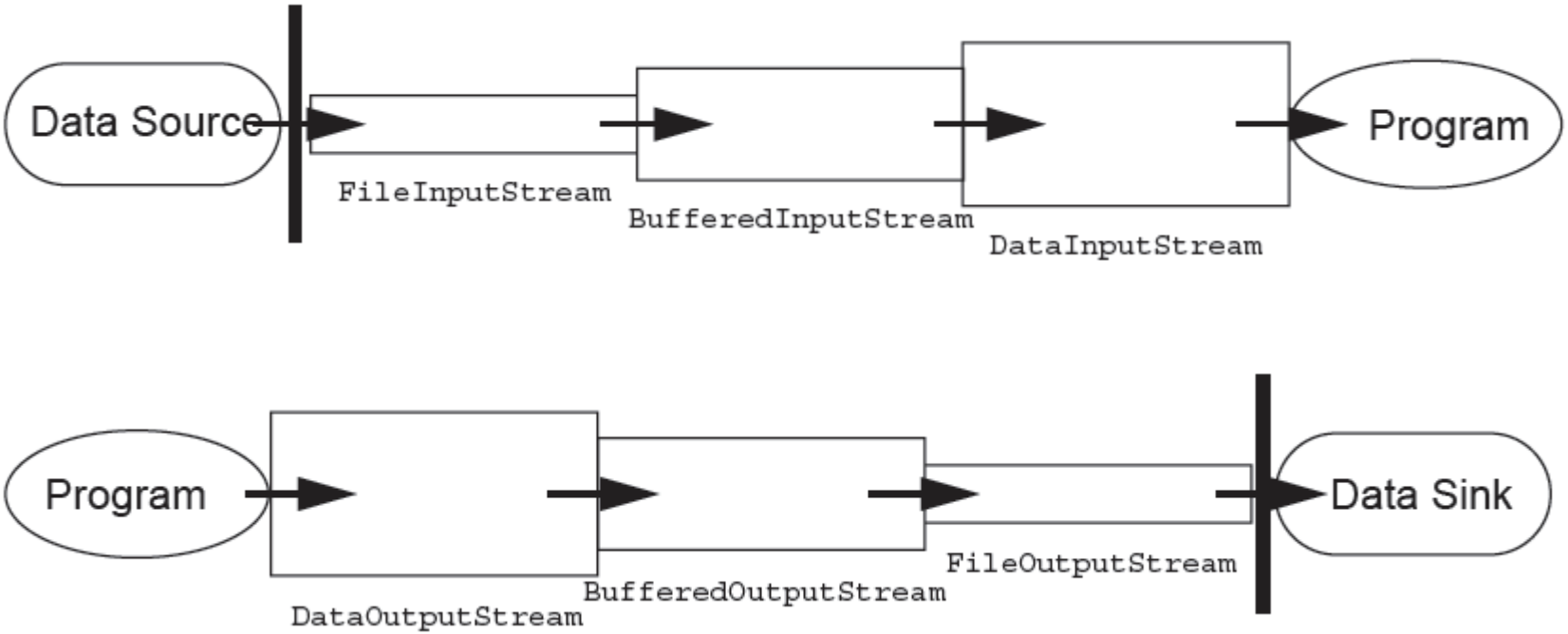
Algunas veces un flujo de salida acumula escrituras antes de enviarlas, el siguiente método permite forzar el envío:

```
void flush()
```

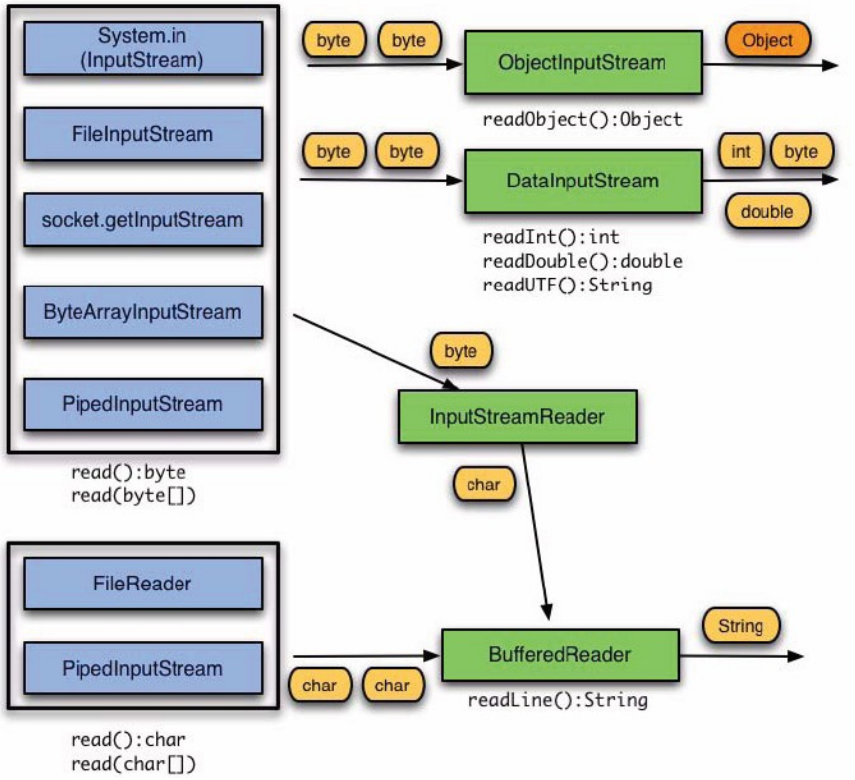
Flujos de nodos

Type	Character Streams	Byte Streams
File	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`
Memory: array	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
Memory: string	`StringReader` `StringWriter`	N/A
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`

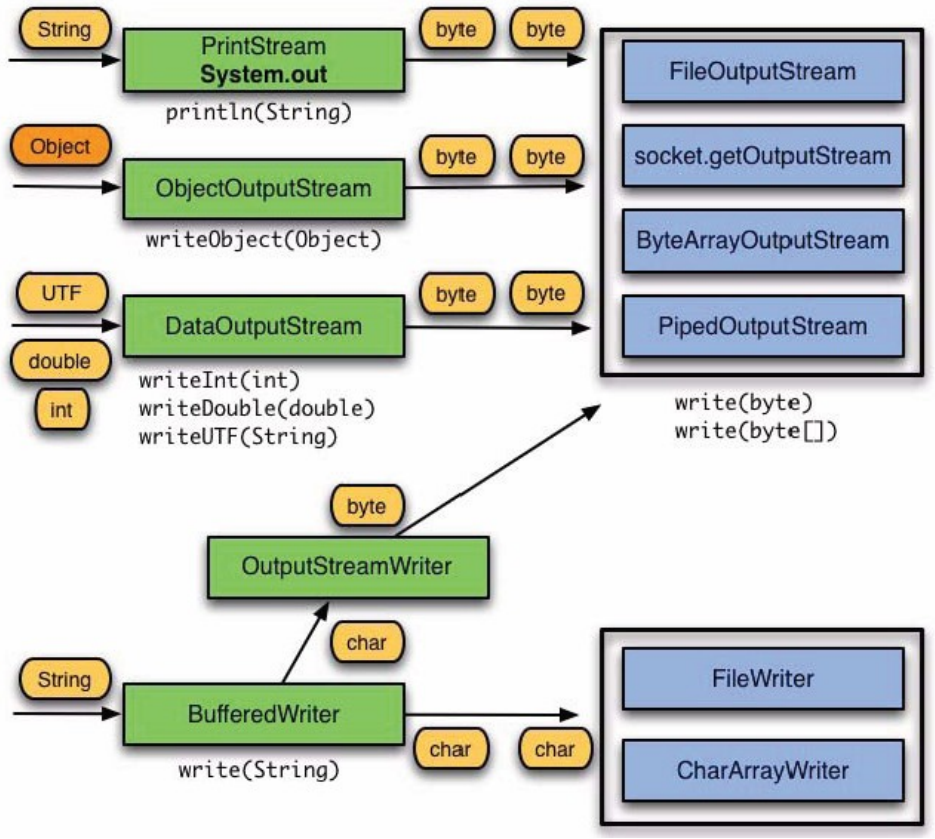
Encadenado de flujos E/S



Encadenado de flujos E/S



Encadenado de flujos E/S



Serialización

Es el mecanismo de salvar objetos como una secuencia de bytes y luego, cuando sea necesario, reconstruirlos, a partir de esa secuencia de bytes, en una copia del objeto.

Para que objetos de una clase especifica puedan ser serializables, la clase debe implementar la interfaz `java.io.Serializable`. Esta interfaz no tiene métodos y solo sirve como un marcador que indica que la clase que implementa la interfaz pueda ser considerada para serialización.

Cuando un objeto es serializado, solo los campos del objeto son preservados, los métodos y los constructores no forman parte del flujo serializado.

Consola – E/S

Muchas aplicaciones deben interactuar con el usuario. Alguna interacción es algunas veces completada con entrada y salida de texto hacia la consola (usando el teclado como entrada estándar, y la ventana del terminal como la salida estándar).

Java soporta la entrada y salida a consola con tres variables estáticas publicas en la clase `java.lang.System`:

`System.out` se refiere a la ventana del terminal de usuario.
`System.in` se refiere al teclado del usuario
`System.err` se refiere a la ventana del terminal de usuario.

La escritura a la salida estándar se puede lograr mediante los métodos sobrecargados `print` y `println` de la variable `System.out`.

Archivos – E/S

La clase `File` proporciones múltiples utilidades para el manejo de archivos y la obtención de información sobre ellos. En Java, un directorio es otro archivo. Se puede crear un objeto `File` que represente un directorio y utilizarlo para identificar otros archivos.

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

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

File myDir = new File("My Docs");
     myFile = new File(myDir, "myFile.txt");
```

La clase `File` define métodos, independientes de la plataforma, que permiten manipular un archivo mantenido por el sistema nativo de archivos. Sin embargo, no permite tener acceso al contenido de ellos.

Ejercicio

Swing

Swing es un conjunto de componentes mejorados que proporcionan componentes sustitutos para aquellos originales en AWT, y adicionalmente componentes avanzados. Estos componentes permiten crear interfaces gráficas de usuario con el tipo de funcionalidad esperada en las aplicaciones modernas.

Swing tiene una característica especial, permite que los programas adopten el look&feel de la máquina plataforma, o permite adoptar el look&feel propio de aplicaciones Java. Además, se tiene la posibilidad de crear un look&feel propio, o modificar alguno ya existente.

Los componentes Swing están diseñados siguiendo la arquitectura MVC. De forma teórica, estos elementos, deberían ser representados por tipos de clases diferentes. En la practica, Swing sigue una arquitectura separable, en este punto de vista, la vista y el controlador están unidos en un objeto composite sencillo.

Paquetes de Swing

`javax.swing` proporciona un conjunto de componentes ligeros tales como `JButton`, `JFrame`, `JCheckBox`, y muchos más

`javax.swing.border` proporciona clases e interfaces para dibujar bordes especializados.

`javax.swing.event` provee soporte para eventos disparados por componentes `Swing`.

`javax.swing.table` proporciona clases e interfaces para el manejo de tablas.

`javax.swing.tree` proporciona clases e interfaces para manejos de `JTree`.

`javax.swing.text` proporciona clases e interfaces que manejan componentes de textos editables y no editables.

`javax.swing.text.html` proporciona la clase `HTMLEditorKit` y las clases auxiliares para crear editores de texto `HTML`.

`javax.swing.text.html.parser` provee el interpretador `HTML` por defecto junto a sus clases auxiliares.

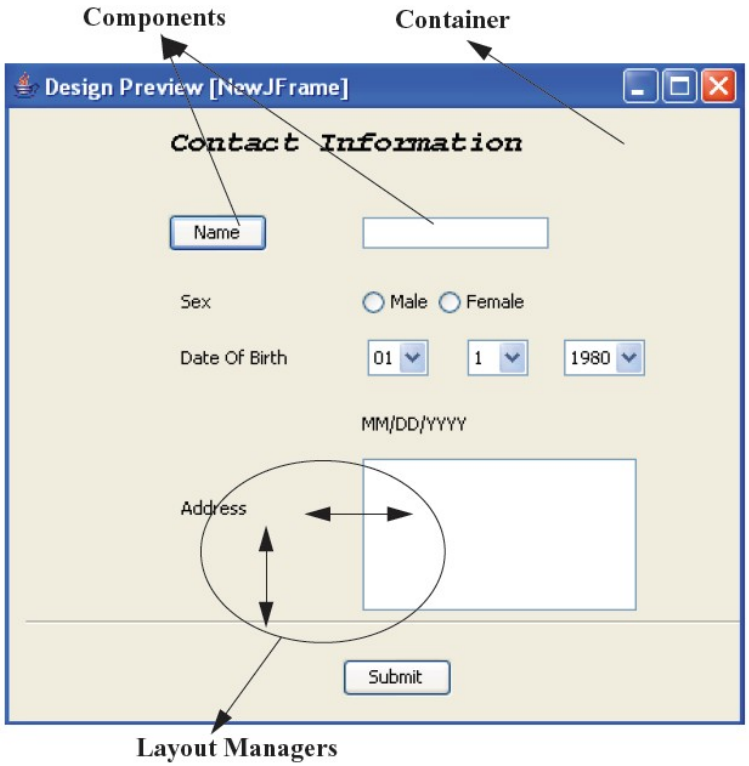
Composición de una GUI

Contenedores, están en el tope de la jerarquía de contención de una GUI. Todos los componentes son agregados a estos contenedores. `JFrame`, `JDialog`, `JWindow` y `JApplet` son esos elementos.

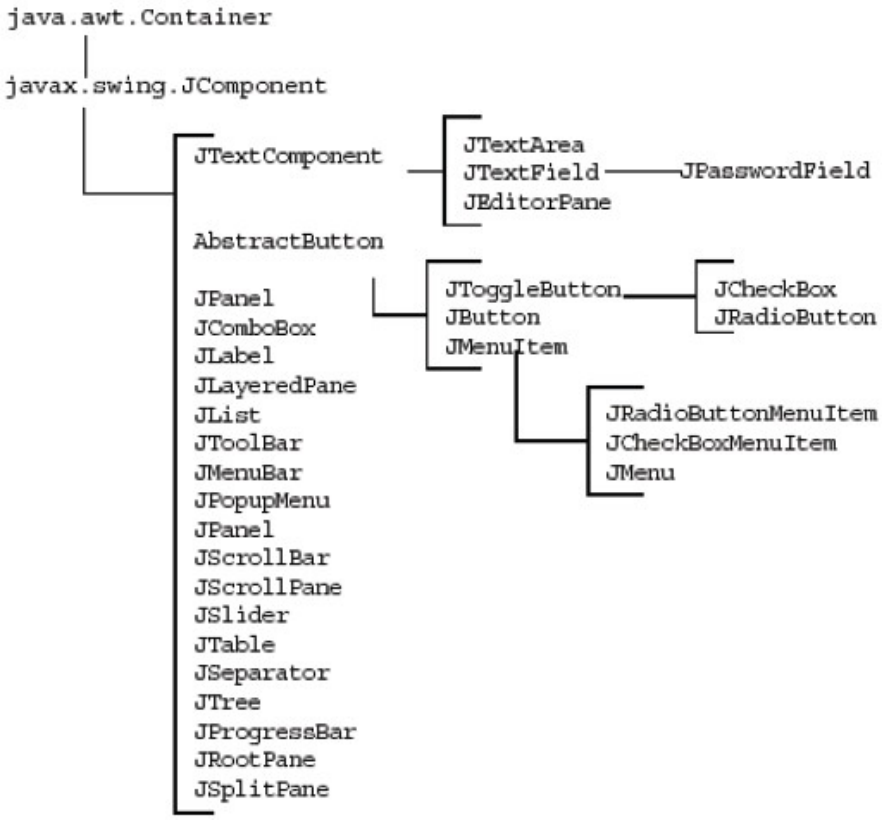
Componentes, todo elemento derivado de la clase `JComponent`, por ejemplo: `JComboBox`, `JAbstractButton` y `JTextComponent`.

Manejadores de distribución, son los responsables de la distribución de los componentes dentro de un contenedor. `BorderLayout`, `FlowLayout`, `GridLayout` son algunos de los ejemplos de estos elementos.

Composición de una GUI

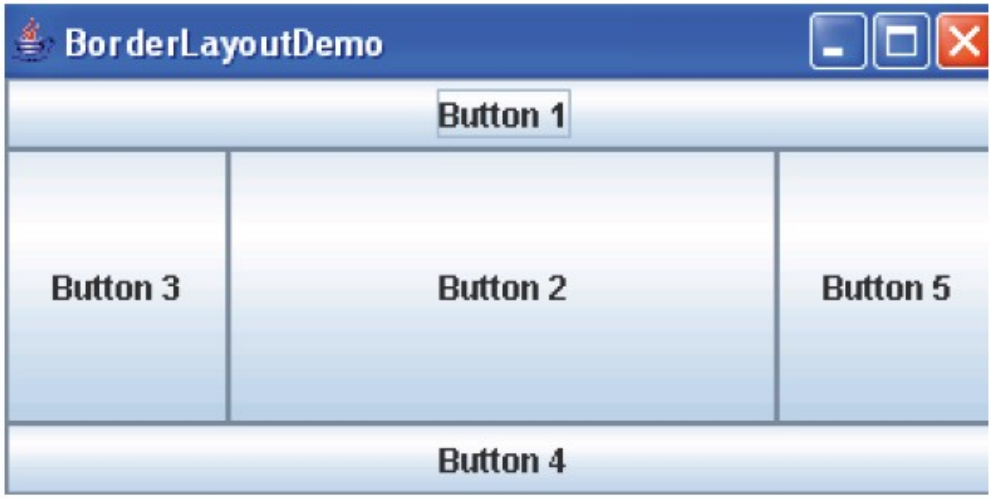


Jerarquía de componentes Swing



BorderLayout

`BorderLayout`, ordena los componentes en cinco diferentes regiones: `CENTER`, `NORTH`, `SOUTH`, `EAST` y `WEST`. Este manejador limita a uno el número de elementos por región.



BorderLayout

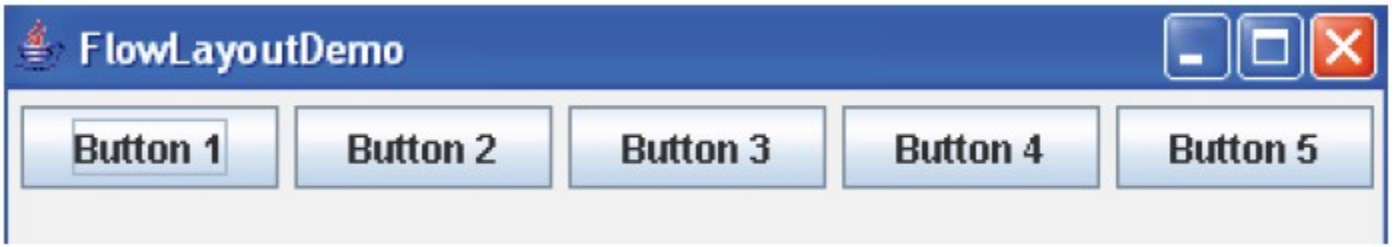
```
import java.awt.*;
import javax.swing.*;
public class BorderExample {
  private JFrame f;
  private JButton bn, bs, bw, be, bc;
  public BorderExample() {
       f = new JFrame("Border Layout");
       bn = new JButton("Button 1");
       bc = new JButton("Button 2");
       bw = new JButton("Button 3");
       bs = new JButton("Button 4");
       be = new JButton("Button 5");
  }
// continúa...
```

BorderLayout

```
  public void launchFrame() {
        f.add(bn, BorderLayout.NORTH);
        f.add(bc, BorderLayout.CENTER);
        f.add(bw, BorderLayout.WEST);
        f.add(bs, BorderLayout.SOUTH);
        f.add(be, BorderLayout.EAST);
        f.setSize(400, 200);
        f.setVisible(true);
        }
  public static void main(String[] args) {
        BorderExample guiWindow2 = new BorderExample();
        guiWindow2.launchFrame();
  }
}
```

FlowLayout

`FlowLayout`, ordena los componentes es una fila, por defecto de izquierda a derecha. La orientación puede cambiarse, así como el espaciamiento horizontal y vertical entre los componentes.



FlowLayout

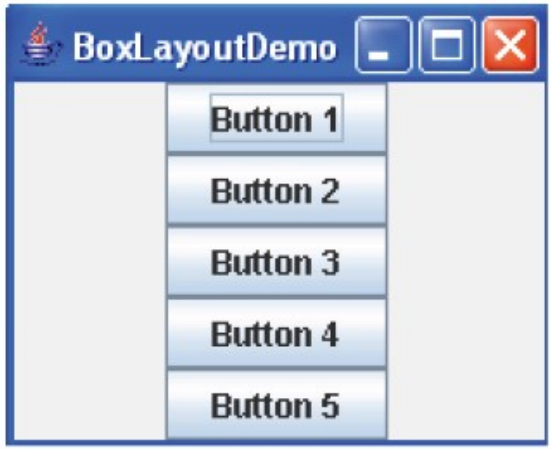
```
import java.awt.*;
import javax.swing.*;
public class FlowExample {
  private JFrame f;
  private JButton b1, b2, b3, b4, b5;
  public BorderExample() {
       f = new JFrame("GUI Layout");
       b1 = new JButton("Button 1");
       b2 = new JButton("Button 2");
       b3 = new JButton("Button 3");
       b4 = new JButton("Button 4");
       b5 = new JButton("Button 5");
  }
// continúa...
```

FlowLayout

```
  public void launchFrame() {
        f.setLayout(new FlowLayout());
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.pack();
        f.setVisible(true);
  }
  public static void main(String[] args) {
        FlowExample guiWindow = new FlowExample();
        guiWindow.launchFrame();
  }
}
```

BoxLayout

`BoxLayout`, ordena los componentes vertical u horizontalmente.

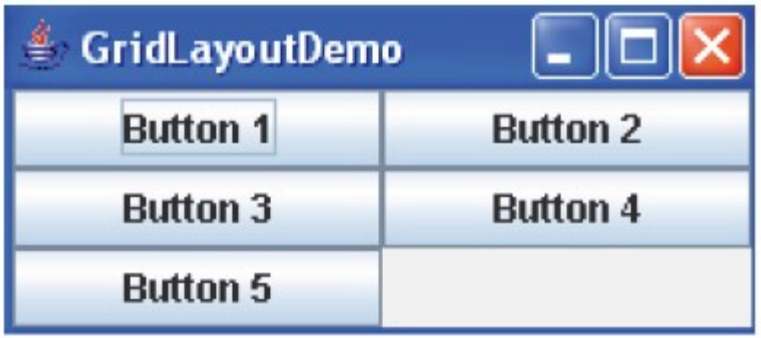


CardLayout

`CardLayout`, ordena los componentes como una pila de cartas. Cada carta acepta un componente sencillo a desplegar

GridLayout

`GridLayout`, ordena los componentes en filas y columnas. Cada componente ocupa la misma cantidad de espacio en el contenedor. Cuando se crea una grilla, se especifica el número de filas y columnas. En caso de no especificarse, se crea una fila y una columna.



GridLayout

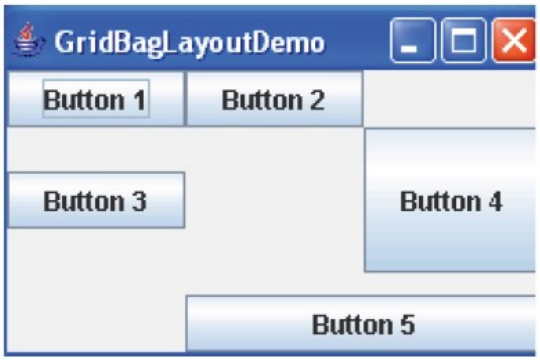
```
import java.awt.*;
import javax.swing.*;
public class GridExample {
  private JFrame f;
  private JButton b1, b2, b3, b4, b5;
  public BorderExample() {
       f = new JFrame("Grid Example");
       b1 = new JButton("Button 1");
       b2 = new JButton("Button 2");
       b3 = new JButton("Button 3");
       b4 = new JButton("Button 4");
       b5 = new JButton("Button 5");
  }
// continúa...
```

GridLayout

```
  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.pack();
        f.setVisible(true);
  }
  public static void main(String[] args) {
        GridExample grid = new GridExample();
        grid.launchFrame();
  }
}
```

GridBagLayout

`GridBagLayout`, ordena los componentes en filas y columnas similar al manejador anterior, pero proporciona una amplia variedad de opciones de flexibilidad para la redimensión y posicionamiento de componentes.



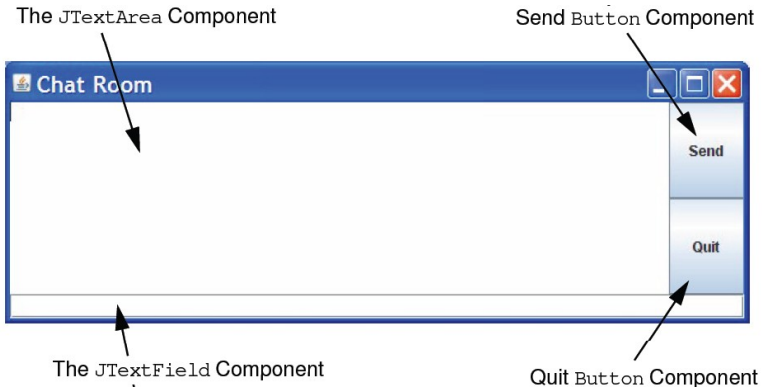
Construcción de una GUI

Puede ser creada usando cualquiera de las dos técnicas presentadas a continuación:

Construcción programática, esta técnica usa código para crear la GUI. Es útil para aprender construcción de GUI. Sin embargo, es un proceso muy laborioso para usar en ambientes de producción.

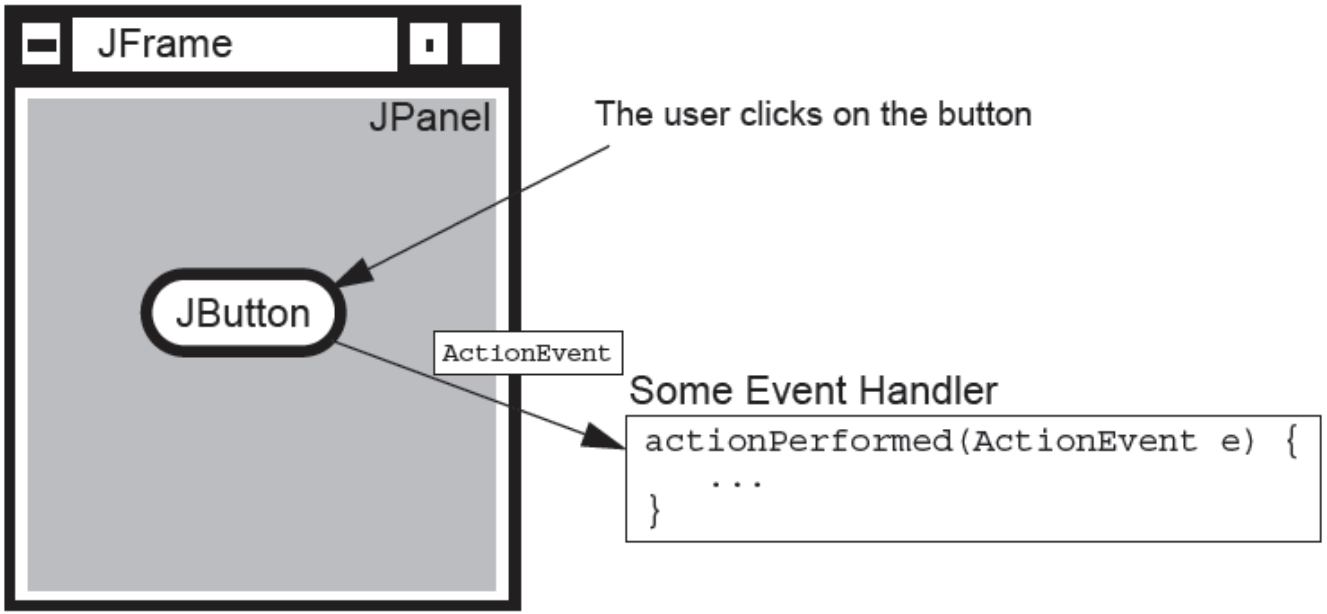
Construcción usando herramientas constructoras de GUI, esta técnica permite crear una GUI. El desarrollador usa un acercamiento visual para arrastrar-y-soltar contenedores y componentes en un área de trabajo. La herramienta permite el posicionamiento y la redimensión de los contenedores y componentes usando el ratón. Con cada paso, la herramienta automáticamente genera las clases Java para reproducir la GUI.

Ejercicio



1. Crear clase `ChatClient`
    1. Agregar 4 variables de instancia para los componentes: `JTextArea`, `JTextField` y 2 `JButton`.
    2. Agregar un constructor público que inicialice cada componente.
    3. Crear método `launchFrame` que construya la distribución de los componentes, usando `JFrame` con manejo `BorderLayout` para los componentes de texto (`WEST` y `SOUTH`) y un `JPanel` con manejo `GridLayout` para los botones (`CENTER`)
    4. Crear un método `main`, instanciar un objeto new `ChatClient` e invocar el método `launchFrame`

Eventos


JFrame
JPanel
JButton
The user clicks on the button
ActionEvent
Some Event Handler
actionPerformed(ActionEvent e) {
...
}

Fuente de un evento

La fuente de un evento es el generador del evento. Por ejemplo, el click del mouse en un componente `JButton` genera una instancia de la clase `ActionEvent` con el botón como fuente. La instancia de `ActionEvent` es un objeto que contiene información acerca del evento que ha tomado lugar.

Manejador de evento, es un método que recibe el objeto evento, lo descifra, y procesa la interacción del usuario.

Modelo de delegación, los eventos son enviados a los componentes desde los cuales el evento originó, pero depende de cada componente, propagar el evento a una a más clases registradas, llamadas `listeners`. Estas clases contienen los manejadores de eventos.

Cada evento tiene una interfaz para un `listener` correspondiente.

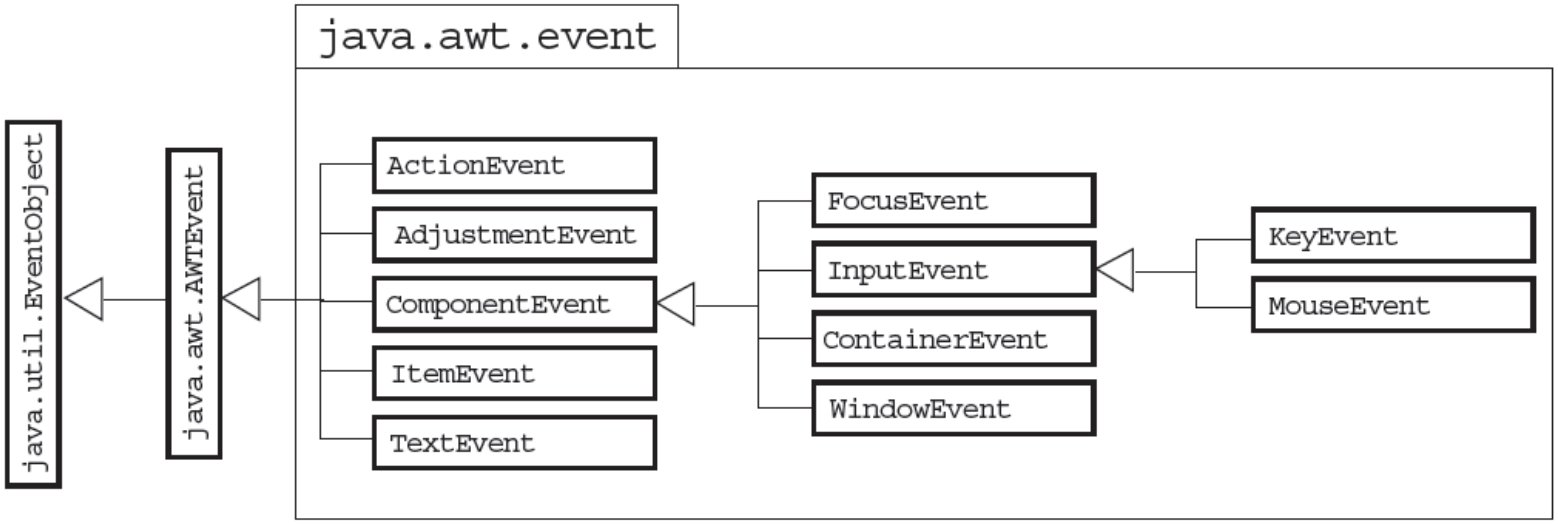
Ejemplo de un Listener

```
import javax.swing.*;
import java.awt.*:
public class TestButton {
        private JFrame f;
        private JButton b;
        public TestButton() {
                f = new JFrame("Test");
                b = new JButton("Press Me!");
                b.setActionCommand("ButtonPressed");
        }
        public void launchFrame() {
                b.addActionListener(new ButtonHandler());
                f.add(b, BorderLayout.CENTER);
                f.pack();
                f.setVisible(true);
        }
// continúa...
```

Ejemplo de un Listener

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

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

Categorías de evento



Categorías de evento

Category	Interface Name	Methods
Action	`ActionListener`	`actionPerformed(ActionEvent)`
Item	`ItemListener`	`itemStateChanged(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)`

Ejercicio

A partir de la clase pre-existente `ChatClient`, crear manejadores básicos de eventos.

Modificar clase `ChatClient`

1. Agregar un `ActionListener` al boton de `Send`. Este `listener` debe extraer el texto del `JTextField` y desplegarlo en el `JTextArea`. Usar una clase interna para ello, llamada `SendHandler`.
2. La clase interna implementa la interfaz `ActionListener`, especificamente el método `actionPerformed`.
3. Agregar el `ActionListener` creado, al `JtextField`
4. Agregar un `WindowListener` al `JFrame` de la GUI. Este `listener` debe cerrar el programa. Usar una clase interna para ello, llamada `CloseHandler`, implementar el método `windowClosing`, que invocará la instrucción `System.exit(0);`
5. Agregar un `ActionListener` al botón `Quit`. El `listener` debe cerrar el programa. Usar un clase interna anónima.
6. Compilar, ejecutar y verificar el programa

JMenuBar

Un `JMenuBar` es un componente para un menú horizontal. Se puede agregar aun objeto JFrame solamente.

```
f = new JFrame("JMenuBar");
mb = new JMenuBar();
f.setJMenuBar(mb);
```

JMenuBar

Un `JMenuBar` es un componente para un menú horizontal. Se puede agregar aun objeto JFrame solamente.

```
f = new JFrame("JMenuBar");
mb = new JMenuBar();
f.setJMenuBar(mb);
```

JMenuItem

Los componentes `JMenuItem` son los nodos desplegables de un menú. Ellos son agregados al menú para completarlo .

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

Ejercicio

A partir de la clase pre-existente `ChatClient`, mejorar la GUI agregando un menú

Modificar clase `ChatClient`. Agregar un menú de File. Este menu debe incluir un elemento de menú Quit que termine el programa. Para ello:

1. En el método launchFrame, agregar un JMenuBar, luego un JMenu para File, y un JMenuItem para la operación Quit.
2. Agregar un ActionListener, clase interna anónima para el JMenuItem.
3. Agregar el JMenuItem al JMenu.
4. Agregar el JMenu al JMenuBar.
5. Agregar el JMenuBar al frame

Compilar, ejecutar y verificar el programa

Threads (Hilos)

Un `thread` o contexto de ejecución, es considerado ser la encapsulación de un CPU virtual con su propio código de programa y data. La clase `java.lang.Thread` permite crear y controlar hilos.

```
public class ThreadTester {
	public static void main(String[] args) {
		HelloRunner r = new HelloRunner();
		Thread t = new Thread(r);
		t.start();
	}
}
// continúa...
```

Threads (Hilos)

```
public class HelloRunner implements Runnable {
        int i;
        public void run() {
                while (true) {
                        System.out.println("Hello " + i++);
                        if (i == 50) {
                                break;
                        }
                }
        }
}
```

Threads (Hilos)

Un ambiente de programación `multithreaded` (multihilos) permite crear múltiples hilos basados en la misma instancia `Runnable`. Esto se puede hacer así:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

En este caso, ambos hilos comparten la misma data y código.

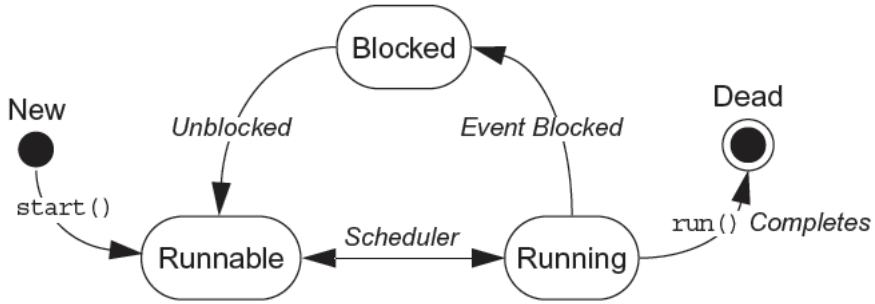
Para resumir, un hilo de ejecución es referido a través de una instancia de un objeto `Thread`. El hilo comienza su ejecución al inicio del método `run` de una instancia `Runnable` cargada. La data sobre la que el hilo de ejecución trabaja es tomada de la instancia especifica de `Runnable`, la cual es pasada al constructor de `Thread`.

Inicio de ejecución de un Thread

Un hilo recién creado no empieza su ejecución automáticamente. Se debe hacer la llamada al método `start`.

```
t.start();
```

La llamada a dicho método coloca el CPU virtual del hilo en estado `runnable` (ejecutable), significa que se ha convertido en viable para ser ejecutado por la JVM. Lo que no necesariamente significa que el hilo se ejecuta inmediatamente.



Fin de un hilo

```
public class Runner implements Runnable {
  private boolean timetoQuit = false;
  public void run() {
        while ( !timeToQuit ) {
          // hacer algo hasta que se diga que
          // se termina
        }
          // liberar recursos antes que el run() termine
  }
  public void stopRunning() {
        timeToQuit = true;
  }
}

public class ThreadController {
        private Runner r = new Runner(;
        private Thread t = new Thread(r);
        public void startThread() {   t.start();  }
        public stopThread() {   r.stopRunning();  }
}
```

Uso de synchronized

Proporciona un mecanismo que permite al programador controlar hilos de ejecución que comparten data. Un problema puede surgir cuando multiples threads están accesando data compartida. Es necesario un mecanismo que asegure que dicha data compartida está en un estado consistente antes que un hilo en particular la empiece a usar.

En Java, cada objeto tiene una bandera asociada a él. Se puede pensar que es una bandera de bloqueo. La palabra reservada synchronized permite la interacción con esa bandera y proporciona acceso exclusivo al código que afecta la data compartida de dicho objeto.

```
public class MyStack {
        ...
        public void push(char c) {
                synchronized (this) {
                        data[idx] = c:
                        idx++;
                }
        }
        ...
}
```

Uso de synchronized

También puede expresar el manejo de la bandera de bloqueo de la siguiente manera

```
public class MyStack {
        ...
        public synchronized void push(char c) {
                data[idx] = c:
                idx++;
        }
        ...
}
```

Una u otra manera funcionan y garantizan el uso exclusivo del recurso compartido. Usar `synchronized` en la declaración del método, tiene su desventaja si, el método es muy extenso, se corre el riesgo de bloquear el acceso a la data compartida, por más tiempo que el necesario.

Ejercicio

A partir de un proyecto ya existente, crear una clase `multithreaded`.

1. Crear la clase `PrintMe`, implementa la interfaz `Runnable`, el método `run` ejecuta la siguiente acción 10 veces: Imprimir el nombre del hilo actual (`Thread.currentThread().getName()`) y entonces duerme por 2 segundos.
2. Crear la clase de prueba `TestThreeThreads`, instanciar `PrintMe`, iniciar 3 hilos a partir de esa instancia. Especificar un nombre para cada hilo, sugerencia: `Larry`, `Moe` y `Curly`. Iniciar cada hilo.
3. Compilar, ejecutar y verificar el programa

Redes y sockets

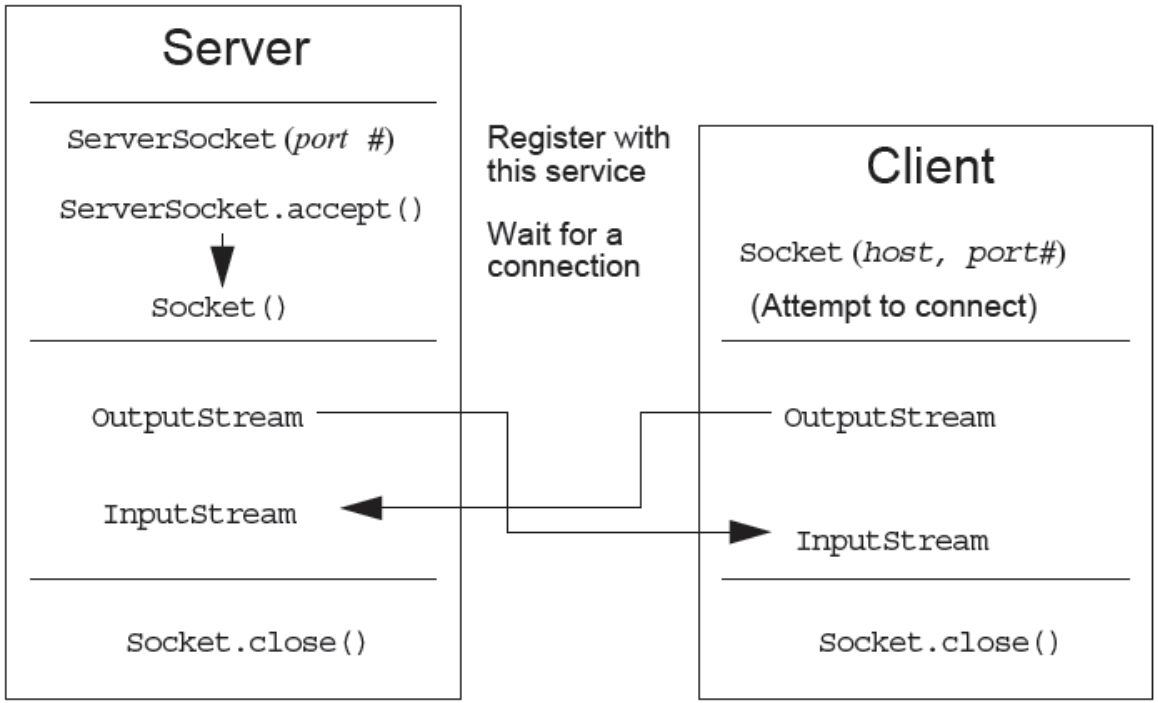
`Socket` es el nombre dado, en un modelo de programación particular, a los extremos de un enlace de comunicación entre dos procesos. Debido a la popularidad de este modelo de programación, el termino ha sido reusado en otros modelos de programación, incluyendo Java.

Cuando los procesos se comunican a través de la red, Java usa el modelo de flujo de datos. Un `socket` puede tener 2 flujos: uno de entrada de datos y otro de salida de datos. Un proceso envía data a otro proceso a través de la red escribiendo en el flujo de salida asociado con el `socket`. Un proceso lee data escrita por otro proceso mediante la lectura desde el flujo de entrada asociado al `socket`.

Para establecer la conexión, una maquina debe ejecutar un programa que espera dicha conexión. Y una segunda máquina debe intentar llegar a la primera.

Similar a un sistema telefónico, en la cual una parte debe hacer la llamada, mientras que la otra parte está esperando al lado del teléfono que la llamada se realice.

Redes y sockets



Servidor TCP/IP sencillo

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

public class SimpleServer {
  public static void main(String args[]) {
        ServerSocket s = null;

        // se registra el servicio en el puerto 5432
        try {
          s = new ServerSocket(5432);
        } catch (IOException e) {
          e.printStackTrace();
        }

        // ejecutar el bucle escuchar/aceptar para siempre
        while(true) {
          try {
// continúa...
```

Servidor TCP/IP sencillo

```
                    // espera aqui y escucha conexiones
                    Socket s1 = s.accept();
                    // Obtiene el flujo de salida asociado al socket
                    OutputStream s1out = s1.getOutputStream();
                    BufferedWriter bw = new BufferedWriter(new
OutputStreamWriter(s1out));
                    // se envia un string
                    bw.write("Hello Net World!");
                    // se cierra la conexión,
                    // pero no el socket del servidor
                    bw.close();
                    s1.close();
                    } catch (IOException e) {
                      e.printStackTrace();
                    }
                }
            }
}
```

Cliente TCP/IP sencillo

```
import java.net.*;
import java.io.*;
public class SimpleClient {
  public static void main(String args[]) {
        try {
          // se abre la conexión al servidor, en el puerto 5432
          Socket s1 = new Socket("127.0.0.1", 5432);
          // Obtiene el flujo de entrada asociado al socket
          InputStream is = s1.getInputStream();
          DataInputStream dis = new DataInputStream(is);
          // se lee la entrada de datos y se imprime
          System.out.println(dis.readUTF());
          // se cierra la conexión
          dis.close();
          s1.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
  }
}
```

Ejercicio

A partir de la clase pre-existente `ChatClient`, `ChatServer` y `Connection`, implementar la conexión del cliente del chat con el servidor del chat.

Modificar la clase `ChatClient`.
Agregar variables de instancia para los flujos de entrada (`BufferedReader`) y de salida (`PrintStream`), para la conexión del `Socket`.
Agregar método `doConnect` para iniciar la conexión mediante `socket` al servidor.

1. Obtener la dirección (`localhost`) y el puerto (5432)
2. Crear la conexión al servidor
3. Instanciar los flujos de entrada y salida de datos.
4. Lanzar el `thread` para el lector `RemoteReader` e iniciar el hilo.
5. Usar `catch` para capturar cualquier excepción.

Agregar la invocación `doConnect` al método `launchFrame`.
Modificar la clase anidada `SendHandler` para enviar el texto de los mensajes al flujo de salida del `socket`. Borrar el código que despliega el mensaje en el `text` area.
Crear la clase anidada `RemoteReader` que implementa la interfaz `Runnable`, el método `run` debe leer una línea a la vez, desde el flujo de entrada del `socket` en un bucle infinito
Compilar, ejecutar y verificar el programa