ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS TELEMÁTICOS

ANÁLISIS Y DISEÑO DE SOFTWARE

David Pérez Sanz ps10david@hotmail.com

Primera Edición: Septiembre de 2015

"No intentes refugiarte en lo más profundo de tu alma,
ni dejes tus sueños olvidados en un cajón.
Mantén tu mirada sedienta de explorar nuevos mundos
y nunca dándose por vencida. La vida es un abrir y cerrar de ojos"

Este libro ha sido elaborado con material de **Juan Antonio de la Puente Alfaro**, **Jose Antonio Mañas Argemi** y **Carlos Ángel Iglesias Fernandez** pertenecientes al equipo docente de la asignatura *Análisis y diseño de software*, asignada al Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid.

Capítulo 1

Introducción

1.1. Clases y objetos

Clases

Está compuesta por una serie de datos(atributos) y de procedimientos que tienen algo en común.

```
public class Circulo{
  private int radio;

  public Circulo(int radio){
    this.radio = radio;
  }
  public int getRadio(){
    return this.radio;
  }
  public int area(){
    return Math.PI * Math.pow(radio, 2);
  }
}
```

Si se trata de una **clase ejecutable**, esta dispondrá de un método con la cabecera:

```
public static void main(String[] args){...}
```

Objetos

Un **objeto** es un caso particular de una clase.

```
Circulo c1 = new Circulo(3.5);
```

En este caso la **referencia** al objeto es `c1`. Para llamar a los componentes(atributos o métodos) de un objeto:

```
c1.area();
```

Al comparar objetos de clases con `==`, comparamos referencias por ello:

```
Circulo c1=new Circulo(1.0);
Circulo c2=c1;
c1==c2         //true
```

Sin embargo

```
Circulo c1=new Circulo(1.0);
Circulo c2=new Circulo(1.0);
c1==c2         //false
```

1.2. Tipos y expresiones simples

Variables y Constantes

La declaración de una **variable** tiene la forma:

```
int radio;
```

Para generar una **constante**, únicamente añadimos `final`:

```
final int radio=3;
```

Podremos calificar como `final` una variable cuyo valor no queremos que se pueda modificar, un atributo que no dejamos modificar, un método que no queremos que se pueda sobrescribir(en una clase derivada) ó una clase que queramos que no se pueda extender.

Tipos primitivos

- **Tipos enteros:** `byte, short, int, long`
- **Tipos reales:** `float, double`
- **Tipos booleano:** `false, true`
- **Tipos carácter:** `char` (Los caracteres están ordenados: `A < B` ; `a < b`)

Tipos enumerados

Creamos un enumerado:

```
public enum ColoresSemaforo{
 ROJO, AMBAR, VERDE;
}
```

Se utiliza igual que un tipo primitivo:

```
...
 ColoresSemaforo i; → Declaración
 i = ColoresSemaforo.ROJO; → Inicialización ó asignación de valor
 if(i == ColoresSemaforo.VERDE){...} → Comprobación de valor(No son objetos)
...
```

Operadores

- **Operadores aritméticos:** `+, -, *, /, %`

  Observaciones: + y - seguidos de un tipo primitivo seleccionan el signo del operando y si dividimos enteros: 6/3=2 pero 5/3=1

- **Operadores relacionales:** `>, >=, <, <=, ==, !=`
- **Operadores condicionales o lógicos:** `&& , || , !, & , |`
- **Operadores de asignación de valor:** `+=, -=, *=, /=, &= , |=`

  Observaciones: Si queremos sumar o restar 1 a un valor, lo que haremos será: `i++` (postincremento), `++i` (preincremento), `i--` (postdecremento), `--i` (predecremento)

Conversiones de tipo

Implícita De tipos 'pequeños' a 'grandes': `byte` → `short` → `int` → `long` → `float` → `double`

Explícita Forzamos la conversión de tipos: `(int) 10.0, (float) (5/4)`

Orden de precedencia

La prioridad de los operadores se muestra a continuación:

Precedencia	Tipo de Operador	Operadores
1	Operadores prefijo o postfijo	`++, --, +, -, !,...`
2	Conversión de tipo	`(tipo) expresión`
3	Multiplicativos	`*, /,%`
4	Aditivos	`+, -`
5	Relacionales	`<, >, <=, >=`
6	Igualdad	`==, !=`
7	AND lógico	`&&`
8	OR lógico	`||`
9	Asignación	`=, +=, -=, *=, /=, &= , |=`

Cuadro 1.1: Precedencia

Arrays

Unidimensional La declaración típica de un array unidimensional de longitud 4 sería

```
int [] numeros = new int [4]
```

Para saber la **longitud** del array utilizamos el método `length`

```
numeros.length
```

Multidimensional La declaración de un array de varias dimensiones

```
tipo [][] referencia = new tipo [filas][columnas]
int [][] numerosZ2 = new int [4][3]
int [][][] numerosZ3 = new int [4][3][2]
```

En este caso el método `length` tiene varias funciones dependiendo de su sintaxis

`numeros.length` ⇒ Nº de filas (4)
`numeros.length[0]` ⇒ Nº de elementos de la fila 0 (3)

Observaciónes: Debemos tener cuidado con el acceso a una posición inexistente en un array. Además de la forma habitual, podemos crear un array de forma rápida:

```
m= { {0, 1, 2, 3}, {7, 8}, {40, 25, 32} };
```

1.3. Métodos

Cabecera

Tiene la siguiente forma:

```
acceso tipo nombre (parametros) excepciones
public int calcula(double a, boolean x) throws Exception {...}
```

Cuerpo

Conjunto de sentencias que realizan la función que deberá cumplir el método. Salimos del método cuando llegamos a } o hasta encontrar un `return;`.

Signatura

Forma de describir el método en base a su nombre, la clase a la que pertenezca y a los tipos de parámetros que recibe. En una misma clase no puede haber dos métodos con la misma signatura.

Invocación de métodos

Hay tres formas:

- Fuera de la clase en la que se define el método: `objeto.metodo(argumentos)`
- Fuera de la clase en la que se define un método static: `clase.metodo(argumentos)`
- Dentro de la clase en la que esta definido: `metodo(argumentos)`

1.4. Sentencias

Condicionales

- if: `if (condicion) { sentencias; }`
- if...else:

```
if (condicion) {
 sentencias;
} else {
 sentencias;
}
```

- if...else if:

```
if (condicion1) {
 sentencias;
} else if (condicion2) {
 sentencias;
} else if (condicion2) {
 sentencias;
 ...
} else {      → Opcional, pero si se pone será siempre al final
 sentencias;
}
```

- switch...case:

```
switch (expresion) {
 case x: sentencia1; break;
 case y: sentencia1; break;
 case z: sentencia1; break;
 ...
 default: sentencia;      → Opcional, pero si se pone será siempre al final
 }
```

Si no añadiéramos el `break` al darse un caso se ejecutarían las sentencias de ese caso y todas las de los casos que están por debajo de este.

Bucles

- **while:** Se ejecuta 0 ó más veces.

```
while (expresion) {
 sentencias;
}
```

- **for:** Se ejecuta un numero fijo de veces

```
for (int i=0; i<33; i++) {
 sentencias;
}
```

  Bucle for para **recorrido de colecciones:**

```
for (Pared p : paredes) {
 sentencias;
}
```

- **do...while:** Se ejecuta 1 o más veces.

```
do {
 sentencias;
} while (i<40);
```

Instrucciones dentro de bucles

- **break:** Causa la salida del bucle que lo encierra.
- **continue:** Hace que el control de flujo salte de la sentencia actual al comienzo del bucle.

1.5. Excepciones

Lanzamiento

Las excepciones son lanzadas por los métodos debido a comportamientos indebidos, para ello:

- Deben anunciarlo en la cabecera:

```
...() throws TipoDeExcepcion {
```

- Crear y lanzar el objeto Exception:

```
throw new TipoExcepcion("motivo");
```

Manejo

Para el manejo de excepciones incluimos un bloque **try...catch**

```
try {
 ... //codigo que se intenta ejecutar
} catch (TipoExcecion1 e) {
 ... //codigo que se ejecuta cuando un catch coincide con la excepcion
} catch (TipoExcecion2 e) {
 ...
} finally {
 ... //Si se pone, se ejecutará siempre
}
```

1.6. Encapsulación mediante clases

Constructores

Disponemos de la siguiente clase:

```
public class Punto{
 private double x;
 private double y;
}
```

En la que no hemos definido un constructor. *Java* introduce un constructor por defecto con parámetros nulos(en este caso `0.0` para ambos atributos) y al cual se llamaría:

```
Punto q = new Punto();
```

Si definimos nosotros el constructor en la clase, *Java* no agrega el constructor por defecto y no podemos utilizarlo.

Podemos escribir varios constructores en una clase:

```
public class Punto{
 private double x;
 private double y;

 public Punto(double x, double y) {
 this.x=x;
 this.y=y;
 }
 public Punto() {
 this(0.0, 0.0)
 }
}
```

Esta ultima sentencia llama al otro constructor pasándole `(0.0, 0.0)` como parámetros. Al crear un objeto de esta clase podemos utilizar cualquiera de los dos constructores llamándoles por su signatura.

Comparación de objetos

Hay dos formas de compararlos:

- Con == comparamos referencias y solo es true si ambas referencias apuntan al mismo objeto:

  ```
  Punto p = new Punto(1.0, 2.0);
  Punto q = new Punto(1.0, 2.0);
  Punto r = p;

  p==q; //false ⇒ No son el mismo objeto
  p==r; //true ⇒ Sí son el mismo objeto
  ```

- Con el método `equals()`, que habremos definido en la clase, lo que comparamos es el contenido de los dos objetos distintos:

  ```
  p.equals(q); //true ⇒ Porque son iguales
  ```

Control de acceso

Los tipos de acceso son:

- **Acceso público** (`public`): El acceso no está restringido y podemos usar el elemento libremente. Aplicable a **clases**, **atributos** y **métodos**.
- **Acceso privado** (`private`): Los elementos privados sólo pueden ser usados dentro de la clase que los define, lo que impide su invocación exterior. Aplicable a **atributos** y **métodos**.
- **Acceso protegido** (`protected`): Los elementos protegidos sólo pueden ser usados dentro de la clase que los define, aquellas clases que la extiendan y cualquier clase en el mismo paquete.
- **Acceso de paquete** (No se pone nada): El acceso a estos componentes es libre dentro del paquete en el que se define la clase.

Elementos de clase: `static`

Variables de clase El atributo static es compartido por todos los objetos, solo existe uno para todos ellos y si un objeto lo modifica, todos lo ven modificado.

Métodos de clase Un método static se puede llamar referido a la clase sin crear un objeto. Únicamente puede trabajar directamente sobre atributos static.

1.7. Polimorfismo y Extensión

Herencia

Partimos de este ejemplo:

Clase base:

```
public class Punto{
 private double x, y;

 public Punto(double x, double
   y){
  this.x=x; this.y=y;
 }
 public int getX(){...}
 public int getY(){...}
 public String toString(){...}
 public int distancia(Punto p){
  return Math.sqrt((x-p.x)*(x-p.x)+
     (y-p.y)*(y-p.y));
 }
}
```

Clase derivada:

```
public class PuntoPolares extends
   Punto{
 private double ro, theta;

 public PuntoPolares(double ro,
   double theta){
  super(ro*Math.cos(theta),
     ro*Math.sin(theta));
  this.ro=ro; this.theta=theta;
 }
 public int getRo(){...}
 public int getTheta(){...}
 public String toString(){...}
}
```

La clase derivada únicamente podrá acceder a los elementos públicos de la clase base. Siempre y cuando se haya declarado un constructor en la clase base, el constructor de la clase derivada deberá incluir como primera sentencia `super()` con argumentos para los parámetros que recibe el constructor de la clase base, ya que al crear un objeto de la clase derivada, también se crea un objeto de la clase base.(Nunca podremos usar `super()` y `this()` a la vez).

El elemento `super` también podemos emplearlo, siempre y cuando se haga de forma privada en la clase derivada, para llamar a métodos con la misma signatura en ambas clases (el método de la clase base quedaría *sombreado*). Por ejemplo en el ejemplo de antes podríamos haber implementado el método `toString()` de la siguiente forma:

```
public int toString(){
 return super.toString() + " , en polares: Ro " + ro + " , Theta " + theta;
}
```

Interfaces

Una interfaz refleja elementos que las clases tienen en común (normalmente métodos). En los métodos sólo se pone la cabecera, se puede incluir el lanzamiento de excepciones y nunca hay constructores ni métodos static.

```
interface Coordenada {
 double x();
 double y();
 double distancia(Coordenada q);
}
```

Nunca se podrán crear objetos de una interfaz, pero sí se pueden crear referencias que la apunten. Una clase puede implementar una o varias interfaces, para ello deberá contener en la cabecera:

```
class A implements Coordenada {...}
```

La clase que implementa la interfaz deberá incluir todos los métodos de esta y deberán tener acceso `public`. Si el método en la interfaz anuncia el lanzamiento de excepciones, podremos lanzarlas o no lanzarlas en la clase que la implementa.

Compatibilidad de Tipos

Ahora podremos definir objetos cuya clase referenciadora(puede ser una clase normal, abstracta o interfaz) sea distinta de la clase instanciadora(no puede ser una interfaz ni una clase abstracta):

```
Animal a = new Gato(...);
```

Upcasting: Una referencia puede tener objetos de su propia clase o de cualquiera de sus derivadas (como en el ejemplo anterior en el que Gato deriva de Animal).

Downcasting: Para hacerlo hay que forzar la conversión, ya que no siempre está permitido.

```
Gato g=(Gato) a;
```

Podría saltar excepción si `a` no fue instanciado como Gato.

Polimorfismo

Aunque hayamos hecho upcasting, al llamar a un método de un objeto siempre se ejecuta el de la clase instanciadora, nunca el de la clase referenciadora. Aun así, el método llamado debe existir en la clase referenciadora.

```
Animal a=new Gato(...);
a.dice(); //Llamada al método dice de la clase Gato
```

Sin embargo:

```
Object a=new Gato(...);
a.dice();      → No compila porque en Object no existe el método dice()
```

1.8. Colecciones

Listas

En una lista se respeta el orden en el que se insertan los objetos, admite duplicados y el tamaño de esta se adapta dinámicamente a lo que sea necesario. A continuación creamos una lista con objetos de tipo `E`.

```
List<E> lista = new ArrayList <E>();
```

Métodos más comunes

Los métodos que más se utilizan con listas:

`boolean add(E elemento)`	→	Añade el elemento al final de la lista
`void add(int posicion, E elemento)`	→	Añade el elemento En la posición indicada
`void clear()`	→	Elimina todos los elementos de la lista
`boolean contains(E elemento)`	→	Devuelve `true` si la lista contiene el elemento
`boolean equals(Object x)`	→	Compara el objeto indicado con la lista
`E get(int posicion)`	→	Devuelve el elemento que está en la posición indicada
`int indexOf(E elemento)`	→	Devuelve la posición del primer elemento que coincida con el especificado. Devuelve `-1` si el elemento no está
`boolean isEmpty()`	→	Devuelve `true` si no hay elementos en la lista
`E remove(int posicion)`	→	Devuelve y elimina el elemento de la posición especificada
`boolean remove(E elemento)`	→	Elimina el primer objeto que coincida con el especificado
`E set(int posicion, E elemento)`	→	Reemplaza el elemento de la posición indicada por este
`int size()`	→	Devuelve el número de elementos de la lista

Conjuntos

En los conjuntos no se respeta el orden e inserción de los objetos(los objetos no están ordenados en el conjunto), no admiten duplicados y el tamaño se adapta dinámicamente a lo que sea necesario. Para crear un conjunto de objetos de tipo `E`:

```
Set<E> conjunto = new HashSet <E>();
```

Métodos más comunes

Los métodos que más se utilizan con conjuntos:

`boolean add(E elemento)`	→	Añade el elemento al conjunto, sólo si no estaba ya
`void clear()`	→	Elimina todos los elementos del conjunto
`boolean contains(E elemento)`	→	Devuelve `true` si el conjunto contiene el elemento
`boolean equals(Object x)`	→	Compara el objeto indicado con el conjunto
`boolean isEmpty()`	→	Devuelve `true` si no hay elementos en el conjunto
`boolean remove(E elemento)`	→	Elimina el elemento especificado si está contenido en el conjunto
`int size()`	→	Devuelve el número de elementos del conjunto

Diccionarios

Los diccionarios se componen de valores(V) a los que se les asigna una clave(K). El respeto al orden varía según la clase que utilicemos, las claves no pueden estar duplicadas y el tamaño se adapta dinámicamente a lo que sea necesario. Los diccionarios se crean:

```
Map<K, V> diccionario = new HashMap <K, V>();
```

Métodos más comunes

Los métodos que más se utilizan con diccionarios son:

`void clear()`	→	Elimina todos los elementos del diccionario
`boolean containsKey(Object clave)`	→	Devuelve `true` si el diccionario contiene un elemento con esa clave
`boolean containsValue(Object value)`	→	Devuelve `true` si el diccionario contiene un elemento con ese valor
`boolean equals(Object x)`	→	Compara el objeto indicado con el diccionario
`V get(Object clave)`	→	Devuelve el valor asociado a la clave especificada
`boolean isEmpty()`	→	Devuelve `true` si no hay elementos en el diccionario
`Set<K> keySet()`	→	Devuelve un conjunto con las claves del diccionario
`V put(K clave, V valor)`	→	Asocia o añade el valor especificado con la clave especificada. Devuelve `null` si no existía un elemento con tal clave
`V remove(Object clave)`	→	Elimina el elemento(clave y valor) cuya clave es la especificada. Devuelve el valor que tenía asociado o `null` si no existía un elemento con tal clave
`int size()`	→	Devuelve el número de elementos del diccionario

1.9. Estilo y Pruebas

Documentación en Javadoc

A continuación se muestra un ejemplo de una clase con su correspondiente documentación en Javadoc:

```
/**
 * Descripción de la clase
 *
 * @author juanmanuel
 * @version 3.4
 * @see clase Geometría
 *
 */
public class Circulo{
 private int radio;

/**
 * Funcionalidad del método
 *
 * @param radio se utiliza para...
 *
 */
 public Circulo(int radio){
  this.radio=radio;
 }
/**
 * Funcionalidad del método
 *
 * @return El radio del circulo
 * @exception excepcion se lanza excepcion si ... //tambien con @trows
 *
 */
 public int getRadio(){
  return this.radio;
 }
}
```

Pruebas en JUnit4

Para crear una clase de pruebas en Java, definiremos primero las variables privadas como atributos que necesitemos. Es importante que nuestra clase de pruebas contenga un método etiquetado como `@Before` que inicialice las variables privadas que hayamos definido. Después de ese método probaremos cada uno de los métodos de la clase, para ello incluiremos la etiqueta `@Test` seguido de la cabecera del método de pruebas que implementaremos empleando la siguiente serie de métodos(previamente habiendo importado `import static org.junit.Assert.*;`)

`assertEquals(X esperado, X real)`	→	Compara un resultado esperado con el resultado obtenido, determinando que la prueba pasa si son iguales y que falla si son diferentes
`assertArrayEquals(Object[] esperado, Object[] real)`	→	Compara un array esperado con el array obtenido
`assertSame(X esperado, X real)`	→	Comprueba que los dos parámetros son exactamente el mismo objeto (==)
`assertNotSame(X esperado, X real)`	→	Comprueba que los dos parámetros **no** son exactamente el mismo objeto
`assertFalse(boolean resultado)`	→	Verifica que el resultado es `false`
`assertTrue(boolean resultado)`	→	Verifica que el resultado es `true`
`assertNull(Object resultado)`	→	Verifica que el resultado es `null`
`assertNotNull(Object resultado)`	→	Verifica que el resultado no es `null`
`fail`	→	Sirve para detectar que estamos en un sitio del programa donde NO deberíamos estar

Por ejemplo una clase de pruebas de la clase Circulo podría ser:

```
public class TestCirculo{
 private int radio1, radio2;

 @Before
 public inicializa(){
  radio1= 3; radio2= 2
 }

 @Test
 public void testGetRadio(){
  Circulo c1 = new Circulo(radio1);
  Circulo c2 = new Circulo(radio2);
  assertEquals(radio1, c1.getRadio();
  assertEquals(radio2, c2.getRadio();
 }

 @Test
 public void testArea(){ ... }
}
```

Capítulo 2

Diseño de Algoritmos

2.1. Algoritmos recursivos

Introducción

Partiendo de la estructura general de un método cualquiera:

```
double distancia (Punto q){
  double dx= x - q.getX();
  double dy= y - q.getX();
  return Math.sqrt(dx*dx + dy*dy);
}
```

Donde `q` es el argumento del método y `dx` y `dy` son variables locales que ha creado éste para realizar su función. Estos argumentos y variables locales se crean con la llamada al método y desaparecen al terminar de realizar éste su labor. Si hay varias llamadas simultáneas al método, cada una tendrá sus propias variables; lo cual permite que un método se pueda volver a llamar a sí mismo(**llamada recursiva**).

Los métodos recursivos son métodos que contienen llamadas recursivas y que en cada llamada convergen a una condición de parada que el programador habrá establecido. Por ejemplo: *Cálculo del factorial de un numero:*

```
... int factorial(int n){
  if(n < 1) return 1;
  return n*factorial(n - 1)
}
```

La condición de parada es que `n` sea menor que 1 y al llamar al método es seguro que convergerá a la condición de parada ya que se actualiza el parámetro pasado como argumento en cada llamada. En este caso se ha decidido que el método calculará el factorial de un número y devolverá un 1 en cuanto el parámetro sea menor que 1. En cambio, podría haberse utilizado una excepción para indicar que no se puede obtener un resultado válido por ejemplo si `n` es negativo. La traza de ejecución si llamamos pasando como parámetro 5 será:

```
factorial(5) = 5*factorial(4)
     = 5*(4*factorial(3))
     = 5*(4*(3*factorial(2)))
     = 5*(4*(3*(2*factorial(1))))
     = 5*(4*(3*(2*(1*factorial(0)))))
     = 5*(4*(3*(2*(1*1))))
     = 120
```

Como se ya se ha dicho anteriormente, en cada llamada al método recursivo se crean variables locales independientes. Se podría pensar si se puede programar el algoritmo de forma iterativa, pero habría que plantearse si se está optimizando el tiempo de ejecución y la memoria utilizada.

Recursión frente a iteracción

La recursión y la interacción son formas de conseguir que un programa haga muchas veces lo mismo con alguna variante y ambas se pueden programar la una a partir de la otra aunque:

- **Iteración → Recursión** Normalmente es sencillo de programar.
- **Recursión → Iteración** Puede ser muy complicado de programar.

Por ejemplo se presentan dos formas de programar como obtener los términos de la *serie de Fibonacci*:

Forma recursiva

```
int fibonacci(int n) {
  if (n < 1) {
    return 0;
  }
  if (n == 1) {
    return 1;
  }
  return fibonacci(n - 1) +
              + fibonacci(n - 2);
}
```

Forma iterativa

```
int fibonacci2(int n) {
 if (n < 1) {
   return 0;
 }
 int f0 = 0;
 int f1 = 1;
 for (int i = 2; i < n; i++) {
   int fn = f0 + f1;
   f0 = f1;
   f1 = fn;
 }
 return f1;
}
```

¡¡MUY IMPORTANTE!! Hay que tener especial cuidado con estos casos:

* **Fallo en la condición de parada:** El programa la pase por alto entrando en un bucle de llamadas infinito.
* **Convergencia a la condición de parada:** El programa no actualiza la variable en cada llamada.
* **Variables de clase:** Atributos `static` de los que solo hay uno que comparten todos.

2.2. Análisis de complejidad

Se define algoritmo como serie de pasos que llevan a resolver, de forma efectiva, un problema. Un problema podrá resolverse habitualmente con varios algoritmos distintos. Se define programa como un algoritmo escrito usando un lenguaje de programación. Un algoritmo se podrá programar de distintas formas, más o menos eficientes. No todos los algoritmos son iguales, unos requieren más recursos que otros(más tiempo de ejecución, más memoria,...), al igual que no todos los programas son iguales, hay mejores programadores y mejores compiladores que optimizan de una forma u otra los recursos requeridos por el programa.

Problemas y algoritmos

No se conocen algoritmos para todos los problemas, hay problemas de los que se sabe que no hay algoritmos que los resuelvan (*problemas no computables*). Hay problemas para los que todos los algoritmos conocidos necesitan una cantidad de recursos que hace inviables los algoritmos que los solucionan(*problemas intratables*).

Complejidad

Para enfocar la comparación de algoritmos se mide primero cómo crece el consumo de un recurso(tiempo de ejecución, espacio en memoria...) en función del tamaño n del problema y a continuación se analiza la función $f(n)$. Normalmente se compara ese algoritmo $f(n)$ con otro algoritmo $g(n)$ para saber cual de ellos es más eficiente.

La complejidad se define como una relación de orden total entre funciones de consumo de recursos ($f(n)$ y $g(n)$), para estimarla calculamos:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = K$$

- si K = ∞, se dice que $f(n)$ tiene mayor complejidad que $g(n)$.
- si K = 0, se dice que $f(n)$ tiene menor complejidad que $g(n)$.
- si K $\neq \infty$ y K $\neq 0$, se dice que $f(n)$ y $g(n)$ tienen la misma complejidad.

Funciones de referencia

Se suelen manejar estas funciones porque son sencillas y aparecen con mucha frecuencia.

Función	**Nombre**
$f(n) = 1$	Constante
$f(n) = log(n)$	Logaritmo
$f(n) = n$	Lineal
$f(n) = n \cdot log(n)$	
$f(n) = n^2$	Cuadrática
$f(n) = n^a$	Polinomio (de grado a > 2)
$f(n) = a^n$	Exponencial (a > 2)
$f(n) = n!$	Factorial

Ordenes de complejidad

Es un conjunto de funciones que crecen de la misma forma y por tanto son igual de complejas. Habitualmente se denominan "$O(\cdot)$". Para cada uno de estos conjuntos se suele identificar un miembro $f(n)$ que se utiliza como representante de la clase:

Orden	**Nombre**
$O(1)$	Orden constante
$O(log(n))$	Orden logaritmo
$O(n)$	Orden lineal
$O(n \cdot log(n))$	
$O(n^2)$	Orden cuadrático
$O(n^a)$	Orden polinomial(a > 2)
$O(a^n)$	Orden exponencial (a > 2)
$O(n!)$	Orden factorial

La siguiente tabla muestra el comportamiento de diferentes funciones para valores crecientes de n:

n	O(1)	O(lg n)	O(n)	O(n lg n)	$O(n^2)$	$O(n^5)$	$O(5^n)$	O(n!)
10	1µs	2µs	10µs	23µs	100µs	100ms	10s	4s
20	1µs	3µs	20µs	60µs	400µs	3s	3a	63 mil años
30	1µs	3µs	30µs	102µs	900µs	20s	28 millones de años	
40	1µs	4µs	40µs	148µs	1,6ms	100s		
50	1µs	4µs	50µs	196µs	2,5ms	300s		
60	1µs	4µs	60µs	246µs	3,6ms	800s		
70	1µs	4µs	70µs	297µs	4,9ms	33m		
80	1µs	4µs	80µs	351µs	6,4ms	50m		
90	1µs	4µs	90µs	405µs	8,1ms	1h40m		
100	1µs	5µs	100µs	461µs	10ms	2h46m		

Como se puede apreciar, las funciones de mayor orden de complejidad crecen de forma más desmesurada cuando el volumen de datos se hace mayor.

Cálculo de complejidad

A continuación se enumeran un conjunto de reglas prácticas para el cálculo de la complejidad algorítmica:

1. Las **sentencias sencillas** como **instrucciones** que se ejecutan **una sola vez** tienen complejidad $O(1)$ y los **conjuntos** de k **sentencias sencillas** $k \cdot O(1) \rightarrow O(1)$

2. La complejidad de una **secuencia** de instrucciones es la **suma** de las complejidades que la **forman**.

   `s1;` $\in O(1)$
   `s2;` $\in O(1)$
   $O_{Total} = O(1) + O(1)$

3. La complejidad en una **estructura de selección** es la suma de la evaluación de la condición más la complejidad de el caso más complejo de las posibles selecciones.

   `if(c1) { s1; }` c1 $\in O(1)$
   `else if(c2) { s3; }` c2 $\in O(n)$, s3 $\in O(log(n))$
   `else { s2; }`
   $O_{Total} = O(n) + O(log(n))$

4. Si $f(n) \in O(g) \rightarrow k \cdot f(n) \in O(g)$ donde k es una constante.

5. **Propiedad de suma**:

   - $f_1(n) \in O(g)\ y\ f_2(n) \in O(g) \quad \rightarrow \quad f_1(n) + f_2(n) \in O(g)$
   - $f_1(n) \in O(g)\ y\ f_2(n) \in O(h) \quad \rightarrow \quad (f_1(n) + f_2(n)) \in O(g + h)$
   - $O(g) \subset O(h) \quad \rightarrow \quad O(g + h) \in O(h)$

6. **Propiedad de multiplicación**: $f_1(n) \in O(g)\ y\ f_2(n) \in O(h) \quad \rightarrow \quad (f_1(n) \cdot f_2(n)) \in O(g \cdot h)$

7. **Potencia**: Si $a < b \quad \rightarrow \quad O(n^a) \subset O(n^b)$

8. Un **polinomio de grado** k tiene **complejidad** $O(n^k)$

9. **Bucles:**
   La complejidad en **bucles** es la suma de la complejidad de la evaluación de la condición y la complejidad del cuerpo del bucle multiplicado por la complejidad del cálculo del número de interacciones del bucle.

   ```
   while (c1) {
     s1; }
   ```
   $O_{Total} = O(1) \cdot O(1)$

   - Hay **algunos casos particulares** en bucles que tienen condición $\in O(1)$:
     - Si el **número** de **interacciones** es **fijo** (es un número entero k que no cambia durante la ejecución del bucle) e independiente al tamaño N del problema y el **cuerpo** del bucle es $O(1)$:
       ```
       for (int i = 0; i < k; i++){
         s1; }
       ```
       $O_{Total} = k \cdot O(1) \rightarrow O(1)$
     - Si el **número** de **interacciones** depende del tamaño N del problema y el cuerpo del bucle es $\in O(1)$:
       ```
       for (int i = 0; i < N; i++){
         s1; }
       ```
       $O_{Total} = N \cdot O(1) \rightarrow O(n)$
   - Si el **número** de **interacciones** depende del tamaño N del problema y el cuerpo del bucle es $\in O(n)$:
     ```
     for (int i = 0; i < N; i++){
       for (int j = 0; j < N; j++){
        s1; } }
     ```
     $O_{Total} = N \cdot N \cdot O(1) \rightarrow O(n^2)$
   - Si el bucle externo se realiza N veces, el interno $1, 2, 3, ...N$ veces y el cuerpo del último bucle es $\in O(1)$:
     ```
     for (int i = 0; i < N; i++){
       for (int j = 0; j < i; j++){
        s1; } }
     ```
     $O_{Total} = 1 + 2 + 3 + ... + N = \frac{N \cdot (1+N)}{2} \rightarrow O(n^2)$ (Suma de serie aritmética)
   - Si la **variable de control** (`i`) **no** varía de **forma lineal**:
     ```
     int i = 1;
     while (i < N ) {
       s1;
       i *= 2; }
     ```
     Al cabo de k iteraciones el valor de `i` será 2^k. Por tanto: $2^k \geq N \rightarrow k = log_2(N)$
     Con lo cual: $O_{Total} = O(log(n))$
   - Si el bucle externo se ejecuta un número de veces dependiente de N y el bucle interno $\in O(log(n))$:
     ```
     for (int i = 0; i < N; i++){
       int c = i;
       while (c > 0 ) {
        s1;
        c /= 2; } }
     ```
     $O_{Total} = N \cdot O(log(n)) = O(n \cdot log(n))$

10. En las **llamadas a métodos** la complejidad total es la **peor** de las **complejidades** de su **interior**

Ejemplo: Sucesión de Fibonacci

La serie de Fibonaccci es una serie de números naturales muy conocida y estudiada. Su definición es:

$$f(0) = 1$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)\ para\ n \geq 2$$

Comparamos distintas soluciones para calcular $f(n)$:

- **Algoritmo recursivo**

```
int fibo1(int n) {
 if (n < 2) {
   return 1;
 } else {
   return fibo1(n - 1) + fibo1(n - 2);
 }
}
```

- **Algoritmo recursivo con memoria o cache**

```
int[] tabla = new int[20];
int fibo3(int n) {
 if (n < 2)
   return 1;
 if (n < tabla.length) {
   int resultado = tabla[n];
   if (resultado > 0)
     return resultado;
   resultado = fibo3(n - 1) + fibo3(n - 2);
   tabla[n] = resultado;
   return resultado;
 }
 return fibo3(n - 1) + fibo3(n - 2);
}
```

- **Algoritmo iterativo**

```
int fibo4(int n) {
 int n0 = 1;
 int n1 = 1;
 for (int i = 2; i <= n; i++) {
   int ni = n0 + n1;
   n0 = n1;
   n1 = ni;
 }
 return n1;
}
```

- **Cálculo directo** Calculando directamente mediante la fórmula de Binet:

$$f(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

```
static final double SQRT_5 = Math.sqrt(5);
int fibo5(int n) {
 if (n < 2)
   return 1;
 n += 1;
 double t1 = Math.pow((1 + SQRT_5) / 2, n)
      - Mat.pow((1 - SQRT_5) / 2, n);
 return (int) Math.round(t1 / SQRT_5);
}
```

Si se miden los tiempos de ejecución de cada una de los algoritmos para varios casos, se obtiene lo siguiente:

N	fibo1	fibo3	fibo4	fibo5
10	812.001	699.112	6.569	158.460
20	3.397.022	723.332	6.979	154.766
30	392.534.843	8.739.927	6.979	153.945
40		57.582.944	7.800	153.944
50		2.947.764.873	8.621	153.945
	$O(1{,}6^n)$	O(n)	O(n)	O(1)

Donde se muestra el orden de complejidad de los algoritmos, siendo el cálculo directo mediante la fórmula de Binet la opción más simple y el algoritmo recursivo el menos eficiente.

Conclusiones

Finalmente se ha de analizar el algoritmo que se va a programar. Para problemas de tamaño pequeño, simplemente, hay que optimizar el código, para problemas de tamaño mayor ($n \to \infty$), además, hay que elegir un buen algoritmo. Algunos algoritmos son muy buenos para un n grande pero son muy ineficientes para un n pequeño.

Si en un programa se combinan dos o más algoritmos, el que tenga mayor complejidad impone su ley, ya que cuando se manejan grandes volúmenes de datos, el de baja complejidad tiene una contribución despreciable $\left(\lim_{n\to\infty}(an^2+bn+c) \approx an^2\right)$. Por lo que habrá que empezar modificando el algoritmo peor.

Problemas NP

Hasta ahora se ha hablado de algoritmos para solucionar problemas. Una clase de problemas son los NP, que se caracterizan porque requieren probar todas las posibilidades para encontrar la solución y un tiempo polinómico para saber si se ha encontrado la solución buscada. Por ejemplo: *Dado un conjunto de números enteros, ¿existe un subconjunto no vacío de ellos donde la suma de sus elementos sea 0?*

$\{-2, -3, 15, 14, 7, -10\}$

La solución buscada sería: *Sí existe y es el* $\{-2, -3, 15, -10\}$

2.3. Algoritmos para diccionarios

Ya se conoce una forma de implementar diccionarios utilizando la interfaz `Map` predefinida entre las colecciones de *Java* en la cual se asigna a cada valor contenido una clave. También se podría implementar un diccionario utilizando la interfaz `Set` de *Java*.

Si se definiese una interfaz genérica que implementaran todos los diccionarios, esta debería tener los siguientes métodos:

- Un método que introduzca un nuevo valor en el diccionario.

```
public void put(Comparable clave, Object valor);
```

- Un método que obtenga el valor asociado a una clave y que devuelva null si no está la clave.

```
public Object get(Comparable clave);
```

- Un método que elimine el objeto asociado a la clave del diccionario y lo devuelva. Devolvería null si no estuviera la clave.

  ```
  public Object remove(Comparable clave);
  ```

- Un método que devuelva el número de elementos almacenados en el diccionario.

  ```
  public int size();
  ```

La clave pertenecerá a una clase que implemente la interfaz `Comparable` que nos permite establecer una relación de orden ($a < b$, $b > c$) entre objetos de clases la implementan.

```
interface Comparable {
 int compareTo()
}

int cmp = clave1.compareTo(clave2)
```

Donde `cmp` tomará los siguientes valores:

- `cmp` < 0 si `clave1` $<$ `clave2`
- `cmp` $= 0$ si `clave1` $=$ `clave2`
- `cmp` > 0 si `clave1` $>$ `clave2`

Puede ocurrir que se intentara introducir una clave que ya está contenida en el diccionario con un valor distinto, en ese caso se utilizará el algoritmo más sencillo, indicando si se reemplaza el valor o se duplica. Si existieran duplicados en el diccionario al obtener o al borrar valores se sacaría o borraría lo primero que se encuentre.

Implementación de diccionarios con arrays

En este caso la complejidad del diccionario varía en función de como se implemente, si se tiene el array ordenado o desordenado, si acepta duplicados...

Los métodos definidos anteriormente tendrían las siguientes complejidades:

- `put` tendría $O(n)$ si se recorre el array primero en busca de duplicados o en cambio, $O(1)$ si se añade el elemento al final sin comprobar la existencia de estos.
- `get` y `remove` tendrían ambos complejidad de $O(n)$ ya que habría que recorrer el array comparando.

Arrays ordenados

Con objeto de simplificar la complejidad en la búsqueda de datos en un array ordenado se utiliza la **búsqueda binaria** que consiste en:

1. Mirar en la **mitad del array**, si ahí esta el dato buscado, esa es la respuesta
2. Si en la mitad del array hay un **dato demasiado grande**, se busca entonces por la **izquierda**
3. **Si no**, hay que buscar por la **derecha**

Por ejemplo, para buscar en un rango [a, z) en el array `claves[]`:

```
private int busca(Comparable clave,
          int a, int z) {
  while (a < z) {
  int m = (a + z) / 2;
  int cmp = claves[m].compareTo(clave);
  if (cmp == 0)
    return m;
  else if (cmp < 0)
    a = m + 1;
  else
    z = m;
  }
  return a;
}
```

Por ejemplo si se quisiera buscar en el array:
`int[] datos = 2, 3, 5, 7, 11, 13, 17, 19;`

- La posición del 3:
  `busca(3, 0, 8)`
  `a` = 0; `z` = 8; `m` = 4; `datos[m]` = 11
  `a` = 0; `z` = 4; `m` = 2; `datos[m]` = 5
  `a` = 0; `z` = 2; `m` = 1; `datos[m]` = 3
  `return 1` ← **Posición del dato**

- En que posición **estaría** el 4:
  `busca(4, 0, 8)`
  `a` = 0; `z` = 8; `m` = 4; `datos[m]` = 11
  `a` = 0; `z` = 4; `m` = 2; `datos[m]` = 5
  `a` = 0; `z` = 2; `m` = 1; `datos[m]` = 3
  `a` = 2; `z` = 2
  `return 2` ← **Posición en la que estaría el dato**

Con la **búsqueda binaria** la complejidad de encontrar un dato en un array se reduce de $O(n)$ a $O(log(n))$ y la forma más eficiente de implementarla es mediante un bucle(**iterativa**), por delante de la recursiva.

Ordenación perezosa

A la hora de programar diccionarios cuyos datos requieran un cierto orden para el usuario, hay que plantearse cual es la opción mas interesante, que dependerá de la complejidad de cada operación:

→ Inserción ordenada y búsqueda binaria

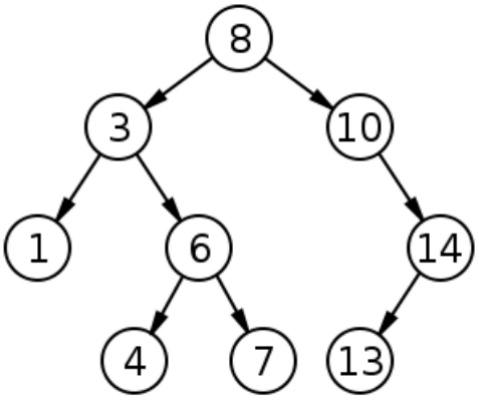
→ Inserción sin ordenar y búsqueda lineal

→ Inserción, ordenación y búsqueda binaria

La **ordenación perezosa** se basa en meter datos sin importar el orden y ordenarlos cuando se necesite extraer alguno. Habrá que valorar en cada caso cual es la ordenación optima en función de lo que se necesite.

Árboles binarios de búsqueda

Para facilitar el problema de reordenar un diccionario en el que se han ido introduciendo datos con la *"ordenación perezosa"* se utilizan estructuras de datos llamadas **árboles binarios de búsqueda**, cuya representación se hace mediante grafos:



Donde cada elemento que lo conforma se llama nodo y cada *nodo padre* tiene, como mucho, dos *hijos*. Todos los descendientes por la **izquierda** del nodo tienen valores **menores** que el *padre* y todos los descendientes por la **derecha** tienen valores **mayores** que el *padre*. Además en un árbol binario **no hay duplicados**.

En *Java* un arbol binario que implementa la interfaz `Diccionario`, semi-definida al comienzo del punto **3**, está compuesto por nodos y tiene la siguiente estructura:

```
public class DiccionarioArbol
 implements Diccionario {
   private class Nodo {
     Comparable clave;
     Object valor;
     Nodo izq;
     Nodo der;
 }
 private Nodo raiz;
 private int n;
```

Quedando la estructura del nodo:

La **búsqueda** en el árbol binario se hace, al igual que en cualquier diccionario, con el método `get`; en este caso se ha programado un método privado auxiliar que devuelve el objeto buscado a partir de un nodo y una clave pasados como parámetro:

```
 public Object get(Comparable clave) {
   return get(raiz, clave);
 }

 private Object get(Nodo nodo, Comparable clave) {
   if (nodo == null)
     return null;
   int cmp = nodo.clave.compareTo(clave);
   if (cmp == 0)
     return nodo.valor;
   else if (cmp > 0)
     return get(nodo.izq, clave);
   else
     return get(nodo.der, clave);
 }
```

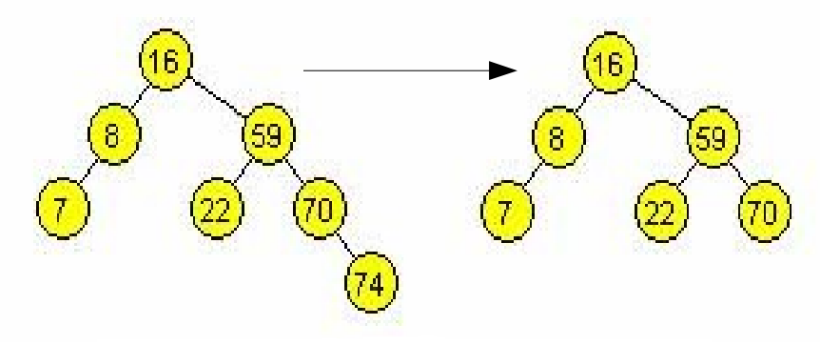
La **inserción** se hace, como anteriormente, con el método `put`. Cabe destacar de nuevo, que en un árbol binario **no** puede haber **duplicados**, la solución optada es reemplazar el valor en caso de que la clave ya exista en el árbol. De nuevo se ha definido un método auxiliar privado para simplificar el proceso:

```
 public void put(Comparable clave, Object valor) {
   raiz = put(raiz, clave, valor);
 }

 private Nodo put(Nodo nodo, Comparable clave, Object valor) {
   if (nodo == null) {
     nodo = new Nodo();
     nodo.clave = clave;
     nodo.valor = valor;
     n++;
     return nodo;}
```

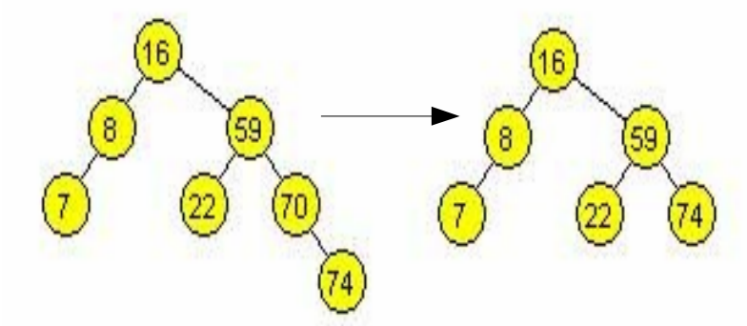
```
  Comparable claveDelNodo = nodo.clave;
  int cmp = claveDelNodo.compareTo(clave);
  if (cmp == 0)
    nodo.valor = valor;       // reutilizamos el nodo
  else if (cmp > 0)
    nodo.izq = put(nodo.izq, clave, valor);
  else
    nodo.der = put(nodo.der, clave, valor);
  return nodo;
}
```

A la hora de **eliminar** un nodo, pueden darse tres casos:

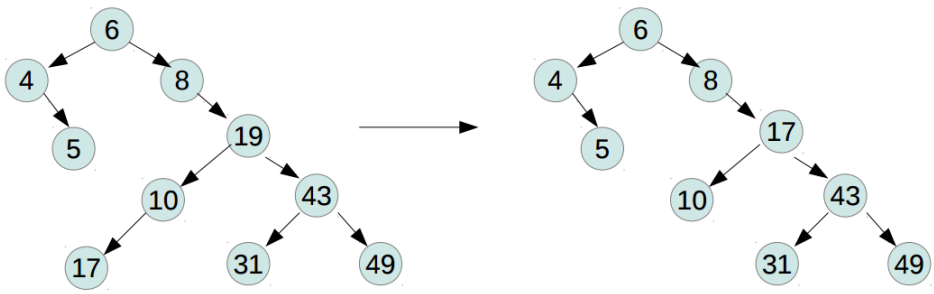
- que el nodo **no** tenga *hijos*, en este caso se borra el nodo tal cual está. *Ejemplo: Borrar el 74:*



- que el nodo tenga **un *hijo***, en cuyo caso se borra y se reemplaza por el *hijo*. *Ejemplo: Borrar el 70:*



- que el nodo tenga **dos *hijos***, en este tercer caso se busca el mayor *hijo* izquierdo (o el menor *hijo* derecho) y se reemplaza por el que se quiere eliminar. *Ejemplo: Borrar el 19:*

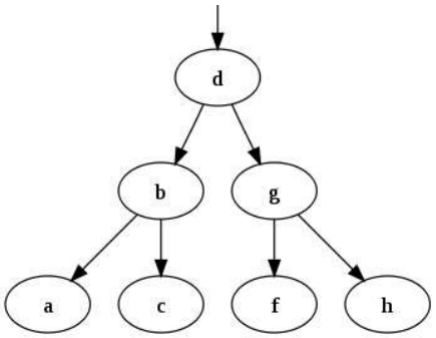


Complejidad de un árbol binario

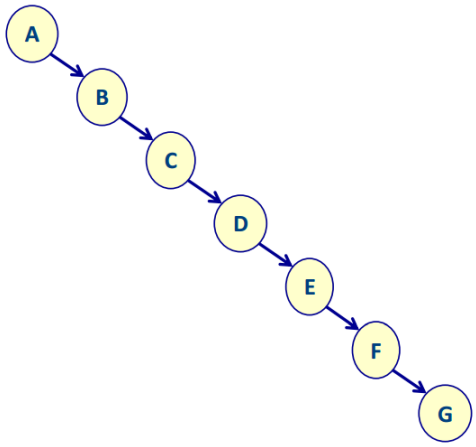
Para analizar la complejidad de un árbol binario primero han de definirse dos conceptos sobre su estado:

Árbol binario lleno: Es aquel en el que cada nodo tiene o dos o ningún hijo, pero en ningún caso un nodo *padre* puede tener un solo *hijo*.

Árbol binario completo o equilibrado: es un árbol binario completamente lleno con la posible excepción del último nivel.



Árbol binario degenerado: Árbol creado a partir de una lista de valores ordenada; es decir, que los valores se han ido introduciendo uno tras otro respetando su propio orden.



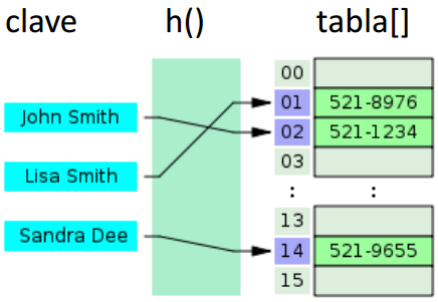
Todas las operaciones recorren el árbol para realizar, al menos, una búsqueda. Tomando el árbol degenerado como excepción y considerando árboles equilibrados, la complejidad de todas ellas se reduce a orden logarítmico($O(log(n))$):

- **Recuperación(get)** e **Inserción(put)**: Si se trata de un árbol completo: $O(log(n))$, en cambio, si se degenera: $O(n)$
- **Eliminación(remove)**: La complejidad se compone de una combinación de las anteriores: $O(busca\ nodo) + O(busca\ reemplazo) + O(inserta\ reemplazo\ en\ nodo\ original) =$
  $= O(log(n)) + O(log(n)) + 1 \rightarrow O(log(n))$

Tablas *hash*

Se ha visto el punto anterior que se podía reducir la complejidad de búsqueda a orden logarítmico($O(log(n))$). Pero se puede reducir, aún más, la complejidad de las operaciones de inserción y búsqueda a complejidad constante($O(1)$) mediante el uso de las **tablas *hash***.

La idea se basa en que se tiene una tabla holgada con espacio para muchos valores. A cada valor se le asigna una clave que habitualmente será única, aunque no tiene por qué, que se calcula mediante una fórmula matemática(**fución *hash***).



Ejemplo: Una agenda telefónica

Función hash

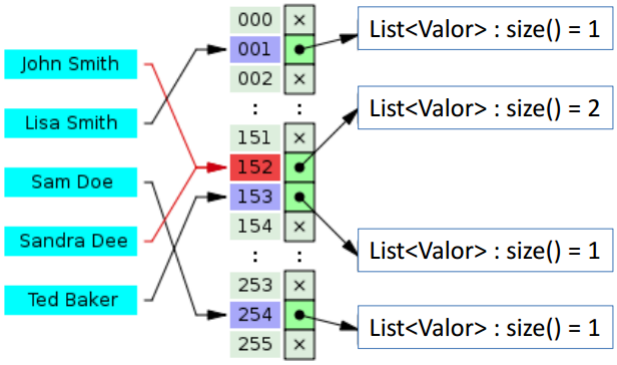
Es una función que toma un objeto y devuelve un "*valor hash*" que normalmente es un entero. Por ejemplo si tenemos una tabla de N posiciones, la función *hash* generará un valor entre 0 y $N-1$ para cada objeto que reciba. En *Java*, todos los objetos tienen un método `int h()` ó `int hashcode()` que genera el código *hash* identificativo implementado de diferentes formas según la clase.

Si no se está programando en *Java* o no se quiere utilizar el `hashcode()` que viene por defecto, habrá que inventar algún algoritmo que sea rápido y que disperse bien los valores evitando las colisiones. Por ejemplo, podría implementarse el siguiente código para obtener claves *hash* en la clase `String`:

```
int h(String s) {
  int n = 0;
  for (char ch : s.toCharArray())
    n = n * 31 + ch;
  return n;
}
```

Si se produce una **colisión** porque el método `h()` ha devuelto el mismo valor para dos objetos diferentes, podríamos optar por dos soluciones:

1. **Direccionamiento cerrado:** Hacer una **lista** con los objetos que tienen la misma clave y realizar una búsqueda lineal en esta cada vez que se quiera obtener uno de los objetos.



La **complejidad** de esta solución depende de la complejidad de las listas. Se realizan operaciones de **inserción**, **búsqueda** y **eliminación** por tanto el orden de complejidad sería: $O(1) + O(operacion\ en\ la\ lista)$. Si la tabla tiene tamaño k y las claves están uniformemente distribuidas:

`lista.size()` $= \frac{n}{k}$ La complejidad quedaría $O(1) + O(\frac{n}{k}) = O(n)$

2. **Direccionamiento abierto:** Buscar otros posibles huecos en la tabla *hash*, de **forma metódica**, hasta que se encuentre un hueco vacío.

   Si la búsqueda(y posterior inserción) de huecos se hace de **forma lineal**(posición $+ i$) se ocuparán los huecos de forma consecutiva formando ristras de valores y si el volumen de datos es muy grande se acabará recorriendo la tabla hasta el final, aumentando con ello los tiempos del proceso.

   Si en cambio la búsqueda se hace de **forma cuadrática**(posición $+ i^2$) se conseguirá solucionar el problema de las ristras de valores, al estar menos aglomerados y existiendo huecos entre ellos.

   Para solucionar aún más el problema, se podría utilizar el ***hashing* doble** que consiste en generar un numero *hash* alternativo($h2$) e intentar guardarlo en: posición $+ i \cdot h2$. Se forman ristras de valores en la tabla, aunque contiene también huecos; con lo cual es poco probable que haya que recorrer la tabla entera en busca de estos.

   Con esta solución si la tabla está **ligeramente ocupada**, hay pocas colisiones y se resuelven rápidamente, con lo cual la **complejidad** se reduce a $O(1)$. En cambio si la **tabla** se **llena**, hay muchas colisiones y esto exige recorrer la tabla muchas veces, por tanto la **complejidad** aumenta a orden lineal ($O(n)$).

Interfaz `Map`

Un ejemplo sencillo de integración de tablas *hash* en *Java* son los diccionarios construidos a partir de la **interfaz** `Map`. Hay dos clases especialmente importantes que la implementan:

- **HashMap:** Utiliza tabla *hash* con **listas** parra colisiones. En el caso de que la tabla se llene, se duplica su tamaño y se reubican los valores en nuevas listas. Además, intenta garantizar $O(1)$.
- **TreeMap:** Implementa un diccionario con **árboles de búsqueda** y, por tanto, mantiene los datos ordenados. Como ya se vió, las operaciones, en general, tienen complejidad logarítmica $O(log(n))$.

Tabla resumen de complejidad

Para finalizar, se muestra a continuación una tabla resumen de todo lo visto en cuanto a complejidad, en el mejor de los casos, en el tema:

	put	get	remove
arrays	O(1) [con dups]	O(n)	O(n)
arrays ordenados	O(n)	O(log n)	O(n)
árbol binario	O(log n)	O(log n)	O(log n)
hash con listas	O(1) si k > N O(N) si k << N		
hash abierto carga << 1	O(1)		
HashMap (con rehash)	O(1)		
TreeMap	O(log n)		

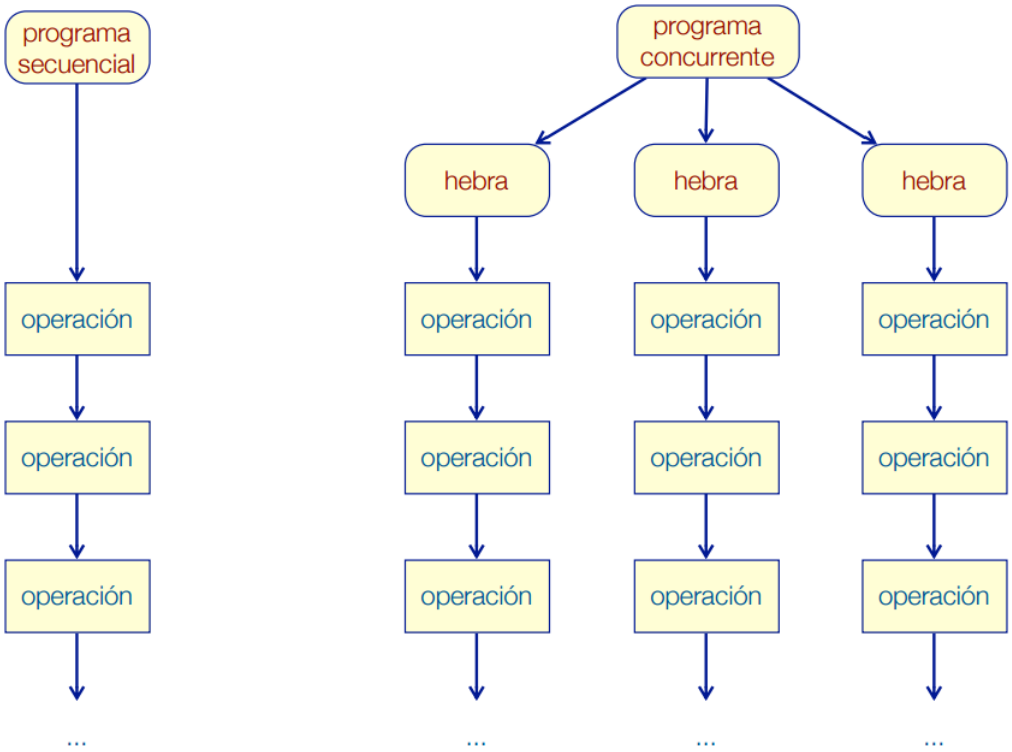
Capítulo 3

Programación concurrente

3.1. Introducción

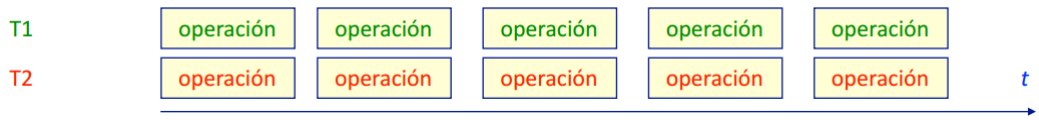
Hasta ahora se ha trabajado con programas que ejecutan una operación tras otra, en secuencia; aunque el orden podía variar (mediante estructuras de selección, bucles,...). Esta clase de programas únicamente podían hacer una sola cosa a la vez ya que solo había un flujo de ejecución que seguía la secuencia de sentencias. A partir de ahora, estos programas pasarán a denominarse **programas secuenciales**.

Habitualmente se necesita que un programa haga varias cosas al mismo tiempo, es decir que varias tareas o actividades progresen en paralelo. Para solucionar este problema se crean los **programas concurrentes** que tienen varios flujos de ejecución y cada uno de ellos ejecuta una secuencia de operaciones. En *Java* se llaman *threads* o hebras.

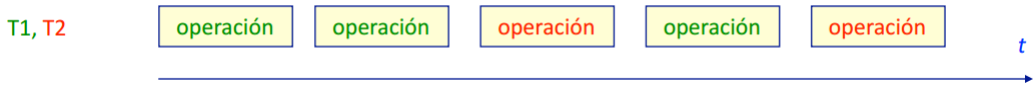


Este modelo de programas se ejecutan de diferentes formas dependiendo del número de procesadores del sistema:

- En un sistema con varios procesadores se puede ejecutar cada tarea en un procesador. La ejecución de cada una es simultánea (**paralelismo físico**).



- En un sistema monoprocesador se intercalan las operaciones de las tareas, es decir, se hace multiplexación en el tiempo (**paralelismo lógico**). Se podría decir en este caso que la concurrecia es *simulada* y será la máquina virtual o el sistema operativo el que se encargue de realizar el proceso



Desde el punto de vista lógico, ambas formas son equivalentes a la hora de realizar las funciones del programa de cara al usuario.

Esta forma de programación es muy útil en la gran mayoría de los casos; por ejemplo permiten un mayor aprovechamiento de las *CPUs*, permiten poder dejar una tarea esperando algo mientras las demás progresan (interfaces de usuario reactivas, servidores con varios clientes, sistemas de control,...). En general todos los programas son concurrentes.

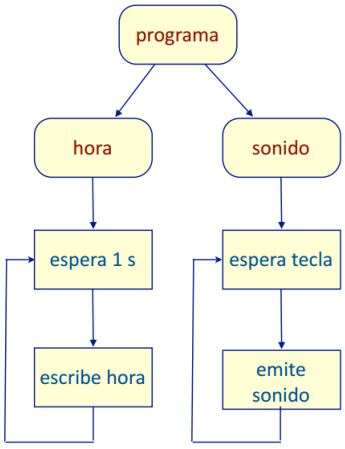
Los programas concurrentes deben cumplir las siguientes cuatro propiedades fundamentales que el programador debe asegurar:

- Corrección
- Seguridad
- Vivacidad
- Equidad

Ejemplo de programa concurrente

Se plantea hacer un programa (que se terminará de elaborar más adelante) que realice dos actividades:

- Que escriba la hora cada segundo
- Que emita un sonido cuando se pulse la tecla *intro*



Este programa sería muy complicado de hacer con un programa secuencial, por lo que se recurre a la programación concurrente con dos hebras:

La hebra encargada de imprimir la hora contendrá, entre otras cosas, lo siguiente:

```
while(true) {
  sleep(1000);
  System.out.println(
  new Date().toString());
}
```

Y la encargada de emitir el sonido:

```
while(true) {
  nextLine();
  beep();
}
```

3.2. Hebras o *threads* en *Java*

En *Java* las actividades concurrentes se llaman ***threads*** o **hebras**. Para su implementación conviene conocer la clase `Thread` y la interfaz `Runnable` que se muestran a continuación:

```
public interface Runnable {
 public void run ();
}
```

```
public class Thread implements Runnable {

 /* constructores */
 public Thread();
 public Thread(Runnable target);

 void run(); // código concurrente

 void start(); // arrancar ejecucion

 /* suspender ejecución un tiempo */
 static native void sleep(long millis)
   throws InterruptedException;

 /* esperar que termine la hebra */
 void join() throws InterruptedException;

 /* interrumpir la ejecución de la hebra */
 public void interrupt()
   throws SecurityException;
 ...}
```

Vistas las composiciones de la interfaz y la clase anteriores, hay varias formas de crear hebras utilizando estas:

1. **Extendiendo la clase** `Thread` y redefiniendo el método `run()`, que contendrá el código que se ejecuta concurrentemente con otras hebras. Aparte, habrá que arrancar la ejecución de cada hebra para que esta empiece a ejecutarse mediante el método `start()`.

   ```
   class Tarea extends Thread {
    @Override
    public void run() {...}        // código concurrente
   }
   Tarea t = new Tarea();
   t.start();
   ```

2. **Implementando la interfaz** `Runnable` y, nuevamente, redefiniendo el método `run()`. Ha de quedar claro que siempre se va a arrancar la ejecución de un objeto de la clase `Thread` por lo que en este caso se va a utilizar el segundo constructor de dicha clase(`public Thread(Runnable target)`).

   ```
   class Tarea implements Runnable {
    @Override
    public void run() {...}        // código concurrente
   }
   Tarea t = new Tarea();
   Thread h = new Thread(t)
   h.start();
   ```

3. **Mediante un objeto `Runnable` interno**: En este caso el método `run()` queda oculto y no se puede invocar desde fuera.

```
public static void main(String[] args){
 Runable r = new Runnable(){
  public void run(){...}        // código concurrente
 };
 new Thread(r).start();
}
```

4. **Haciendo el método `run()` interno con `Thread`**: Similar al caso anterior pero utilizando la clase `Thread`. El método `run()` queda oculto nuevamente.

```
public static void main(String[] args){
 Thread h = new Thread(){
  public void run(){...}        // código concurrente
 };
 h.start();
}
```

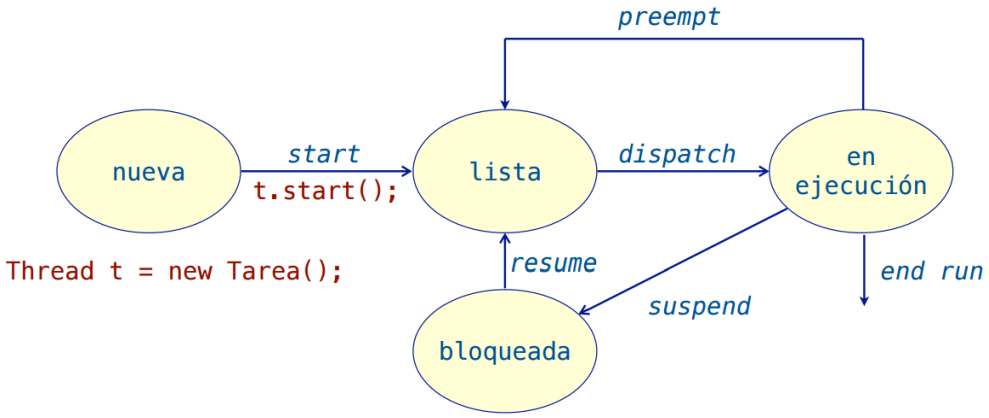
Cabe destacar que las opciones más utilizadas son las dos primeras.

El **número de hebras** existentes en un programa serán: Una **hebra inicial** que ejecuta el método `main()` y **todas las que se arranquen** en el programa con `start()`. Además la máquina virtual y la interfaz gráfica pueden crear hebras adicionales.

Una hebra **termina** cuando termina el método `run` o cuando se fuerce mediante una interrupción. Un **programa termina de forma satisfactoria** cuando termina el método `main()` y todas las demás hebras que hayan arrancado en el programa. El programa **termina** también, aunque de forma **forzada**, si se hace `exit()` desde alguna hebra o si se lanza una excepción que se propaga fuera de `run()` o `main()`.

Estados de las hebras

Más adelante se acabará de entender por qué es necesario definir estados para las hebras, aunque podría empezarse a pensar en los criterios que tiene un sistema monoprocesador para multiplexar en el tiempo las actividades de las hebras. Este pondrá en ejecución una hebra durante un intervalo de tiempo y la bloqueara durante otro para que el resto de hebras puedan continuar con su ejecución. El siguiente esquema muestra un diagrama de estados de las hebras:



Nota: Una tarea que ha terminado no se puede volver a arrancar.

Continuación del ejemplo anterior

Ahora podemos dar un paso más allá con el ejemplo planteado en el punto anterior. De esta forma, las hebras quedarían:

La hebra `EscribeHora`:

```
import java.util.*;
public class EscribeHora
     extends Thread {
 @Override
 public void run() {
  try {
   while (true) {
     sleep(1000); // esperar 1000 ms
     System.out.println
       (new Date().toString());
   }
  } catch (InterruptedException e) {
   return; // terminar esta hebra
  }
 }
}
```

Y la hebra `Sonido`:

```
import java.awt.Toolkit;
import java.util.Scanner;
public class Sonido extends Thread {
 @Override
  public void run() {
   Scanner sc = new Scanner(System.in);
   while(true) {
    sc.nextLine();
    Toolkit.getDefaultToolkit().beep();
   }
  }
}
```

Finalmente habrá que crear un programa que arranque ambas hebras:

```
public class Reloj {
 public static void main(String[] args) {
   Hora hora = new Hora () ;
   Sonido sonido = new Sonido();
   hora.start();
   sonido.start();
 }
}
```

Interrupciones

A veces durante la ejecución de un programa una hebra necesita interrumpir a otra. Para ello se utiliza el método `interrupt()`, que generará diferentes comportamientos en función del estado de la hebra que se quiere interrumpir:

- Si la hebra está **bloqueada** en un `wait`,`join` o `sleep` se lanzará la excepción `InterruptedException`. Por lo tanto la hebra que quiere interrumpir a la otra deberá incluir un bloque `try...catch` o indicar que el método puede lanzar InterruptedException.
- Si la hebra está **bloqueada** en **otras operaciones**, la excepción que se lanza será distinta.
- Si la hebra **no está bloqueada**, entonces sigue funcionando normalmente, aunque un indicador revela si hay alguna interrupción pendiente.

Cabe destacar que lanzar una interrupción no es una buena forma de detener la ejecución, es preferible implementar un método que se encargue de detener la hebra y que será al que llame la hebra que quiere detener la ejecución.

3.3. Datos compartidos y exclusión mutua

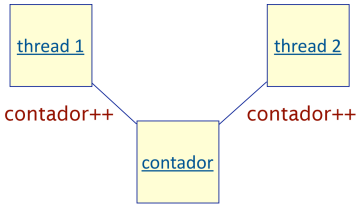
Datos compartidos

Las hebras de un programa pueden ser **independientes**, que la ejecución de una hebra no afecte a lo que hagan las demás, pero a menudo es más interesante que varias hebras **cooperen** para realizar una función común.

Un **mecanismo de cooperación** muy corriente consiste en el uso de **variables compartidas**, es decir variables a las que tienen acceso varias hebras y que pueden consultar el valor de la variable en un determinado momento o modificarlo.

Ejemplo

Vamos a suponer que tenemos un contador formado por dos hebras que incrementan concurrentemente el valor de este:



En un principio parece que el programa cumplirá su función de forma correcta, pero si se mira un poco más a fondo se descubrirá que el programa no cumple de forma adecuada con su función ya que el resultado dependerá de en qué orden se entrelace la ejecución de las operaciones de cada hebra. A esto se le llama **condición de carrera**.

Operaciones atómicas

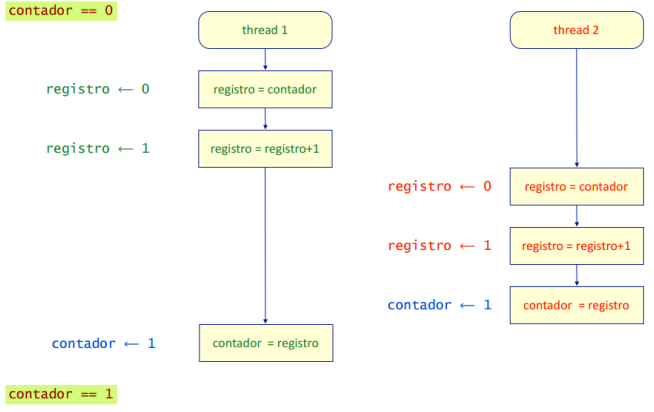
Siguiendo con el ejemplo anterior; la operación `contador++` no se ejecuta de una sola vez, se descompone en:

```
registro = contador  // registro es una variable temporal local de cada thread
registro++           // incrementar el valor del registro
contador = registro  // actualizar el valor del contador
```

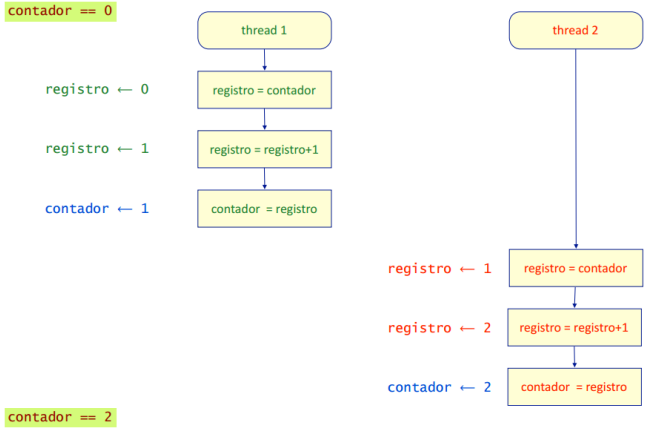
Las operaciones que no se pueden descomponer en otras se llaman **operaciones atómicas**. En este caso `contador++` no es atómica, pero las operaciones en las que se descompone sí que lo son.

Esto viene a decir que cuando varias hebras se **ejecutan concurrentemente**, se **entrelaza la ejecución** de sus instrucciones atómicas. Es en este momento en el cual se hace presente el problema que se planteaba en el ejemplo y por tanto, podrán darse las siguientes dos situaciones al ejecutar el programa:

Secuencia 1:



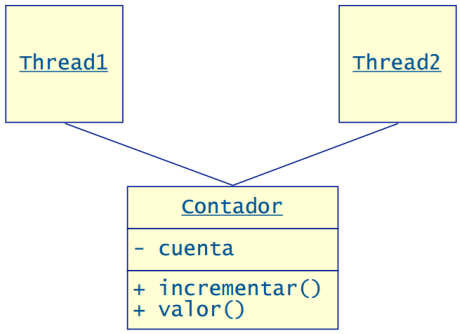
Secuencia 2:



Donde se aprecia que el resultado varía en función de las velocidades de ejecución relativas de las hebras. El resultado de un programa no debe depender de estos detalles ya que cada ejecución es distinta y tiene un comportamiento indeterminado con lo que es imposible hacer ensayos o pruebas.

Datos compartidos en objetos

Los campos de datos de objetos a los que se accede desde varias hebras también son datos compartidos y pueden dar lugar a condiciones de carrera independientemente de que se acceda a los datos directamente(en caso de que sean visibles) o mediante métodos.



Ejemplo: Contador

```
public class Contador {

 private long cuenta = 0; // estado
 public Contador (long valorInicial) {
   cuenta = valorInicial;
 }

 public void incrementar () {
   cuenta++; // modifica el estado
 }
 public long valor () { // devuelve el valor
   return cuenta;
 }
}
```

```
public class PruebaContador {
 private static class Incrementa
         extends Thread {
   Contador contador;

   public Incrementa (Contador c) {
    contador = c;
   }
   public void run () {
   ...
    contador.incrementar(); // región crítica
   ...
   }
 }
   public static void main(String[] args) {
    Contador contador = new Contador(0);
    Thread thread1 = new Incrementa(contador);
    Thread thread2 = new Incrementa(contador);
 /* las dos hebras comparten
           el objeto contador */
    thread1.start();
    thread2.start();
    ...
   }
}
```

Regiones críticas y exclusión mutua

Ya se ha visto el problema que existe cuando se tienen variables compartidas en un programa. Los segmentos de código en que se accede a variables compartidas se denominan **regiones críticas**.

Para evitar condiciones de carrera es necesario asegurar la **exclusión mutua** entre las hebras en las regiones críticas. Cuando una hebra está en una región crítica, ninguna otra puede acceder a los mismos datos con los que está trabajando la primera. Es decir; primero una hebra efectúa todas las operaciones de su región crítica mientras las demás esperan, luego otra pasa a realizar sus operaciones de región crítica y así sucesivamente. El orden entre ellas no importa, siempre que no se entrelacen las operaciones elementales.

De esta forma se consigue que las operaciones con variables compartidas sean **atómicas**.

Cerrojos

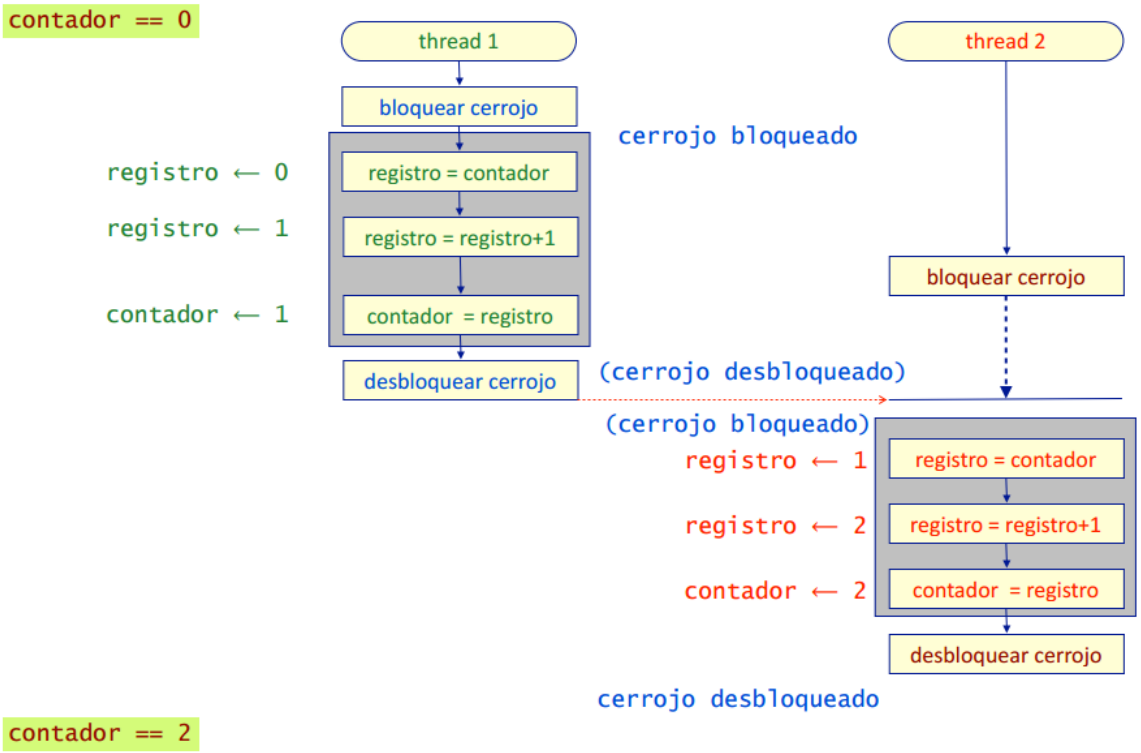
Para asegurar la exclusión mutua entre hebras se utiliza un mecanismo de sincronización llamado **cerrojo** o ***lock***. Los cerrojos tienen dos estados: **bloqueado** (cerrado) y **desbloqueado** (abierto) y dos operaciones atómicas:

- **Bloquear(*lock*)**: No permite que otras hebras accedan a la región crítica. Si el cerrojo estaba bloqueado, la hebra queda esperando a que se desbloquee.
- **Desbloquear(*unlock*)**: Permite al resto de hebras la posibilidad de acceder a la región crítica. Únicamente entrará una de ellas que bloqueará el cerrojo.

Para asegurar exclusión mutua debe ejecutarse en todas las hebras la siguiente secuencia:

1. `lock()` → **Bloquear cerrojo**
2. Realizar las **operaciones** en la **región crítica**
3. `unlock()` → **Desbloquear cerrojo**

De esta forma, siguiendo con el **ejemplo** anterior:



Aunque, los cerrojos son un mecanismo de bajo nivel que puede resultar complicado en programas complejos, siendo fácil equivocarse. Por lo que es recomendable integrar la exclusión mutua con objetos y usar métodos y bloques sincronizados en *Java*: *"Que trabaje el compilador, no el programador"*.

Monitores

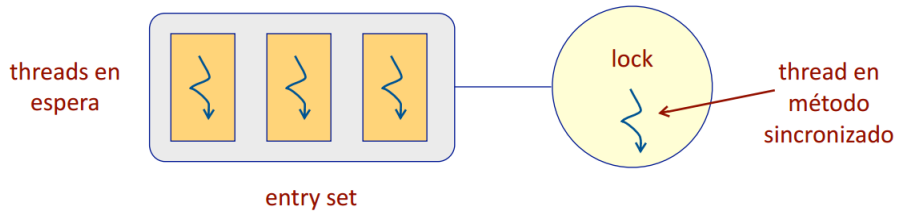
Java proporciona mecanismos de sincronización abstractos mediante los bloques y métodos sincronizados o `synchronized`; que definen partes del programa que se ejecutan en exclusión mutua. Son el compilador y la *JVM* los encargados de gestionar los cerrojos de forma implícita.

Un **método sincronizado** se ejecuta en exclusión mutua con los demás métodos sincronizados del mismo objeto y las **llamadas concurrentes** a métodos sincronizados se **ejecutan en exclusión mutua**

```
public class ContadorSincronizado {

 private long cuenta = 0; // estado
 public ContadorSincronizado
           (long valorInicial) {
  cuenta = valorInicial;
 }

 public synchronized void incrementar () {
  cuenta++;
 }
}
```

```
...
 public void run () {
  ...
  contador.incrementar(); // región crítica
  ...
 }
...
```

Cada objeto tiene asociado un cerrojo. Al ejecutar un método o bloque sincronizado, se bloquea el cerrojo. Si ya estaba bloqueado, la hebra que invoca el método se suspende y se queda esperando en una lista asociada al objeto(`entry set`). Al terminar la hebra, se desbloquea el cerrojo y si hay hebras esperando, se reanuda la ejecución de una de ellas. Es el propio compilador el que genera el cerrojo y los bloqueos y desbloqueos.



Es muy importante recordar que **sólo** se ejecutan en exclusión mutua los **métodos sincronizados**, si se olvida sincronizar algún método se pueden producir condiciones de carrera. Los métodos de clase(`static`) también se pueden sincronizar, ya que se usa un cerrojo para la clase y uno por cada objeto. Los **constructores no** se pueden **sincronizar** porque sólo se invocan al crear un objeto.

Un monitor es una clase que encapsula datos compartidos y operaciones sobre los mismos y deben cumplir:

- **Todos** los atributos han de ser `private`.
- **Todos** los métodos deben ser `synchronized`.

De esta forma se asegura el correcto acceso a datos compartidos. Una clase que puede usarse desde varias hebras sin problemas se dice que es segura *thread safe*.

Ejemplo: Dos formas de programar algo que aparentemente son iguales pero una de ellas puede dar lugar a condiciones de carrera:

Forma INCORRECTA:

```
public class Variable {
 private long valor = 0;
 public Variable (long valorInicial) {
   valor = valorInicial;
 }
 public synchronized void
              modificar(long v) {
   valor = v;
 }
 public synchronized long valor() {
   return valor;
 }
}
...
Variable v = new Variable(0);
...
long x = v.valor(); // ¡La región crítica es
x +=1;              // todo esto!!
v.modificar(x);     // ¡No está sincronizada!
...
```

Forma CORRECTA:

```
public class Variable {
 private long valor = 0;
 ...
 public synchronized void incrementar() {
   valor++;
 }
 ...
}
...
v.incrementar (); // Ahora está sincronizado
...
```

Los **bloques sincronizados** permiten definir regiones críticas mutuamente exclusivas con una granularidad menor.

```
synchronized(objeto) {
  instrucciones
}
```

Las instrucciones del cuerpo se ejecutan en exclusión mutua. Se utiliza el cerrojo asociado al objeto. Pero en general es más complejo de usar que los métodos sincronizados.

Datos volátiles

Algunos campos de datos se pueden declarar `volatile`, esto indica que el valor se puede cambiar desde otra hebra, por lo que asegura lecturas y escrituras atómicas en tipos primitivos.

Esto puede ser eficiente si lo único que se hace es cambiar el valor en una hebra y se lee en otra; por ejemplo: para parar la ejecución de una de ellas.

Es mejor no utilizarlo a no ser que se esté completamente seguro de que un dato puede ser volátil.

Ejercicio: Se propone realizar la *Parte 1* del ejercicio de *Gestión de un estacionamiento*

3.4. Sincronización condicional

Ya se ha visto el caso en el que las hebras acceden por turnos a regiones críticas para evitar que se produzcan condiciones de carrera. A veces una hebra tiene que esperar hasta que se cumpla una determinada **condición** para realizar sus operaciones; entonces **suspenderá** su ejecución hasta que se cumpla la condición. En un determinado momento, otra hebra hace que se cumpla y **avisa** a la que estaba esperando para que pueda reanudar la ejecución. Este tipo de sincronización se denomina **sincronización condicional**.

En el ejercicio propuesto de la *gestión del estacionamiento*, las hebras de la clase `Sensor` deberán avisar cuando entra o sale un coche. No deben entrar coches si el estacionamiento está completamente

lleno (`n >= capacidad`) y si un sensor llama a `entra()` cuando `n >= capacidad`, deberá suspender su ejecución hasta que haya sitio.

Espera y aviso

Espera condicional

El método `wait()` suspende la ejecución de la hebra que lo invoca y se usa para esperar una determinada condición:

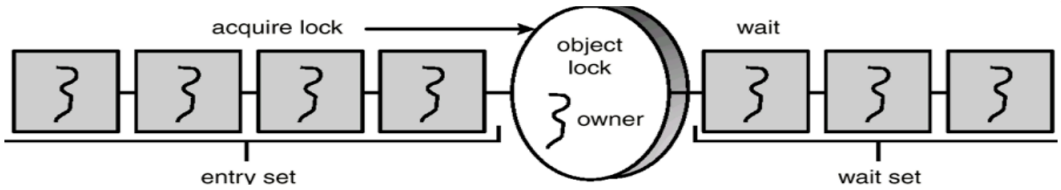
```
while (!condición)
 wait();
```

¡MUY IMPORTANTE! Siempre se pone **dentro de un bucle** y nunca en una sentencia condicional (`if...else`, `switch...case`, ...).

Únicamente se puede invocar dentro de un método o bloque sincronizados y al invocarlo se libera el cerrojo del objeto atómicamente. Puede lanzar una excepción `InterruptedException` y, en caso de que se lance, habrá que manejarla o propagarla.

Aviso de condición

Los métodos `notify()` y `notifyAll()` reanudan la ejecución de las hebras suspendidas por haber hecho `wait()` y se utilizan para avisar del cambio de estado de alguna condición. La diferencia se basa en que el primero reanuda la ejecución de **una** hebra **cualquiera** (aleatoria) mientras que el segundo reanuda la ejecución de **todas** las hebras suspendidas. Únicamente se puede usar desde métodos o bloques sincronizados. No se libera el cerrojo del objeto hasta que termina el método o sentencia sincronizada, la hebra o hebras que reanudan la ejecución pueden tener que *competir* con otras hebras,e incluso entre sí, para adquirir el cerrojo y poder continuar.



Siempre que **varias** hebras puedan estar esperando **condiciones distintas**, hay que usar `notifyAll()`. Sólo conviene hacer `notify()` si todas las hebras esperan la misma condición, ya que sólo una hebra puede avanzar cuando se cumple la condición. En caso de duda, utilizar siempre `notifyAll()`.

Ejercicio: Se propone realizar la *Parte 2* del ejercicio de *Gestión de un estacionamiento*

3.5. Fiabilidad

Propiedades de los programas concurrentes

Ya se han introducido estas cuatro propiedades fundamentales que el programador deberá asegurar en el programa concurrente, ahora se va a ampliar un poco el concepto de cada una:

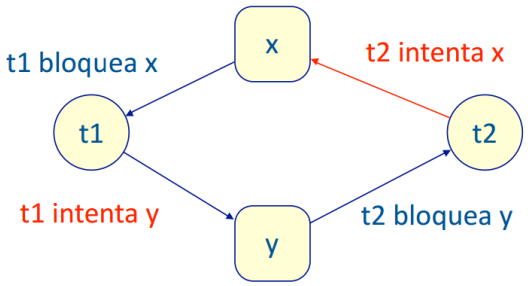
- **Correción**(*correctness*): El resultado siempre deberá ser correcto. Se puede usar *JUnit* para realizar pruebas unitarias sobre el código. Ha de tenerse en cuenta el problema del indeterminismo, es decir, los diferentes resultados que puede generar el programa en cada ejecución.
- **Seguridad**(*safety*): Nunca pasará nada malo durante la ejecución del programa y nunca se dará el caso de las condiciones de carrera.

- **Vivacidad**(*liveness*): En algún momento se hará lo que se tiene que hacer, aunque en instantes concretos haya hebras bloqueadas o *dormidas*.
- **Equidad**(*fairness*): Todas las hebras tienen la posibilidad de avanzar y ninguna perjudica al resto. Los recursos se reparten de forma justa. Habrá de tenerse en cuenta el problema de la inanición, que una hebra nunca accede a un recurso.

Interbloqueos

Los **interbloqueos** son situaciones en las que varias hebras están suspendidas esperando unas a otras sin que ninguna pueda progresar. Se pueden producir cuando se usan recursos de forma exclusiva y se asignan a distintas hebras, se dice entonces que hay **espera circular**. Vease en el siguiente ejemplo:

```
// thread 1 = t1
...
synchronized(x){ // acceso exclusivo a x
 synchronized(y){...} // acceso exclusivo a y
}
...
// thread 2 = t2
...
synchronized(y){ // acceso exclusivo a y
 synchronized(x){...} // acceso exclusivo a x
}
...
// también puede pasar con monitores
```

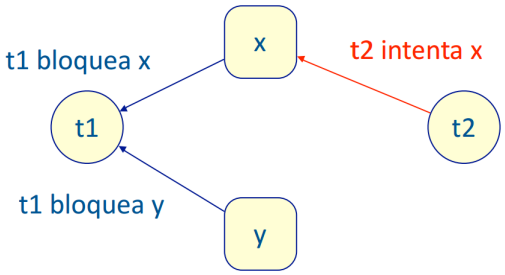


Los interbloqueos se pueden producir si se cumple alguna de estas condiciones necesarias, denominadas Condiciones de Coffman:

- Hay **exclusión mutua** en el uso de recursos
- Que las hebras puedan **tener** recursos **y mantenerlos** mientras esperan conseguir otros
- Que **no** se le pueda **quitar** un recurso a una hebra
- Hay **espera circular** como en el ejemplo anterior

Los interbloqueos no se pueden detectar mediante pruebas o test *JUnit* ya que ocurren de forma esporádica. Por ello para **prevenirlos** hay que **evitar** que se dé alguna de las anteriores **Condiciones de Coffman**. La forma más sencilla de prevenir la espera circular es acceder a los recursos **siempre en el mismo orden**. Siguiendo con el ejemplo anterior:

```
// thread 1 = t1
...
synchronized(x){ // acceso exclusivo a x
 synchronized(y){...} // acceso exclusivo a y
}
...
// thread 2 = t2
...
synchronized(x){ // acceso exclusivo a y
 synchronized(y){...} // acceso exclusivo a x
}
...
```



De esta forma, ya no se producen interbloqueos con los recursos `x` e `y`.

Ejercicio: Gestión de un estacionamiento

Se dispone de un aparcamiento con varias entradas y salidas, las cuales poseen un sensor que avisa cuando pasa un coche (tanto si sale como si entra). Además se dispone de un supervisor que muestra la ocupación del estacionamiento, es decir el número de coches que se encuentran en el interior.

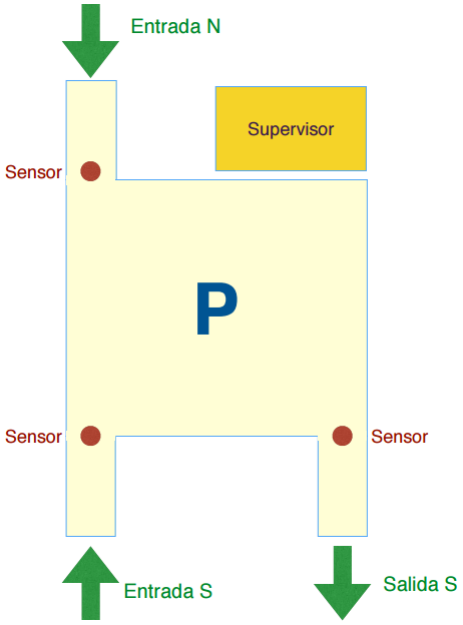


Figura 3.1: Aparcamiento P

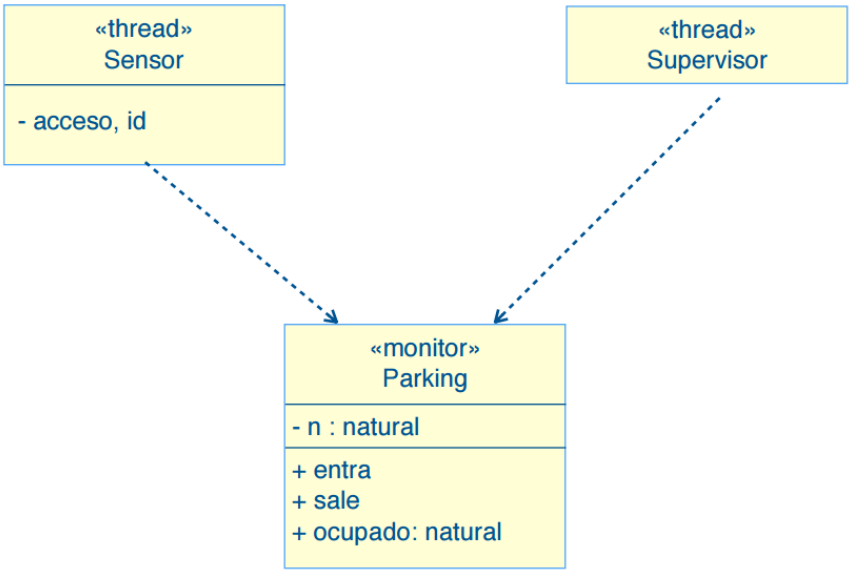
Se pide programar las clases `Sensor`, `Supervisor` y `Parking` teniendo en cuenta lo siguiente:

- La clase `Parking` será el monitor y contendrá 3 métodos:
  - Un método `entra(String puerta)` que aumenta una unidad el número de coches en el aparcamiento.
  - Un método `sale(String puerta)` que disminuye una unidad el número de coches en el aparcamiento.
  - Un método `ocupado()` que permita saber cuantos coches se encuentra en ese momento en el aparcamiento.
- La clase `Sensor` deberá encargarse de controlar los coches que entran y por donde entran.
- La clase `Supervisor` comprobará el estado del aparcamiento (número de plazas ocupadas) cada 10 segundos y deberá mostrarlo por pantalla.

En la **Parte 1**, no tener en cuenta la sincronización condicional, en la **Parte 2** incluir las sentencias necesarias para que haya sincronización condicional.

Solución Parte 1

Básicamente nos están pidiendo un programa con la siguiente estructura:



Las clases pedidas quedarían:

Parking

```
public class Parking {
   private int n = 0; // número de coches en el parking

  // entra un coche por una de las puertas
   public synchronized void entra (String puerta) {
     n++;
   }

  // sale un coche por una de las puertas
   public synchronized void sale (String puerta) {
     n--;
   }

  // consulta
   public synchronized int ocupado() {
     return n;
   }
}
```

Sensor

```
public class Sensor extends Thread {
   private String id;
   private Parking p;
   private Acceso acceso; // ENTRADA o SALIDA

   public Sensor(Parking p, String id, Acceso acceso) {
     this.p = p;
     this.id = id;
     this.acceso = acceso;
   }

   @Override
   public void run() {
     while (true) {
     ... // detecta un coche
       switch (acceso) {
         case ENTRADA: p.entra(id);
                       break;
         case SALIDA: p.sale(id);
                       break;
       }
     }
   }
}
```

Supervisor

```
public class Supervisor extends Thread {
   private Parking p;

   public Supervisor(Parking p) {
     this.p = p;
   }

   @Override
   public void run() {
     // comprueba el estado del parking cada 10 s
     while (true) {
       System.out.println("Número de plazas ocupadas: "
             + p.ocupado());
       try {
         sleep(10*1000);
       } catch (InterruptedException e) {
           System.err.println(e.toString());
       }
     }
   }
}
```

Solución Parte 2

La clase que habría que modificar sería el monitor `Parking`, las otras dos se mantendrían sin cambios.

```
public class Parking {
  private final int capacidad; // número de coches que caben
  private int n = 0; // número de coches que hay

  // constructor
  public Parking (int capacidad) {
    this.capacidad = capacidad;
  }

  // entra un coche por una de las puertas
  public synchronized void entra (String puerta) {
    while (n >= capacidad) wait();
    n++;
  }
  // sale un coche por una de las puertas
  public synchronized void sale (String puerta) {
   n--;
   notifyAll();
  }
  // consulta
  public synchronized int ocupado() {
    return n;
 }
}
```

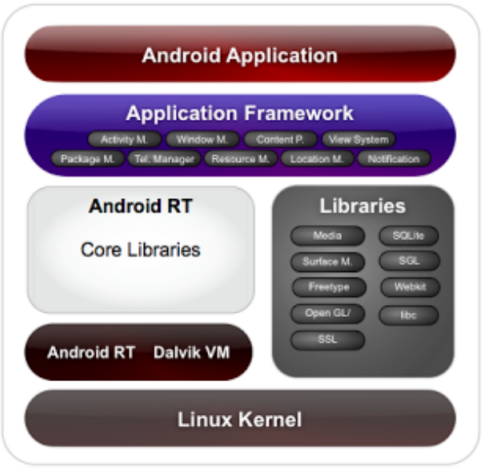
Capítulo 4

Interfaces y aplicaciones móviles

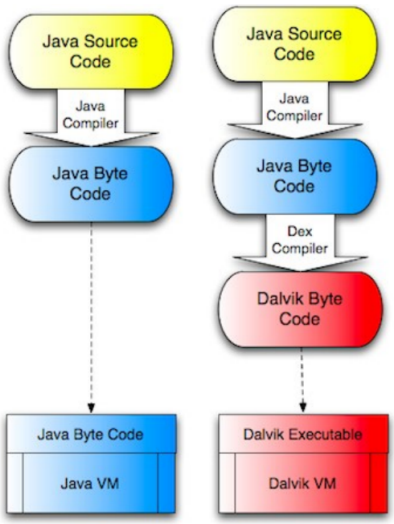
4.1. Introducción a Android

Android es una plataforma software basa en código abierto para dispositivos móviles. Posee ciertas restricciones a nivel de hardware como la baja capacidad de las baterías o de memoria y las velocidades de los procesadores. Es una plataforma diseñada para poderse transportar a múltiples dispositivos.

Las aplicaciones móviles se ejecutan bajo un sistema con la siguiente arquitectura:



Android posee una máquina basada en Java que contiene, a diferencia de este, una máquina virtual especializada: *Dalvik*, optimizada para usar poca memoria y permitir la ejecución simultánea de varias máquinas virtuales. Cada aplicación de Android tiene asignado un usuario en *Linux*, que tiene derechos sobre los ficheros de la aplicación. Las estructuras de las máquinas virtuales de Java y de Android son:



A diferencia del entorno PC, en la ejecución de aplicaciones no hay un método `main()` que arranca el programa. Cuando se ejecuta una aplicación en Android se activa un usuario *Linux*, se pone en

marcha el *runtime* y se llama a código del usuario para que se haga cargo de parte de las actividades. Una **actividad** es una pantalla, cuando se pasa a otra pantalla, la actividad de la primera se suspende y cuando se vuelve a la pantalla inicial, la activdad anterior se reanuda. Cuando una actividad lleva mucho tiempo suspendida, es el propio Android el que se encarga de pararla y destruirla.

El **API de Android** es un conjunto de bibliotecas para interactuar con la pantalla, con el sistema y con la memoria interna y externa. El API está continuamente actualizándose.

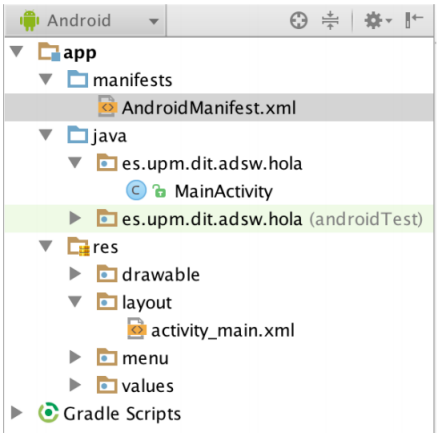
Organización de un proyecto Android

A la hora de elaborar una aplicación se deben distinguir tres partes importantes que la compondrán: el manifiesto, la parte funcional y la interfaz gráfica.

En el **manifiesto** se define la configuración de la aplicación(bloques, permisos) y está definido en XML en el archivo *AndroidManifest.xml*.

La **parte funcional** contiene lo que hará la aplicación, está situada en la carpeta "java" y ahí es donde se crean las actividades y las clases.

La **interfaz gráfica** se construye en la carpeta "res", que se compondrá de los diseños de pantalla(**layouts**), escritos en XML y de otros recursos como imágenes, menús, valores constantes...



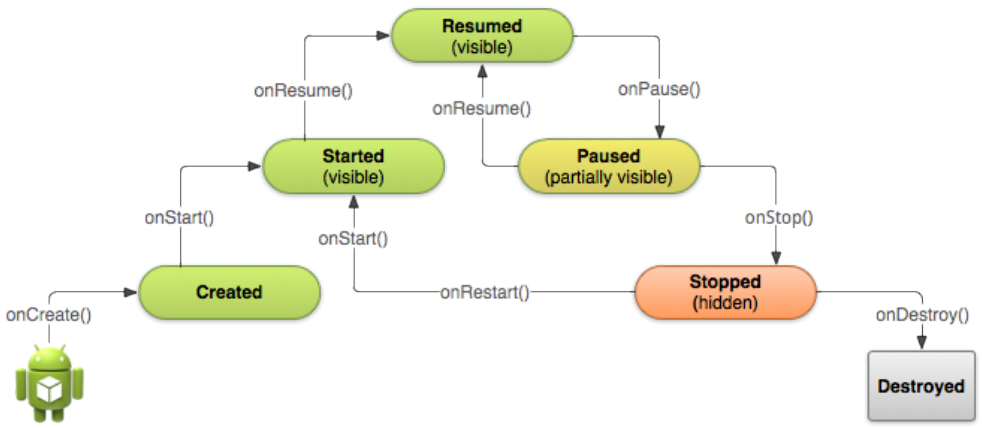
4.2. Desarrollo de aplicaciones en Android

4.2.1. Actividades

Como ya se ha visto, una actividad se corresponde con una pantalla de la aplicación, por tanto, una aplicación normalmente tendrá varias actividades. Al arrancar la aplicación se iniciará la actividad principal(*main activity*). Cada aplicación tiene dos aspectos:

- La parte **funcional** programada en Java, que extiende la clase `Activity` o alguna de sus clases derivadas.
- La parte **gráfica** especificada en xml y compuesta por elementos gráficos(*views*) y su disposición(*layout*), aunque también se pueden crear elementos gráficos programando en Java.

Cada actividad tiene un **ciclo de vida**, es decir, desde que se lanza hasta que se termina, la actividad estará transitará por diferentes estados.



Donde hay estados transitorios(Created, Started...) y tres estados básicos:

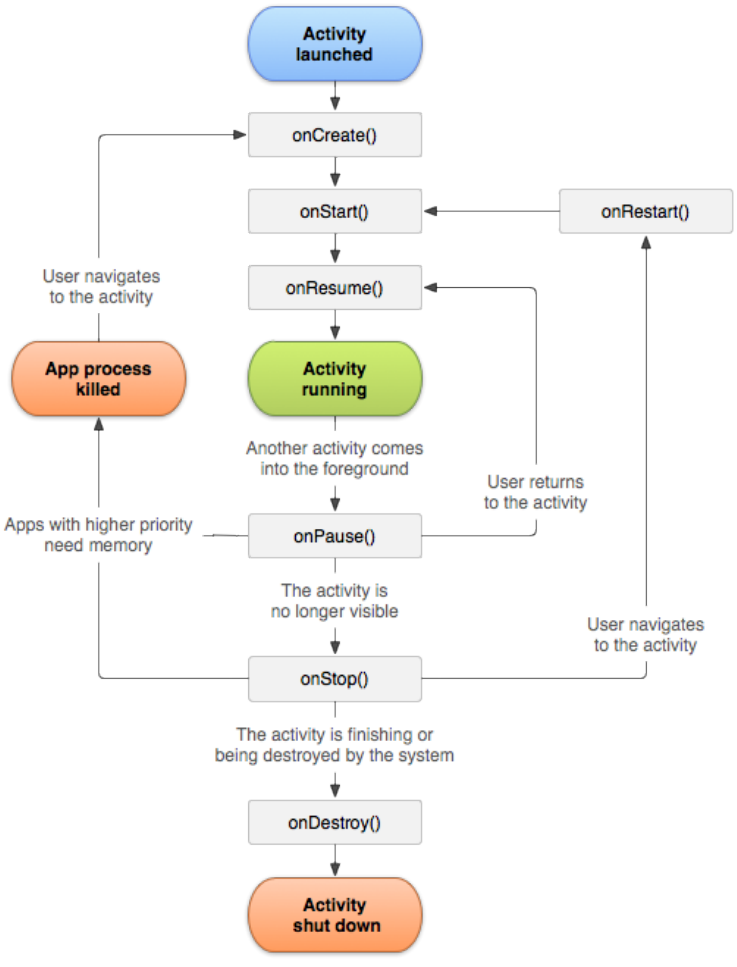
- **Reanudada o Resumed**: Actividad en primer plano con la que interactúa el usuario.
- **Pausada o Paused**: Actividad semi-visible en segundo plano, parcialmente tapada por otra actividad que está en primer plano que no ocupa toda la pantalla. La actividad pausada no permite interacción ni ejecuta código.
- **Parada o Stopped**: Actividad de fondo completamente oculta y que no ejecuta código.

Cuando una actividad no se está ejecutando, el sistema la deposita en una *pila de actividades*, donde la primera actividad de la pila es la última que se ha ejecutado, la segunda la penúltima y así sucesivamente. De esta forma, pueden volver a ser visibles y si es necesario se destruyen para liberar recursos.

En cada transición de estado se invoca un método de la clase correspondiente que define el comportamiento de la actividad, habiendo sido programado previamente.

```
public class MyActivity extends Activity {

  public void onCreate(Bundle bundle){
    super.onCreate(bundle); ...
  }

  protected void onStart() {
    super.onStart(); ...
  }

  protected void onResume() {
    super.onResume(); ...
  }

  protected void onPause() {
    super.onPause(); ...
  }

  protected void onStop() {
    super.onStop(); ...
  }

  protected void onDestroy() {
    super.onDestroy(); ...
  }
  ...
}
```

De esta forma, la actividad rotará entre los distintos estados al llamar a los métodos que conducen a cada uno de estos.



Funcionalidad de los métodos

Habiendo visto ya la estructura general básica de las actividades, se pasa a detallar qué es lo que debe realizar cada método de los anteriores.

onCreate(Bundle savedInstanceState)

Es el primer método que se llama al iniciar la actividad. Debe hacer dos llamadas, una al propio método de la clase base `super.onCreate(savedInstanceState)` para recuperar el estado anterior si fue destruida y otra a `setContentView()` para definir el layout de esta, además debe inicializar los componentes de la actividad. Es el único método que siempre se debe implementar. Por ejemplo una implementación muy simple:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
}
```

Un *Bundle* es un diccionario para almacenar valores, como los ya vistos en temas anteriores. Sirve para guardar el estado de una actividad y para pasar valores de una actividad a otra. Cada objeto tiene asignado una clave y tiene un método get (`TIPO getTIPO(String key)`) y un método put (`void putTIPO(String key, TIPO valor)`) para distintos tipos de objetos.

onStart()

Se ejecuta tras `onCreate(Bundle savedInstanceState)` justo cuando la actividad pasa a ser visible para el usuario y debe actualizar los recursos de la interfaz que va a ser visible. Le sigue `onResume()` si la actividad pasa a primer plano u `onStop()` si se oculta.

onResume()

Es llamado cuando la actividad pasa a primer plano o si el usuario reanuda la actividad que está pausada. Debe reanudar hebras y/o animaciones pausadas además de inicializar los componentes que se liberaron en `onPause()`. Siempre le sigue `onPause()`.

onPause()

Se llama cuando el sistema va a iniciar otra actividad, porque el usuario haya pulsado un determinado botón que cause el cambio de pantalla o porque la aplicación este programada para ello... . Debe parar animaciones u otras acciones que consuman CPU, liberar recursos que puedan afectar al consumo de la batería y guardar cambios de estado para hacerlos persistentes sólo si el usuario espera que se autoguarden.

Debe ser un método rápido para que podamos pasar rápidamente a la siguiente actividad o llamada telefónica o situación del terminal.

onStop()

Produce la parada de la actividad y hace que ya no sea visible. Puede ser llamado tras un `onPause()`. Debe realizar operaciones intensivas en CPU para guardar el estado, como podría ser escribir en la base de datos. El sistema puede llamar a `onDestroy()` si se necesita memoria o, en cambio, se puede llamar a `onRestart()` para relanzar la actividad.

onRestart()

Es llamado cuando una actividad parada se reinicia. Es poco habitual el reescribirlo. Debe reiniciar los recursos que fueron destruidos en `onStop()`. Siempre se llama desde este método a `onStart()`.

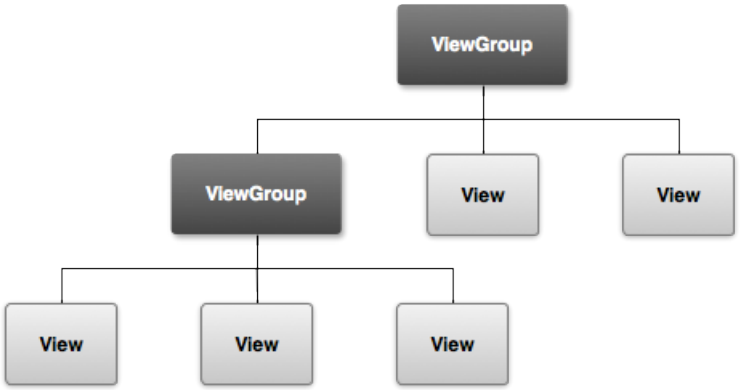
onDestroy()

Finalmente, es llamado para liberar el objeto de la actividad. Usualmente se habrá liberado previamente en `onStop()` y no será necesario su uso. Puede destruir la actividad el sistema o el usuario invocando `finish()`. Debe liberar los recursos que no se han liberado antes (por ejemplo las hebras) y guardar cambios en la configuración (por ejemplo si se ha cambiado la configuración de la actividad(orientación de la pantalla, teclado, idimo)) para reiniciar con esos cambios.

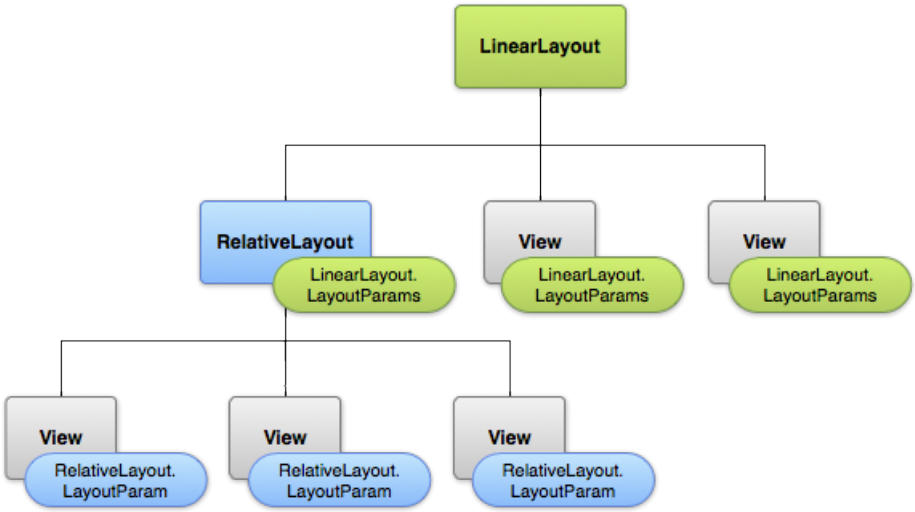
La actividad guarda el estado siempre antes de `onStop()` llamando a `onSaveInstanceState()` y tras `onCreate()` se llama a `onRestoreInstanceState()`

4.2.2. Interfaces: Layouts y vistas

La interfaz gráfica de una aplicación se especifica en xml y se compone de la siguiente estructura jerárquica formada por elementos gráficos(**views**) y grupos de estos(**layouts** ó ViewGroup):



Por ejemplo:



En este caso, *LinearLayout* es el principal y todos los demás se estructurarán en base a este.

Views

Las *Views* son elementos gráficos de una aplicación (textos, botones, cuadros de texto...) y ocupan un área rectangular de la pantalla. La clase `View` es la clase base de todos los objetos gráficos.

Cuando una actividad llama a `setContentView(View view)` le pasa como parámetro la raíz de una interfaz gráfica, que normalmente será un *ViewGroup* y esto hace que se vayan pintando todos los objetos.

Programar interfaces gráficas se puede hacer de dos formas:

1. De forma **declarativa** en XML, lo cual proporciona layouts y componentes estáticos. Por ejemplo:

```
<TextView
   android:id=\@+id/title"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello" />
```

2. De forma programática en Java, creando la interfaz de forma dinámica:

```
TextView title = new TextView(this);
title.setText("Hello");
setContentView(title);
```

La primera forma es la más sencilla y la mas usada. Existe una relación entre los ficheros XML y los ficheros Java de tal forma que a partir de estos últimos se puede acceder a los recursos declarados en los primeros mediante la clase `R` que se genera automáticamente a partir de los ficheros XML, por ejemplo:

```
TextView title = (TextView) findViewById(R.id.title);
```

Los objetos gráficos pueden tener **propiedades** o **atributos** que describan su aspecto o comportamiento. Pueden fijarse en XML o en Java, por ejemplo:

En XML: `android:text= "Hello"` En Java: `title.setText("Hello");`

Cada subclase de `View` tiene un conjunto de atributos: identidad, texto, color, tamaño, márgenes, relleno (*padding*), posición respecto a su contenedor, posición del contenido... Estos se administran mediante las siguientes expresiones y valores:

Expresión	Definición	Valores posibles
`orientation`	Orientación	horizontal, vertical
`layout_width`	Ancho	valor numérico expresado en dp, `fill_parent`, `wrap_content`
`layout_height`	Alto	valor numérico expresado en dp, `fill_parent`, `wrap_content`
`layout_marginX`	Espacio alrededor de la vista `X` = `Top`, `Bottom`, `Left`, `Right` Por ejemplo: `layout_marginTop`	
`layout_gravity`	Como se posicionan las vistas hijas en el contenedor	`top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, `fill`, `fill`, `clip_vertical`, `clip_horizontal`, `start`, `end`
`layout_weight`	Proporción del espacio disponible usado para la lista	1, 2, 3... (Por ejemplo: Tres vistas con pesos 1, 1, 2, ocuparían 1/5, 1/5, 2/5 del total
`layout_x` `layout_y`	Coordenadas x e y de la vista	

`gravity`	Posición del contenido del widget	Los mismos que `layout_gravity`
`id`	Identificador del widget	`@+id/nombre`
`text`	Texto del widget	`@string/nombre`

Las **medidas** pueden definirse mediante las dos siguientes formas:

1. **Tamaño real:** píxeles (px), pulgadas (in), milímetros (mm), puntos(1/72 de pulgada) (pt)
2. **Tamaño abstracto:** Unidades basadas en la densidad física de la pantalla.
   - *dp*: relativo a una pantalla de 160 dpi
   - *sp*: escalado a las preferencias de letra del usuario.

Los más utilizados son los dos úlimos, para que el tamaño de los elementos sea adaptable a múltiples dispositivos. Es recomendable utilizar *sp* para los textos y *dp* para el resto.

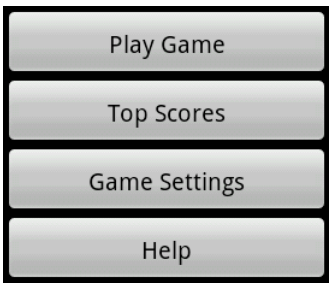
Layouts

Un **layout** es un fichero XML que describe el árbol de componentes (*View*) que representan una pantalla. Hay diversos tipos, algunos de los más comunes son:

- *LinearLayout*: Pinta las vistas según el atributo `orientation`, si es `horizontal`, de izquierda a derecha y si es `vertical`, de arriba a abajo.

`horizontal`

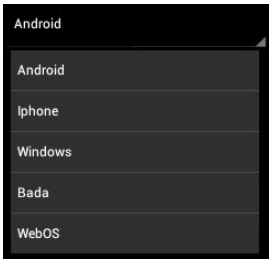
`vertical`



- *TableLayout*: Pinta las vistas en filas y columnas

- *RelativeLayout*: Permite indicar la posición de forma relativa a la vista padre o a otra vista.

En los Layouts se colocan **widgets** para que el usuario interactue con la aplicación, por ejemplo para representar un texto, seleccionar opciones, mostrar imágenes... Algunos de los más comunes son:

`TextView`	Mostrar un texto	
`EditText`	Campo donde el usuario introduce un texto	
`Button`	Botón para pulsar	
`Spinner`	Cuadro de selección	
`ImageView`	Mostrar una imagen. La imagen deberá estar en *res/drawable*	
`ProgressBar`	Barra de progreso de carga	

Un **botón** o ***Button*** puede contener texto, un icono(imagen), *ImageButton*, o ambos y comunica que acción se produce al pulsarlo. Asignar la conducta al botón se puede hacer de dos formas:

1. Definiendo el método que se deberá llamar al pulsar el botón en el XML de su layout mediante el atributo: `android:onClick="metodoALlamar"`, donde en *metodoALlamar* se incluirá únicamente el nombre del método sin paréntesis ni parámetros que reciba el método. Por ejemplo:

```
<Button android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/button1"
    android:textSize="20sp"
    android:id="+id/button1"
    android:onClick="getHora"></Button>
```

2. En Java, implementando la interfaz `OnClickListener` donde habrá de reescribirse el método `onClick(View v)`. El parámetro recibido *View* es el botón que ha sido pulsado y únicamente habrá un método *onClick* para toda la actividad. De esta forma se podrá distinguir que botón ha sido pulsado mediante el parámetro *View* y una sentencia condicional; pudiendo asignar así una funcionalidad distinta a cada botón ó incluso que todos los botones realicen la misma acción. Por ejemplo:

```
public class Activity1 extends Activity implements OnClicKListener{
  ...
  public void onClick(View v){
    if(v.getID() == R.id.button1){...
    }else if(v.getID() == R.id.button2 ){...
    }else if...
  }
}
```

A la hora de elegir una de las dos formas anteriores deberán tenerse en cuenta las ventajas e inconvenientes de cada una de ellas. La primera forma es más intuitiva, pero tiene el inconveniente de que siempre que se pulse el botón este va a tener un comportamiento invariable. En cambio con la segunda forma tenemos el inconveniente de que es más laborioso de programar, pero en cambio las posibilidades de asignación de funcionalidad a los botones son más extensas. Por ejemplo, se quiere diseñar un botón que en función de los datos introducidos en campos anteriores realice una suma o una resta, implementar esto de la forma en XML es sumamente complejo, en cambio hacerlo en Java es bastante más sencillo.

Toast

Un ***Toast*** es un mensaje instantáneo que aparece por pantalla en la actividad presente. Se declara en Java y permite definir su duración y el texto a presentar.


```
Context context = getApplicationContext();
CharSequence text = "Message saved as draft"
int duracion = Toast.LENGTH_SHORT;

Toast toast =
      Toast.makeText(context, text, duration);
toast.show();
```

Menús

Android tiene varios tipos de menús, los más utilizados son:

- Menú contextual que aparece cuando el usuario pulsa durante un tiempo en un elemento
- Menú desplegable modal
- Menú de opciones y barra de acciones, que aparece al pulsar menú. Para crearlo hay que utilizar un recurso *menu* en *res/menu* en XML aunque también se podría realizar con Java. La actividad debe implementar dos métodos:
  - → `onCreateOptionsMenu(Menu menu)` para leer e "insuflar" el fichero XML
  - → `onOptionsItemsSelected(MenuItem item)` para procesar que opción se ha pulsado y realizar la acción correspondiente

Por ejemplo:

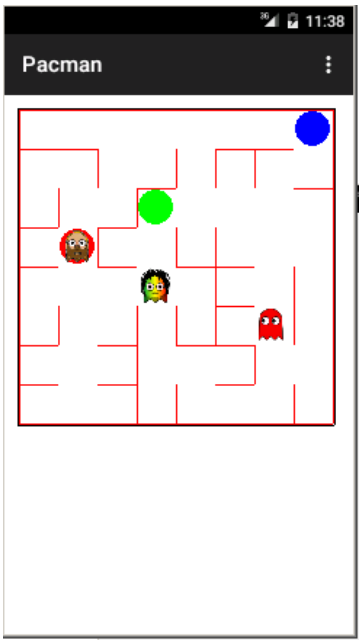
En XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android=
      "http://schemas.android.com/
               apk/res/android" >
  <item
    android:id="@+id/action1"
    android:menuCategory="container"
    android:titleCondensed="info"
    android:title=\@string/informacion" />
</menu>
```

En Java:

```
public boolean onCreateOptionsMenu(Menu menu) {
  getMenuInflater().inflate(R.menu.main, menu);
  return true;}
public boolean onOptionsItemSelected
                        (MenuItem item) {
  switch (item.getItemId()) {
    case R.id.action1:
      Toast.makeText(getApplicationContext(),
         "Info", Toast.LENGTH_SHORT).show();
      return true;
    default:
      return super.onOptionsItemSelected(item);
  }
}
```

Vistas personalizadas

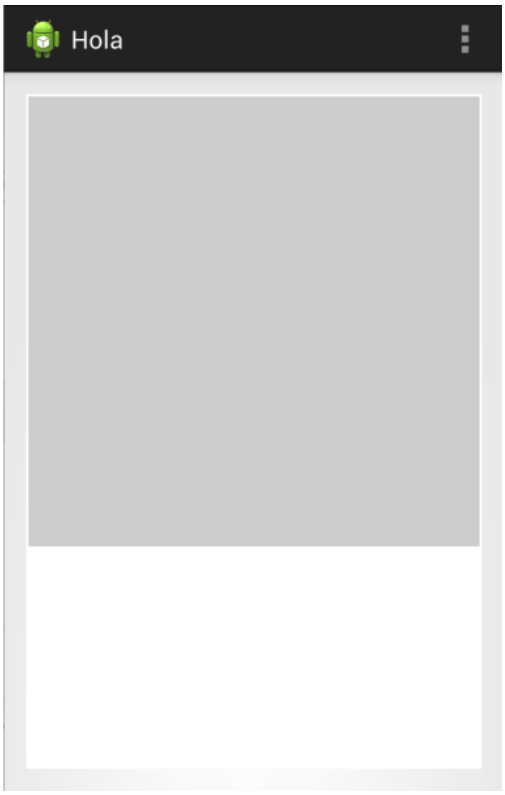


Android añade la funcionalidad de poder crear vistas personalizadas que no sigan el patrón común hasta ahora expuesto. Un posible ejemplo de este caso sería la aplicación mostrada en la imagen de la izquierda. Básicamente consiste dibujar sobre un lienzo el terreno deseado.

Las **vistas personalizadas** son, en realidad, subclases de *View*. Para su creación hay que extender, por tanto, la clase *View* y además implementar ciertos métodos:

- `onDraw(Canvas canvas)` Método llamado de forma automática para dibujar el contenido de la vista. `Canvas canvas` es el lienzo sobre el que se dibuja, habrá que implementar también métodos que se encarguen de pintar sobre dicho lienzo.
- `onTouchEvent(MotionEvent event)` Método llamado cuando se toca la pantalla.

Un ejemplo de posible vista personalizada:




```
public class MyView extends View {
  private static final int MARGEN = 5;
  private Paint paint = new Paint();

  // constructor obligatorio
  public MyView(Context context,
                  AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
  protected void onDraw(Canvas canvas) {
    int ancho = getWidth() - 2*MARGEN;
    int alto = getHeight() - 2*MARGEN;
    int lado = Math.min(ancho,alto);
    canvas.drawColor(Color.WHITE);
    paint.setColor(Color.LTGRAY);
    canvas.drawRect(MARGEN, MARGEN,
            MARGEN + lado, MARGEN + lado,
            paint);
  }
}
```

La vista personalizada se diseña en Java, pero también tiene su parte correspondiente en XML:

```
<es.upm.dit.adsw.hola.MyView
   android:id="@+id/myView"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:gravity="center" />
```

El objeto (*view*) donde se toca puede recibir varios tipos de eventos, que el usuario toque, mueva, levante, toque con dos dedos... Estos eventos son capturados por el método `onTouch()`, por tanto la actividad deberá registrar un **gestor de eventos**:

```
public class MyView extends View {
  // constructor - lo llama el entorno de ejecución
  public MyView(Context context, AttributeSet attrs) {
    super(context, attrs);
    setOnTouchListener(new MyOnTouchListener()); //El gestor es MyOnTouchListener
  }
}
```

El **gestor de eventos** es una clase que implementa la interfaz `OnTouchListener`. Normalmente se define de forma privada en la vista personalizada y debe redefinir el método `onTouch()`

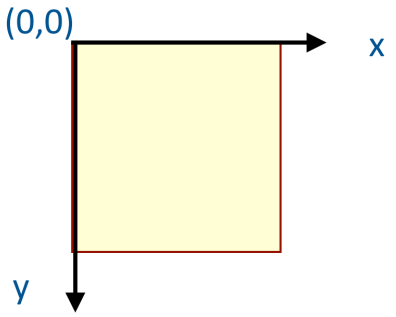
```
private class MyOnTouchListener implements OnTouchListener {
  public boolean onTouch(View v, MotionEvent event) {...}
}
```

Dicho método recibe dos parámetros, la vista en la que se ha pulsado, en este caso sería la vista personalizada y el evento (**MotionEvent**) que ha realizado el usuario en dicha vista. Los eventos más comunes que un usuario realiza son:

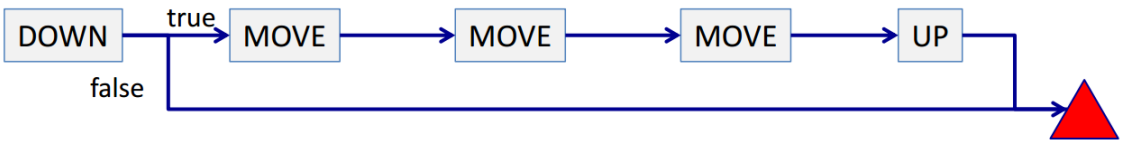
- `ACTION_DOWN` El usuario toca con el dedo
- `ACTION_MOVE` El usuario mueve el dedo sin levantarlo
- `ACTION_UP` El usuario levanta el dedo

De esta forma, se puede conocer que gesto ha realizado el usuario y con los métodos propios de MotionEvent, se puede saber también las coordenadas de la pantalla donde se ha realizado el evento (`float getX()`, `float getY()`), teniendo en cuenta que estos métodos tienen el origen de coordenadas en la esquina superior izquierda y devuelven las distancias en píxeles.

```
public boolean onTouch(View v, MotionEvent event) {
  switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN: { ...
      return true;
    }
    case MotionEvent.ACTION_MOVE: { ...
      return true;
    }
    case MotionEvent.ACTION_UP: { ...
      return true;
    }
  }
  return false;
}
```



Al realizar un evento, este deberá devolver `true` si se desea conocer más de el gesto o `false` si no se desea procesar más eventos relativos a la operación.

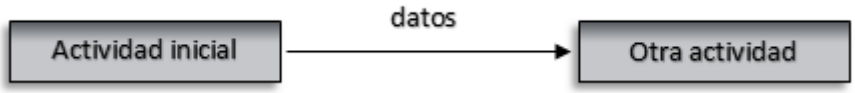


4.3. Comunicación entre actividades

Hasta ahora se ha visto como crear actividades que funcionan de forma autónoma, es decir, no requieren datos de otras actividades. En la gran mayoría de las aplicaciones las actividades se comunican entre sí enviando y recibiendo datos unas de otras. Existen dos mecanismos fundamentalmente importantes para compartir datos entre actividades, mediante intenciones y mediante un *singleton*.

Intenciones

Las **inteciones** o ***Intent*** además de pasar datos de una actividad a otra, sirven también para para cambiar de la actividad actual a la otra actividad a la que se le pasan los datos, es decir, para lanzar la otra actividad desde la inicial. Permiten también devolver un resultado a la actividad inicial al terminar. La intención se compone de dos pasos, creación y arranque.



Para simplemente **pasar** valores desde la actividad inicial y recogerlos en la otra actividad utilizando una intención:

```
// en la actividad inicial
  String datos = "ejemplo"
  Intent intent = new Intent(this, OtherActivity.class); //creación
  intent.putExtra("datos", datos); // crea un Bundle
  startActivity(intent); //arranque

//en la otra actividad para obtener lo que ha mandado la actividad inicial
  String datos = getIntent().getStringExtra("datos");
```

Si se quiere pasar datos desde la actividad inicial, que la otra actividad realice una acción con ellos y que cuando termine devuelva un **resultado** a la actividad inicial, el proceso es similar:

```
// en la actividad inicial
  Intent intent = new Intent(this, OtherActivity.class);
  startActivityForResult(intent, CODIGO); //CODIGO es un entero para distinguir peticiones

// en la otra actividad
  String datos = getIntent().getStringExtra("datos"); //se obtienen los datos pasados
  ...  //se realizan operaciones con los datos, incluso se pueden modificar
  Intent intent = new Intent(); // se crea una nueva intención
  intent.putExtra("datos", datos); // se introducen los datos modificados o no
  setResult(RESULT_OK, intent); //RESULT_OK es el código de resultado que se devuelve
  finish();
```

Por último la actividad inicial **recupera** el resultado y los datos que ha introducido la otra actividad:

```
//en la actividad inicial
  protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == CODIGO) {
      if (resultCode == RESULT_OK) {
        String datos = intent.getStringExtra("datos");
        ...
    }
  }
```

Android tiene predefinidas las llamadas **intenciones abstractas**, que sirven para solicitar tareas abstractas (como ver una página web o llamar a alguien ...). Es el propio sistema el que se encarga de buscar alguna aplicación que ejecute esas tareas. Para la implementación de este tipo de intenciones hay que pasar como parámetro del constructor de `Intent` la acción(`ACTION_VIEW`, `ACTION_CALL`...) correspondiente a realizar con el otro parámetro, que será la página web o el numero de teléfono... Algunas tareas pueden requerir que se declare en el manifiesto algún permiso para llevarse a cabo, por ejemplo la llamada por teléfono requiere uno de estos permisos.

```
String url = "http://www.google.com/";
Uri uri = Uri.parse(url);
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Singleton

Un *Singleton* es una clase de la que sólo existe un objeto, a la que todo el mundo puede acceder y compartir dicho objeto. Una forma de implementación podría ser la siguiente:

```
public class Singleton {
  //éste es el objeto único referido anteriormente
  private static final Singleton instance = new Singleton();

  // constructor privado, que impide que nadie mas genere objetos de esta clase
  private Singleton() {}

  // para obtener una referencia al objeto único de la clase
  public static Singleton getInstance() {return instance;}

  // en los atributos se guardan los datos a compartir
  // habría que declarar tantos como sean necesarios
  private String dato1;
  ...
  // setters & getter, tendrá que haber uno de cada por cada dato guardado
  public void setDato1(String s) {this.dato1 = s;}
  public String getDato1() {return dato1;}
}
```

La funcionalidad de un *Singleton* es simple, se tiene una clase (el propio *Singleton*) que guarda datos, mediante los *setters* y los *getters*, en los atributos que se hayan declarado. Al cambiar de actividad, los datos guardados continuarán intactos (siempre que no se haya programado otra clase que pueda modificarlos al mismo tiempo) en el *Singleton*, por tanto desde la otra actividad se podrán recuperar y modificar, nuevamente mediante los *getters* y *setters* respectivamente. Un ejemplo de uso:

```
// actividad inicial
  private Singleton singleton;
  ...
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_top);
    singleton = Singleton.getInstance();
    ...
  }
  ...
  singleton.setDato1("ejemplo");
```

```
// otra actividad
  private Singleton singleton;
  ...
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_top);
    singleton = Singleton.getInstance();
    ...
  }
  ...
  String dato = singleton.getDato1();
```

Ahora bien, ¿cuándo se usa uno u otro? Cabe recordar que son las **intenciones** las únicas que además de pasar datos, **lanzan** una nueva actividad. Para comunicar pueden utilizarse ambos, teniendo en cuenta lo siguiente:

- Las **intenciones** únicamente aceptan tipos primitivos y *String*
- Un ***Singleton*** acepta todo tipo de datos que se le quieran guardar (tipos primitivos, arrays, objetos de clases...), ya que es el programador el que declara los atributos que necesite. Aunque este mecanismo también acepta tipos primitivos y *String*, es preferible pasar este tipo de datos mediante intenciones, ya que facilitan el trabajo y no es necesario declarar clases auxiliares.

4.4. Bases de datos en Android

Una base de datos esta formada por una o varias tablas compuestas por columnas, que son los atributos de los datos, y filas o tuplas, donde se guardan los datos correspondientes a un elemento. Uno de los atributos de los datos sirve como clave, normalmente será la columna *id*.

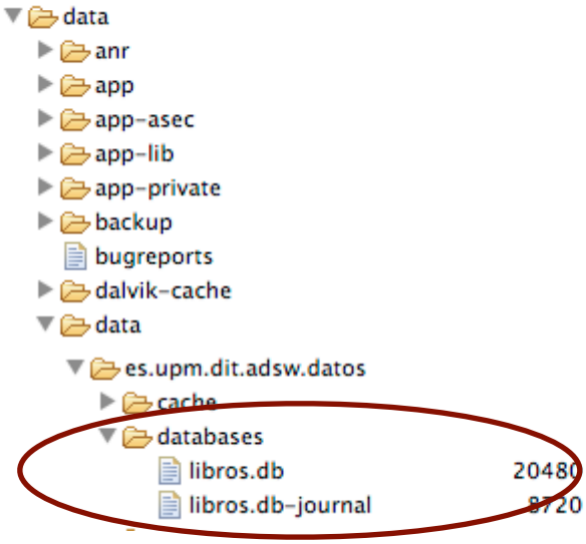
libros

ID	título	autor	año
1	El ingenioso hidalgo Don Quijote de la Mancha	Miguel de Cervantes	1605
2	Conversación en La Catedral	Mario Vargas Llosa	1969
3	La reina del Sur	Arturo Pérez Reverte	2002
4	A Most Wanted Man	John Le Carré	2008

Para el manejo de bases de datos se utiliza el lenguaje **SQL** (*Structured Query Language*), algunos de los comandos más utilizados son:

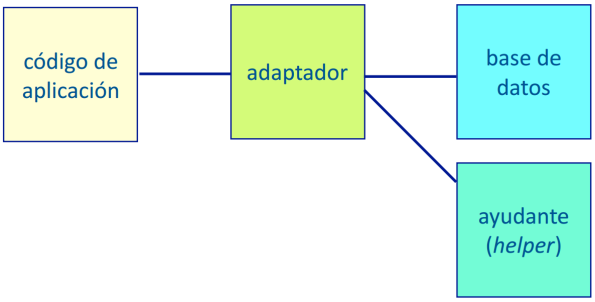
```
//para crear la tabla
create table libros (ID integer primary key autoincrement,título text, autor text,
                     año integer);

//para guardar datos en la tabla
insert into libros (autor, título, año) values('Javier Marías', 'Todas las Almas', 1988);

//para borrar datos de la tabla, en este caso todos los que cumplan una condición
delete from libros where año < '1900';

//para hacer una consulta en la tabla
select título, año from libros where author like '%Pérez%';
```

Aunque, en Android, se utiliza una adaptación de SQL llamada **SQLite**. Se trata de un sistema de gestión de bases de datos para sistemas empotrados, tiene ventajas como contener todas las tablas en un solo fichero y por tanto tener un tamaño reducido, además se trata de un sistema de código abierto. Un inconveniente podría ser que no implementa todo SQL, sin embargo es suficiente para muchas aplicaciones. SQLite se utiliza mucho en navegadores web y dispositivos móviles (Firefox, IOS, Android...).



En Android las bases de datos se almacenan en un fichero en la zona de datos de la aplicación y se pueden copiar al sistema de desarrollo.

Ahora bien, para poder utilizar bases de datos en una aplicación en Android de forma sencilla se utilizan un *adaptador* y un *ayudante* que facilitan las operaciones y además permiten el manejo de bases de datos sin tener grandes conocimientos del lenguaje SQL.



La **base de datos** esta **contenida** en `SQLiteDatabase`, que es la clase básica y fundamental de la estructura anterior. Esta clase posee métodos como:

- `void execSQL(String sql)` que sirve para ejecutar una sentencia SQL, pasada como parámetro, que **no** sea un `SELECT ...`.
- `Cursor rawQuery(String sql, String[] selectionArgs)` se utiliza para ejecutar una consulta mediante `SELECT ...`.

Como se puede apreciar, ambos métodos reciben como parámetro una sentencia SQL para ejecutar, lo cual requiere nuevamente conocer dicho lenguaje. Es precisamente por esto por lo que se utiliza el ayudante, para poder interactuar con la base de datos evitando ese inconveniente.

El **ayudante** es una clase auxiliar para gestionar la creación, el acceso y las versiones de una base de datos. Siempre extiende la clase `SQLiteOpenHelper` y redefine los métodos:

- `void onCreate(SQLiteDatabase db)` al cual se llama automáticamente al crear la base de datos.
- `void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` al cual nuevamente se le llama automáticamente al actualizar a una nueva versión.

El constructor del ayudante debe abrir la base de datos y crearla si no existe, para ello se llama al constructor de `SQLiteOpenHelper` pasándole como parámetro el `context`. Además el ayudante posee métodos como `getWritableDatabase()` ó `getReadableDatabase()` que devuelven un objeto `SQLiteDatabase`.

A continuación se muestra un ejemplo de adaptador que engloba todo lo visto hasta ahora en bases de datos e introduce nuevos conceptos. En general, todas las bases de datos que se construyan en Android tendrán la misma estructura y serán muy similares al ejemplo siguiente.

```
public class LibrosDbAdapter {

  //El adaptador debe tener acceso a la base de datos y al ayudante
  private SQLiteDatabase db;
  private DatabaseHelper dbHelper;

  private final Context context;

  //Por sencillez, se declara un conjunto de Strings como constantes que se
  //utilizan con frecuencia en el código
  private static final String DATABASE_NAME = "libros.db";
  private static final int DATABASE_VERSION = 1;
  private static final String TABLE_NAME = "libros";
  private static final String COL_ID = "_id";
  private static final String COL_TITULO = "titulo";
  private static final String COL_AUTOR = "autor";
  private static final String COL_AÑO = "año";

  //Esta es la sentencia de creación de la base de datos en SQL
  private static final String DATABASE_CREATE = "create table " + TABLE_NAME +
    " (" + COL_ID + " integer primary key autoincrement, " + COL_TITULO + " text
    not null, " + COL_AUTOR + " text not null, " + COL_AÑO + " integer not null);";

  //Clase del ayudante, esto siempre va a ser igual
  private static class DatabaseHelper extends SQLiteOpenHelper {

    DatabaseHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
      db.execSQL(DATABASE_CREATE);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
      onCreate(db);
    }
  }

  //Constructor del adaptador
  public LibrosDbAdapter(Context context) {
    this.context = context;
  }

  //Método para abrir la base de datos, deberá ser llamado al principio,
  // tras crear el adaptador. Este método es muy importante, ya que es el que
  //verdaderamente "crea" la base de datos.
```

```
public LibrosDbAdapter open () throws SQLException {
  dbHelper = new DatabaseHelper(context);
  db = dbHelper.getWritableDatabase();
  return this;
}

//Para cerrar la base de datos
public void close() {dbHelper.close();}

//Añadir un elemento a la base de datos
public long insert (String título, String autor, int año) {
  ContentValues values = new ContentValues();
  values.put(COL_TITULO, título);
  values.put(COL_AUTOR, autor);
  values.put(COL_AÑO, año);
  return db.insert(TABLE_NAME, null, values);
}

//Borrar el elemento de la base de datos cuyo parámetro id coincida con el pasado
// como parámetro, puede programarse para borrar con otro tipo de parámetros
// que no sea el id
public boolean delete (long id) {
  String[] whereArgs = {String.valueOf(id)};
  return db.delete(TABLE_NAME, COL_ID + "=?", whereArgs) > 0; //devuelve el nº
                                                              // de filas borradas
}    //Para borrar la base de datos entera db.delete(TABLE_NAME, null, null);

//Modificar o actualizar la información correspondiente a un elemento de la
// base de datos
public boolean update(long id, String título, String author, int año) {
  ContentValues values = new ContentValues();
  values.put(COL_TITULO, título);
  values.put(COL_AUTOR, autor);
  values.put(COL_AÑO, año);
  String[] whereArguments = {String.valueOf(id)};
  return db.update(TABLE_NAME, values, COL_ID + "=?", whereArgs) > 0;
}

//Método para realizar consultas en la base de datos, en este caso el método
// busca por el autor pasado como parámetro. Se podría programar para que
// busque por el título, el año...
public Cursor selectAutor (String autor) {
  String[] columns = {COL_TITULO, COL_AUTOR}; //Columnas que se quiere que devuelvan,
                                              // null si todas
  String selection = COL_AUTOR + "=?"; //Filas que se quiere que se devuelva,
  String[] selectionArgs = {autor};    // null si todas
  String groupBy = null; //Esto siempre null
  String having = null;  //Esto siempre null
  String orderBy = COL_AÑO; //Si se quiere algún orden en las filas,
                            // null si no se quiere orden específico
  return db.query(TABLE_NAME, columns, selection, selectionArgs,
                        groupBy, having, orderBy);
}
```

Al realizar una consulta en la base de datos, en el ejemplo anterior mediante el método `selectAutor()`, se recibe un **Cursor**. Este elemento se trata de un contenedor de la información que se ha solicitado en la búsqueda. El contenedor posee un apuntador, que se puede mover por la tabla devuelta en la consulta para extraer la información de las filas y columnas que se precise.

Inicialmente, al recibir el cursor con la información, este apunta justo antes de la primera fila. Para navegar por la tabla se utilizan una serie de métodos propios de la clase Cursor:

- `int getColumnIndex(String columnName)` Devuelve el índice de una columna en la tabla de resultados (recordar que se numeran empezando por 0), devuelve -1 en caso de que no exista la columna.
- `boolean moveToFirst()` Mueve el cursor a la primera fila, devuelve `false` si no hay filas. Lo normal es llamar a este método tras recibir el Cursor para empezar a extraer información y evitando así que salten excepciones por acceso a posiciones inexistentes.
- `boolean moveToNext()` Avanza el cursor a la siguiente fila, devuelve `false` si no hay más filas.
- `tipo getTipo(int columnIndex)` Devuelve la información de una columna en la fila actual. Donde tipo habrá que sustituir por: `int`, `long`, `float`, `String`...

Para mostrar una **consulta en pantalla**, o parte de la base de datos, o la base de datos entera se utiliza una `ListActivity` y dos *layouts*:

- Uno para toda la pantalla, que incluye una ListView, cuando la consulta devuelve resultados, y una TextView, que se muestra cuando no hay resultados. Un ejemplo podría ser:

  `activity.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width ="match_parent"
   android:layout_height="match_parent">

   <ListView                          //Se muestra cuando hay resultados
      android:id="@+id/android:list"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>

   <TextView                          //Se muestra cuando no hay resultados
      android:id="@+id/android:empty"
      android:text="No hay datos"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"/>
</LinearLayout>
```

- Otro para las filas del cursor, típicamente se trata de un LinearLayout horizontal con una TextView para cada columna. Un ejemplo con dos columnas sería:

  row.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="horizontal">

   <TextView
      android:id="@+id/text1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>

   <TextView
      android:id="@+id/text2"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>
</LinearLayout>
```

  La actividad se enlaza con los *layouts* anteriores mediante un adaptador:

```
SimpleCursorAdapter(Context context, int layout, Cursor c, String[] from,
                                    int[] to, int flags)
```

  Por último se le pasaría el adaptador a la actividad:

```
void setListAdapter(ListAdapter adapter)
```

Para terminar de comprender esto último, se muestra a continuación un ejemplo de un método que proyecta el contenido del cursor en la interfaz gráfica:

```
private void showList(Cursor cursor) {
  startManagingCursor(cursor);

  String[] from = new String[]{LibrosDbAdapter.COL_TITULO, LibrosDbAdapter.COL_AUTOR};
  int[] to = new int[] {R.id.text1, R.id.text2};

  SimpleCursorAdapter adapter = new SimpleCursorAdapter(this, R.layout.row,
                                                        cursor, from, to);

  setListAdapter(adapter);
}
```

Este método **NO** está contenido en el adaptador de la base de datos, es por eso que se utilizan las referencias `LibrosDbAdapter.COL_TITULO, LibrosDbAdapter.COL_AUTOR` y no `COL_TITULO, COL_AUTOR`.

4.5. Concurrencia en Android

En Android las actividades se ejecutan en una hebra que gestiona la interfaz gráfica (*UI thread*), que captura eventos y dibuja en la pantalla. Si se ejecuta una acción que tarda mucho tiempo, la actividad no responde y lanza un mensaje de error. Es por esto que conviene ejecutar las tareas largas en hebras independientes de esta.

Igual que en Java, las hebras son clases que extienden *Thread* o implementan *Runnable*

El proceso a seguir es el mismo:

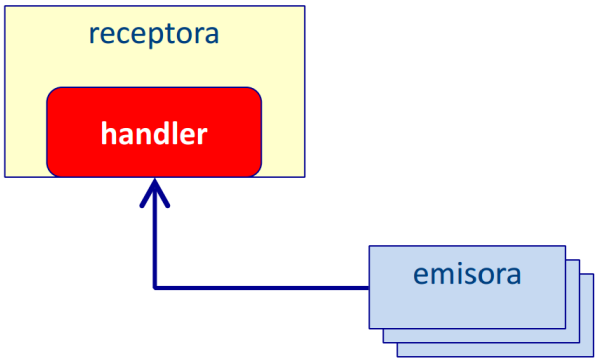
1. Se crea el objeto *Thread*
2. Se arranca con `start()`
3. Se ejecuta el método `run()`
4. Cuando termina el método `run()`, la hebra termina y no se puede volver a arrancar. Hay que tener en cuenta que si la aplicación termina, las hebras no deben seguir ejecutándose.

```
private void operacionLarga() {
  new Thread(new Runnable() {
    public void run() {
      String resultado = ...;
    };
  }).start();
}
```

Ahora bien, tras haber realizado una operación larga en otra hebra independiente, está deberá devolver el resultado. Para ello hay que tener en cuenta que no se puede acceder directamente a la interfaz gráfica (*UI thread*) ya que no es *thread-safe* y tampoco se puede bloquear, con lo cual no se pueden usar monitores. Debido a esto existen mecanismos que facilitan este tipo de tareas.

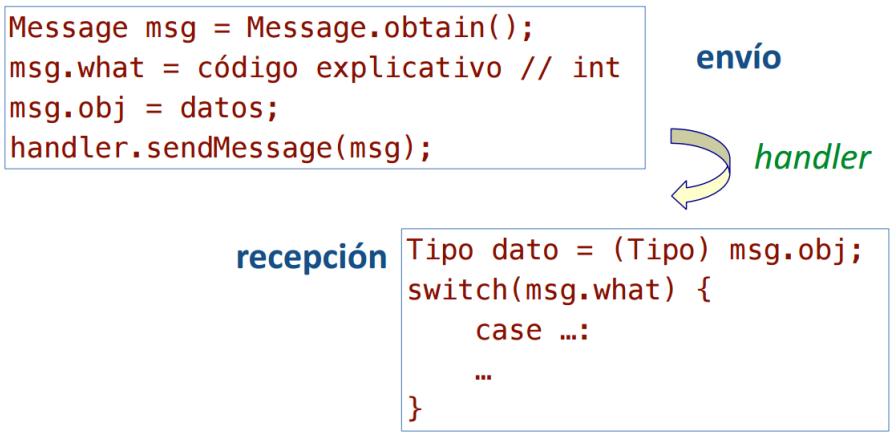
Handler

El ***Handler*** es un mecanismo para la comunicación entre hebras. Esta asociado a una hebra receptora para que el resto de hebras puedan enviarle cosas. Podría asimilarse a un buzón que posee la hebra receptora en el que las hebras emisoras dejan la correspondencia para la anterior.



El *Handler* permite enviar mensajes (`Message`) y código, objetos *Runnable* que se ejecutan en la hebra receptora, así no hay problemas de sincronización. Además, el *Handler* mantiene una cola de peticiones pendientes.

El **envío de mensajes** se hace con un objeto de la clase `Message` que consiste en un envoltorio para enviar objetos de otras clases, además se envía un código explicativo del contenido del mensaje. En realidad lo que se manda es una referencia, no una copia del objeto.



Para enviar un `Message`, lo primero que se hace es obtenerlo mediante `obtain()`, **nunca se debe crear un objeto** `Message`; en el `msg.what` se indica el código explicativo que se trata de un número entero. En la recepción, se deberá forzar la conversión del `msg.what` al tipo o clase del objeto que se haya enviado y podrá realizarse un procedimiento distinto en función del código explicativo que se haya recibido (`switch(msg.what){...}`).

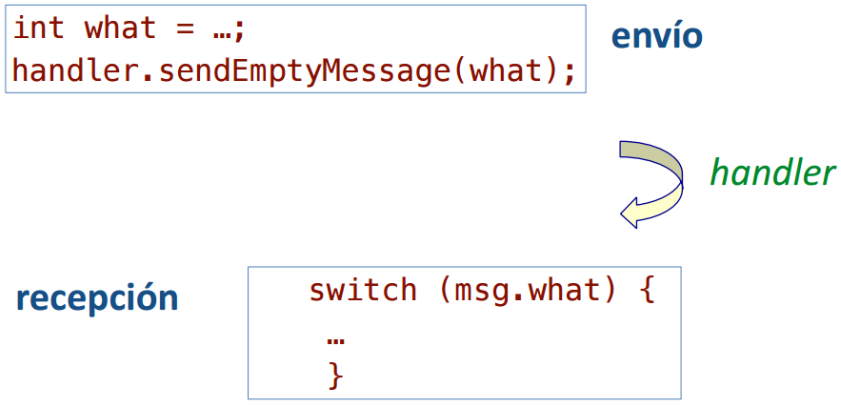
Un ejemplo de implementación de este mecanismo:

```
public class MainActivity extends Activity {

  public static final int DATOS = 0;
  private Worker worker;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    Handler handler = new MyHandler();
    worker = new Worker(handler);
    worker.start();
  }

  private class MyHandler extends Handler {

    @Override
    public void handleMessage(Message message) {
      switch (message.what) {
        case DATOS:
        String datos = (String) message.obj;
        view.Set...(datos);
        break;
      }
    }
  }
}
```

Y la hebra encargada de realizar las operaciones:

```
public class Worker extends Thread {

  private final Handler handler; //Para que la hebra sepa donde mandar el resultado

  public Worker(Handler handler) {
    this.handler = handler;
  }

  public void run() {
    ...
    String resultado = ...;
    int what = DATOS;
    Message message = Message.obtain(handler, what, datos);
    handler.sendMessage(message);
    ...
  }
}
```

También se pueden mandar **mensajes sin datos**, enviando únicamente el un código explicativo. En este caso el esquema de actuación será:



Para el **envío de código** se utiliza el método post en el handler, pasando un elemento *Runnable* como argumento. El código enviado se ejecutaá en la *UIThread*.

```
handler.post(new Runnable() {
  @Override
  public void run() {
    display.setText(datos);
  }
});
```

Android impone la limitación de que no se puede cambiar el entorno gráfico desde una hebra cualquiera, siendo la *UIThread* la única capaz de hacerlo.

Forma INCORRECTA:

```
public void onClick(View v) {
  Thread thread = new Thread(new Runnable() {
    public void run() {
      Bitmap b = loadImageFromNetwork("http...");
      imageView.setImageBitmap(b);
    }
  });
  thread.start();
}
```

Forma CORRECTA:

```
public void onClick(View v) {
  Thread thread = new Thread(new Runnable() {
    public void run() {
      Bitmap b = loadImageFromNetwork("...");
      imageView.post(new Runnable() {
        public void run() {
          imageView.setImageBitmap(b);
        }
      });
    }
  });
  thread.start();
}
```

Tras haber actualizado el entorno gráfico, como en el ejemplo anterior, se debe refrescar la pantalla, para ello se dispone de dos métodos principales:

Mientras se ejecuta la hebra principal:

- `view.invalidate();`

Cuando se esta ejecutando otra hebra:

- `view.postInvalidate();`

AsyncTask

La clase ***AsyncTask*** facilita la ejecución de código en segundo plano (*background*), ya que el uso de hebras directamente es complicado. Este mecanismo consiste en una tarea asíncrona con métodos que se ejecutan en dos hebras distintas:

- Una hebra de fondo separada (***background thread***), para acciones que necesitan tiempo, es decir, como se ha visto antes, para operaciones largas y/o complejas.
- La hebra principal (*UI*) para publicar los resultados.

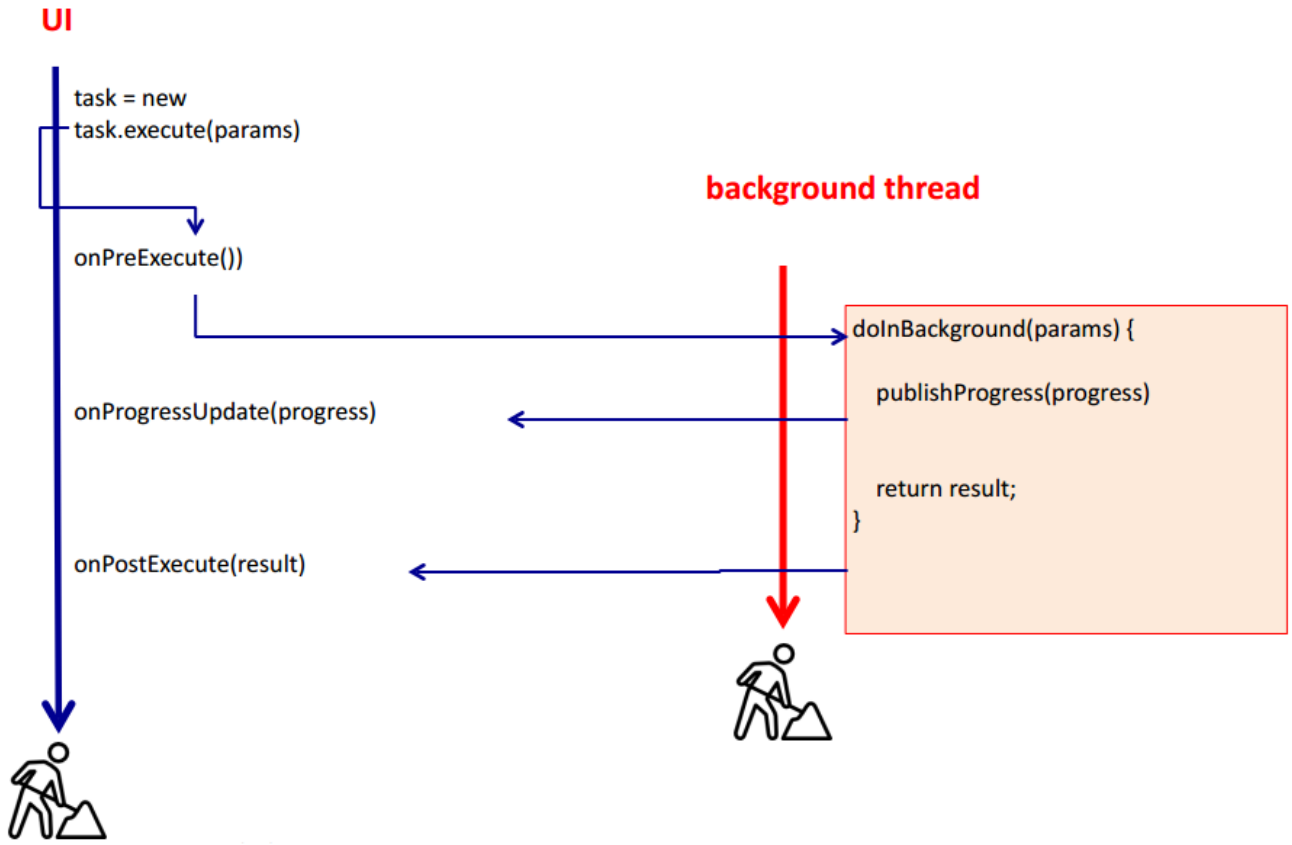
De esta forma se puede asegurar que los métodos se ejecutan en la hebra apropiada.

La clase *AsyncTask* tiene una serie de métodos predefinidos que Android llama y ejecuta automáticamente cada uno en la hebra correspondiente. Dos de esos métodos permiten controlar y mostrar por pantalla el progreso de la tarea que se está realizando en la hebra de fondo, así el usuario podrá conocer que porcentaje de la tarea queda por realizar.

```
//Los tres elementos que se encuentran entre <...> se refieren cada uno:
// Params = Tipo o clase de parámetros que recibe la hebra de fondo
// Progress = De que forma se va a facilitar el progreso (Integer...)
// Result = Tipo o clase de parámetros que recibe la hebra principal como resultado
//           de la operación que ha realizado la hebra de fondo.

class MyClass extends AsyncTask<Params, Progress, Result> {

  // métodos que se ejecutan en la hebra principal (UI)
  void onPreExecute(){...} // inicio
  void onProgressUpdate(Progress ...)
  void onPostExecute(Result)
```

```
// métodos que se ejecutan en la hebra de fondo
Result doInBackground(Params ...) // similar a run()

// método auxiliar
void publishProgress(Progress ...)
```

De esta forma, el esquema de ejecución de este mecanismo sería:



Finalmente, este mecanismo tiene una serie de reglas para su correcto funcionamiento:

* Los objetos se crean en la *UI*.
* El método `execute()` se llama en la *UI* y hace que `doInBackground(...)` se ejecute en la hebra de fondo.
* No se debe llamar manualmente a los métodos de la *AsyncTask*.
* Una *AsyncTask* sólo se puede ejecutar una vez.
* Si se intenta ejecutar varias *AsyncTask*, el sistema las ejecuta en secuencia, es decir, hasta que no termina una, no empieza otra.
* Si hacen falta varias tareas concurrentes, existen métodos para su implementación:
  - `task.executeOnExecutor(executor, params...)`
  - `task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, params...);`

Con todo ello, un ejemplo de uso de una AsyncTask sería:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_e3);

  display = (TextView) findViewById(R.id.e3_display);

  AsyncTask<Void, Integer, Boolean> task = new Task();
  task.execute();
}

private class Task extends AsyncTask<Void, Integer, Boolean> {
  protected void onPreExecute() {
    progress.setMax(100);
    progress.setProgress(0);
  }

  //El hecho de que aparezcan puntos suspensivos tras la clase de parámetros que se
  // recibe en un método quiere decir que se trata de un array

  protected Boolean doInBackground(Void... params) {
    for (int vez = 0; vez < VECES; vez++) {
      Thread.sleep(1000);
      publishProgress((vez + 1) * 10); //Con este método se le manda a la UI como va
                                       // avanzando el progreso de carga
    }
    return true;
  }

  // Este método recibe de publishProgress() el progreso y lo actualiza en la UI
  protected void onProgressUpdate(Integer... progressValues) {
    int progreso = (int) progressValues[0];
    progress.setProgress(progreso);
  }

  protected void onPostExecute(Boolean... result) {...}

}
```