

# Programación de Dispositivos Móviles



## Sesión 2: Java para MIDs. MIDlets

# Índice



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# Java para MIDs



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# Configuración CLDC



- **Características básicas del lenguaje**
  - **Mantiene la sintaxis y tipos de datos básicos del lenguaje Java**
  - **No existen los tipos `float` y `double`**
- **Similar a la API de J2SE**
  - **Mantiene un pequeño subconjunto de las clases básicas de J2SE**
    - **Con una interfaz más limitada en muchos casos**
  - **Excepciones**
  - **Hilos**
    - **No soporta hilos de tipo *daemon***
    - **No soporta grupos de hilos**
  - **Flujos básicos de E/S**
    - **No hay flujos para acceder a ficheros**
    - **No hay tokenizadores**
    - **No hay serialización de objetos**
    - **Destinados principalmente a conexiones de red y memoria**

# Características ausentes



- **No soporta números reales**
  - No existen los tipos `float` y `double`
- **Desaparece el marco de colecciones**
  - Sólo se mantienen las clases `Vector`, `Stack` y `Hashtable`
- **Desaparece la API de *reflection***
  - Sólo se mantienen las clases `Class` y `Object`
- **Desaparece la API de red `java.net`**
  - Se sustituye por una más sencilla (GCF)
- **Desaparece la API de AWT/Swing**
  - Se utiliza una API adecuada para la interfaz de los dispositivos móviles (LCDUI)

# Java para MIDs



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# CLDC y los números reales



- **En CLDC 1.0 no tenemos soporte para números reales**
  - Los tipos `float` y `double` no existen
- **En muchas aplicaciones podemos necesitar trabajar con este tipo de números**
  - P.ej. para cantidades monetarias
- **Podemos implementar números de coma fija usando enteros**
  - Existen librerías como *MathFP* que realizan esta tarea
- **En CLDC 1.1 ya existe soporte para los tipos `float` y `double`**

# Números reales sobre enteros



- Podemos representar números de coma fija como enteros
  - Consideramos que los últimos N dígitos son decimales
  - Por ejemplo, **1395** podría representar **13.95**
- Podremos hacer operaciones aritméticas con ellos
  - Suma y resta
    - Se realiza la operación sobre los números enteros
    - El resultado tendrá tantos decimales como los operandos

$$13.95 + 5.20 \rightarrow 1395 + 520 = 1915 \rightarrow 19.15$$

- Multiplicación
  - Se realiza la operación sobre los números reales
  - El resultado tendrá tantos decimales como la suma del número de decimales de ambos operandos

$$19.15 * 1.16 \rightarrow 1915 * 116 = 222140 \rightarrow 22.2140$$



# Conversión de números reales a enteros



- **Deberemos convertir el entero a real para mostrarlo al usuario**

```
public String imprimeReal(int numero) {  
    int entero = numero / 100;  
    int fraccion = numero % 100;  
    return entero + "." + fraccion;  
}
```

- **Cuando el usuario introduzca un valor real deberemos convertirlo a entero**

```
public int leeReal(String numero) {  
    int pos_coma = numero.indexOf('.');  
    String entero = numero.substring(0, pos_coma - 1);  
    String fraccion = numero.substring(pos_coma + 1, pos_coma + 2);  
    return Integer.parseInt(entero)*100+Integer.parseInt(fraccion);  
}
```

# Java para MIDs



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# Temporizadores en los MIDs



- **Los temporizadores resultan de gran utilidad en los MIDs**
- **Nos permiten programar tareas para que se ejecuten en un momento dado**
  - Alarmas
  - Actualizaciones periódicas de software
  - Etc
- **En CLDC se mantienen las clases de J2SE para temporizadores**
  - `Timer` y `TimerTask`

# Definir la tarea



- **Deberemos definir la tarea que queremos programar**
  - La definimos creando una clase que herede de `TimerTask`
  - En el método `run` de esta clase introduciremos el código que implemente la función que realizará la tarea

```
public class MiTarea extends TimerTask {  
    public void run() {  
        // Código de la tarea  
        // ... Por ejemplo, disparar alarma  
    }  
}
```

# Programar la tarea



- Utilizaremos la clase **Timer** para programar tareas
- Para programar la tarea daremos
  - Un tiempo de comienzo. Puede ser:
    - Un retardo (respecto al momento actual)
    - Fecha y hora concretas
  - Una periodicidad. Puede ser:
    - Ejecutar una sola vez
    - Repetir con retardo fijo
      - Siempre se utiliza el mismo retardo tomando como referencia la última vez que se ejecutó
    - Repetir con frecuencia constante
      - Se toma como referencia el tiempo de la primera ejecución. Si alguna ejecución se ha retrasado, en la siguiente se recupera

# Programar con retardo



- Creamos la tarea y un temporizador

```
Timer t = new Timer();  
TimerTask tarea = new MiTarea();
```

- Programamos la tarea en el temporizador con un número de milisegundos de retardo

```
long retardo = 10000; // 10 segundos  
long periodo = 1000; // 1 segundo  
t.schedule(tarea, retardo); // Una vez  
t.schedule(tarea, retardo, periodo); // Retardo fijo  
t.scheduleAtFixedRate(tarea, retardo, periodo);  
// Frecuencia constante
```

# Programar a una hora



- **Debemos establecer la hora en la que se ejecutará por primera vez el temporizador**
  - Representaremos este instante de tiempo con un objeto `Date`
  - Podemos crearlo utilizando la clase `Calendar`

```
Calendar calendario = Calendar.getInstance();
calendario.set(Calendar.HOUR_OF_DAY, 8);
calendario.set(Calendar.MINUTE, 0);
calendario.set(Calendar.SECOND, 0);
calendario.set(Calendar.MONTH, Calendar.SEPTEMBER);
calendario.set(Calendar.DAY_OF_MONTH, 22);
Date fecha = calendario.getTime();
```

- **Programamos el temporizador utilizando el objeto `Date`**

```
t.schedule(tarea, fecha, periodo);
```

# Java para MIDs



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**



# Serialización manual



- **CLDC no soporta serialización de objetos**
  - **Conversión de un objeto en una secuencia de bytes**
  - **Nos permite enviar y recibir objetos a través de flujos de E/S**
  
- **Necesitaremos serializar objetos para**
  - **Hacer persistente la información que contengan**
  - **Enviar esta información a través de la red**
  
- **Podemos serializar manualmente nuestros objetos**
  - **Definiremos métodos `serialize` y `deserialize`**
  - **Utilizaremos los flujos `DataOutputStream` y `DataInputStream` para codificar y decodificar los datos del objeto en el flujo**

# Serializar



- Escribimos las propiedades del objeto en el flujo de salida

```
public class Punto2D {
    int x;
    int y;
    String etiqueta;
    ...
    public void serialize(OutputStream out) throws IOException {
        DataOutputStream dos = new DataOutputStream( out );
        dos.writeInt(x);
        dos.writeInt(y);
        dos.writeUTF(etiqueta);
        dos.flush();
    }
}
```

# Deserializar



- Leemos las propiedades del objeto del flujo de entrada
- Debemos leerlas en el mismo orden en el que fueron escritas

```
public class Punto2D {  
    ...  
    public static Punto2D deserialize(InputStream in)  
        throws IOException {  
        DataInputStream dis = new DataInputStream( in );  
  
        Punto2D p = new Punto2D();  
        p.x = dis.readInt();  
        p.y = dis.readInt();  
        p.etiqueta = dis.readUTF();  
  
        return p;  
    }  
}
```

# Java para MIDs



- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# Recursos en el JAR



- Hemos visto que podemos añadir cualquier tipo de recursos al JAR de nuestra aplicación
  - Ficheros de datos, imágenes, sonidos, etc
- Estos recursos no se encuentran en el sistema de ficheros
  - Son recursos del JAR
- Para leerlos deberemos utilizar el método `getResourceAsStream` de cualquier objeto `Class`:

```
InputStream in =  
    getClass().getResourceAsStream("/datos.txt");
```

- Es importante anteponer el nombre del recurso el carácter `"/"` para que acceda de forma relativa al raíz del JAR

# Java para MIDs

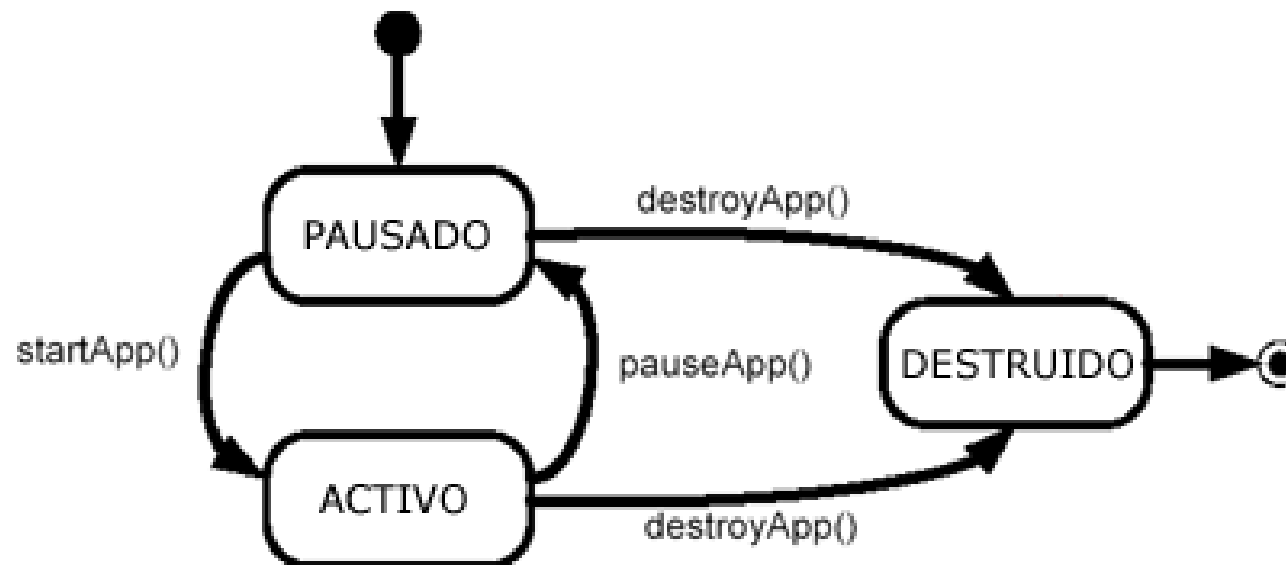


- **Características de CLDC**
- **Números reales**
- **Temporizadores**
- **Serialización de objetos**
- **Acceso a los recursos**
- **MIDlets**

# Ciclo de vida



- La clase principal de la aplicación debe heredar de `MIDlet`
- Componente que se ejecuta en un contenedor
  - AMS = Software Gestor de Aplicaciones
- El AMS controla su ciclo de vida



# Esqueleto de un MIDlet



```
import javax.microedition.midlet.*;

public class MiMIDlet extends MIDlet {
    protected void startApp()
        throws MIDletStateChangeException {
        // Estado activo -> comenzar
    }

    protected void pauseApp() {
        // Estado pausa -> detener hilos
    }

    protected void destroyApp(boolean incondicional)
        throws MIDletStateChangeException {
        // Estado destruido -> liberar recursos
    }
}
```



# Propiedades



- **Leer propiedades de configuración (JAD)**

```
String valor = getAppProperty(String key);
```

- **Salir de la aplicación**

```
public void salir() {  
    try {  
        destroyApp(false);  
        notifyDestroyed();  
    } catch(MIDletStateException e) { }  
}
```