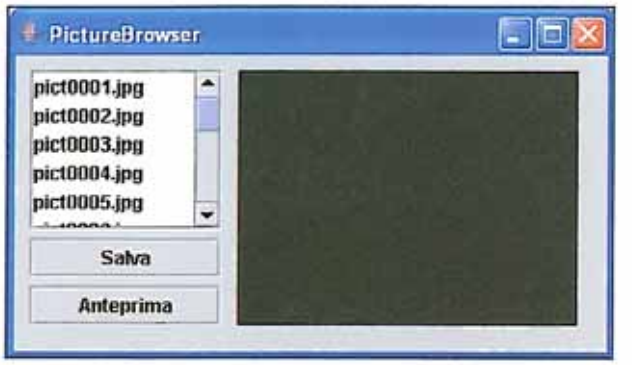
UN 'BROWSER' PER LE FOTOGRAFIE

Tra le caratteristiche offerte dalla nuova libreria Java, fornita nel CD 4, vi è la possibilità di esplorare il filesystem remoto di I-Droid01. Il programma presentato in questo esempio permette di leggere le diverse immagini memorizzate a bordo del robot, visualizzarle e salvarle sul proprio computer.

La nuova libreria Java, che hai trovato nel quarto CD-ROM, allegato al fascicolo 87, introduce alcune nuove caratteristiche che estendono ulteriormente le possibilità di programmazione di I-Droid01. Nell'esempio mostrato nelle prossime pagine realizzeremo un programma che permette di 'navigare' tra le fotografie memorizzate nella memoria Flash del robot, visualizzarne l'anteprima ed eventualmente salvare l'immagine sul proprio personal computer. L'elenco dei file viene ottenuto grazie a una nuova classe Java che permette di accedere alle diverse informazioni relative al filesystem di I-Droid01. L'interfaccia dell'applicazione è costituita da quattro elementi: una lista scorrevole per la selezione delle fotografie, un riquadro per mostrare le foto, due pulsanti per l'anteprima e il salvataggio delle foto sul PC. Le operazioni di anteprima e salvataggio richiedono un trasferimento di dati tra il robot e il PC, per il quale sono necessari alcuni secondi. L'applicazione è costituita da un'unica classe Java, **PictureBrowser**, che analizzeremo nel prossimo paragrafo. Come al solito dovrai salvare il codice sorgente dell'applicazione in un file di testo denominato **PictureBrowser.java**, compilarlo e avviare l'esecuzione da prompt dei comandi.

Sopra, la schermata mostra la consueta procedura per l'avvio del programma PictureBrowser. Come al solito va specificata la porta seriale per effettuare la connessione con il robot.

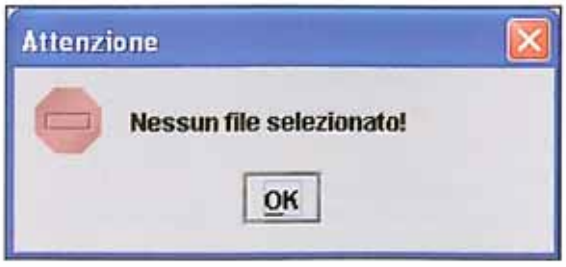


Sopra, l'interfaccia grafica del programma mentre viene visualizzata l'anteprima di una fotografia. Più in alto, invece, la stessa interfaccia subito dopo l'avvio dell'applicazione.

PICTUREBROWSER

La classe **PictureBrowser** fornisce i metodi e gli attributi necessari per la creazione dell'interfaccia grafica e per la gestione della comunicazione tra il robot e il PC. Tra le consuete dichiarazioni di pacchetto notiamo la presenza dell'interfaccia **DirectoryEnumeration**: quest'ultima, implementata internamente alla classe **TransferHandler**, fornisce alcuni metodi per esplorare e interagire con il filesystem di I-Droid01. Come nella maggior parte dei programmi presentati sino a ora, la classe principale del programma (che in questo caso è

anche l'unica) estende **JFrame**. Il metodo **main** crea un'istanza di **PictureBrowser** e avvia l'applicazione, verificando che l'utente abbia passato al programma la porta seriale da usare per la comunicazione con il robot. Analizziamo il costruttore della classe. Per prima cosa vengono creati i due pulsanti **save** e **preview** adibiti, rispettivamente, al salvataggio e alla visualizzazione delle immagini. Il metodo **connetti**, a differenza degli esempi già presentati, crea un'istanza di **TransferHandler** e la registra come 'capability attiva' per mezzo del metodo **addCapability**. Dopo aver effettuato la connessione, il costruttore crea un elenco delle immagini presenti a bordo di I-Droid01 e implementa la **JList pictures** che servirà, appunto, per la loro selezione. L'ultimo degli oggetti grafici deriva da una sottoclasse (**ImagePanel**) di **JPanel** che si occupa di visualizzare le fotografie trasferite dal robot al PC. In particolare è possibile notare come venga ridefinito il metodo **paintComponent**, invocato ogni qual volta si renda necessario effettuare un refresh del contenuto del pannello grafico. Qualora non sia presente in memoria nessuna immagine, il pannello visualizza un semplice riquadro nero. Facciamo un passo indietro e analizziamo i gestori di eventi associati ai due pulsanti. Alla pressione del pulsante **preview**, l'immagine selezionata dall'utente viene trasferita dal robot al PC e visualizzata sull'interfaccia grafica. Per il trasferimento dell'immagine viene invocato il metodo **transferPicture**, del quale ci occuperemo più avanti. Se non è selezionata nessuna immagine, viene visualizzata una finestra di dialogo per notificare il problema all'utente. A questo scopo viene utilizzato il metodo **showMessageDialog** della classe Java **JOptionPane**. La gestione del pulsante **save** è simile, con la differenza che all'utente viene presentata una finestra di dialogo (tramite il metodo **showSaveDialog** della classe **JFileChooser**) per specificare dove salvare l'immagine trasferita da I-Droid01 al PC. Analizziamo ora in dettaglio i rimanenti metodi della classe **PictureBrowser**. Il primo che incontriamo è



Sopra, la finestra di dialogo che viene visualizzata se l'utente preme il pulsante 'Salva' o 'Anteprima' senza che sia stata selezionata un'immagine tra quelle presenti nell'elenco.

getPicturesList. Scopo di questo metodo è collegarsi al robot per ottenere l'elenco delle immagini presenti sulla sua memoria Flash. Per prima cosa viene effettuata una chiamata al metodo **listDirectory** dell'oggetto **transfer**, che restituisce un oggetto **DirectoryEnumeration**.

Sopra, il metodo showSaveDialog della classe JFileChooser mostra un'interfaccia che permette all'utente di selezionare il nome e il percorso di salvataggio dell'immagine da salvare.

Grazie ai metodi di quest'ultimo (**nextEntry**, **getEntry** e **getEntryType**) è possibile esplorare il filesystem remoto del robot e ottenere l'elenco dei file presenti nella cartella 'pictures', che contiene le immagini scattate da I-Droid01. Il nome di ciascun file viene dapprima memorizzato in un oggetto di tipo **Vector** e solo in seguito copiato in un array di **String**. Il motivo di tale passaggio sta nel fatto che non conosciamo a priori il numero di file contenuti all'interno della directory. Contrariamente agli array, che hanno una dimensione fissa, i **Vector** possono crescere in maniera dinamica con l'aggiunta di nuovi elementi. Una volta noto il numero di file viene quindi creato un array di dimensioni prestabilite, restituito poi al chiamante. Il metodo successivo, **transferPicture**, effettua il trasferimento vero e proprio di un'immagine dal robot al computer. Come si può notare dal codice del programma, tale trasferimento avviene dopo aver specificato il nome del file da trasferire a **openInputStream**, che crea un 'canale' per il trasferimento del file. I dati 'grezzi' vengono passati a **ImageIO.read**, che li converte in un oggetto **Image**. L'ultimo metodo (a parte **disconnetti** che, essendo invariato rispetto agli esempi già visti nei fascicoli precedenti, non analizzeremo) è **savePicture**. Il suo scopo è quello di trasferire dal robot l'immagine selezionata dall'utente e salvarla sul PC grazie a un'interfaccia predefinita che permette di selezionare il nome con il quale salvare il file e il percorso di destinazione.

ESEMPI DI PROGRAMMAZIONE

Codice di implementazione dei metodi della classe PictureBrowser. La classe fornisce i metodi necessari alla costruzione dell'interfaccia grafica del programma e all'interazione tra robot e PC. I metodi di DirectoryEnumeration permettono di esplorare il contenuto delle directory del filesystem di I-Droid01. In particolare, nell'esempio riportato, vengono utilizzati i metodi nextEntry, getEntry e getEntryType per ottenere la lista delle immagini memorizzate a bordo del robot. Il trasferimento delle immagini avviene sfruttando la capability transfer. Per la loro visualizzazione, il programma si affida alla classe interna ImagePanel, che estende JPanel. In particolare i dati trasferiti dal robot al PC vengono passati al metodo read della classe ImageIO che, a partire da un flusso di byte, restituisce un oggetto di tipo Image. La stessa classe ImageIO fornisce anche l'utile metodo write, che permette di memorizzare un oggetto Image in un file. Nel codice compaiono alcuni nuovi metodi, come showMessageDialog (della classe JOptionPane) che visualizza una finestra di dialogo con un semplice messaggio, e l'utilissimo showSaveDialog (della classe fileChooser) che permette di selezionare interattivamente un file per il salvataggio.

METODI DELLA CLASSE PICTUREBROWSER

```
package communication.examples;

import communication.handler.internal.InternalHandler;
import communication.handler.ProtocolHandler;
import communication.transport.ConnectionProvider;
import communication.handler.transfer.TransferHandler;
import communication.handler.transfer.DirectoryEnumeration;
import javax.imageio.ImageIO;
import java.io.*;
import java.util.Vector;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;


public class PictureBrowser extends JFrame {


    private InternalHandler internal;
    private TransferHandler transfer;
    private ProtocolHandler protocol;
    private DirectoryEnumeration enumDir;
    private static String portName;
    private JScrollPane scroll;
    private JButton preview;
    private JButton save;
    private JList pictures;
    private Image img = null;
    private ImagePanel panelPicture;
    private Container c;


    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Specifica la porta da usare per la
connessione (es. COM5)");
            return;
        }
        portName = args[0];
        PictureBrowser pictureBrowser = new PictureBrowser();
    } // main


    public PictureBrowser() {
        super("PictureBrowser");
        addWindowListener(
            new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    disconnetti();
                    System.exit(0);
                }
            }
        );
        setBounds(0,0,380,220);
        c = getContentPane();
        c.setLayout(null);
        save = new JButton("Salva");
        save.setBounds(10,115,120,25);
        preview = new JButton("Anteprima");
        preview.setBounds(10,145,120,25);
        c.add(save);
        c.add(preview);
        save.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if (pictures.getSelectedIndex() == -1) {
                        JOptionPane.showMessageDialog(
                        null, "Nessun file selezionato!",
                        "Attenzione",
                        JOptionPane.ERROR_MESSAGE);
                        return;
                    }
                    String pic = (String)pictures.getSelectedValue();
                    transferPicture(pic);
                    savePicture();
                }
        });
        preview.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if (pictures.getSelectedIndex() == -1) {
                        JOptionPane.showMessageDialog(null,
                        "Nessun file selezionato!",
                        "Attenzione",
                        JOptionPane.ERROR_MESSAGE);
                        return;
                    }
                    String pic = (String)pictures.getSelectedValue();
                    transferPicture(pic);
                    panelPicture.repaint();
                }
        });
        if (!connetti()) System.exit(0);
        pictures = new JList(getPicturesList());
        pictures.setSelectionMode(
ListSelectionModel.SINGLE_SELECTION);
        scroll = new JScrollPane(pictures);
        scroll.setBounds(10,10,120,100);
        c.add(scroll);
```

```
        panelPicture = new ImagePanel();
        panelPicture.setBounds(140,10,213,160);
        c.add(panelPicture);
        setVisible(true);
    } // class constructor


    private String[] getPicturesList() {
        int i = 0;
        String[] picFiles = null;
        Vector temp = new Vector();
        try {
            enumDir = transfer.listDirectory("/mnt/disk/pictures/");
            while (enumDir.nextEntry()) {
                String entry = enumDir.getEntry();
                if (enumDir.getEntryType() == DirectoryEnumeration.FILE) {
                        temp.add(entry);
                }
            }
            i = temp.size();
            if (i==0) return null;
            picFiles = new String[i];
            for (int x=0; x<i; x++) {
                picFiles[x]= (String)temp.get(x);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return picFiles;
    } // getPicturesList


    private class ImagePanel extends JPanel {
        int xl, yl;
        public void paintComponent(Graphics g) {
            if (img !=null) {
                xl = img.getWidth(this);
                yl = img.getHeight(this);
                g.drawImage(img,0,0,xl/3,yl/3, this);
            } else {
                g.fillRect(0,0,213,160);
            }
        }
    } // class ImagePanel


    private void transferPicture(String file) {
        try {
            InputStream is = transfer.openInputStream("pictures/" + file);
            img = ImageIO.read(is);
            is.close();
            panelPicture.repaint();
        } catch (Exception e) {
            e.printStackTrace();
        }
    } // transferPicture


    private void savePicture() {
        OutputStream output;
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setFileSelectionMode(JFileChooser.FILES_ONLY);
        int result = fileChooser.showSaveDialog(this);
        if (result == JFileChooser.CANCEL_OPTION) return;
        File filename = new
File(fileChooser.getSelectedFile().toString() + ".jpg");
        if (filename == null || filename.getName().equals("")) {
            JOptionPane.showMessageDialog(this,
            "Nome file non valido!",
            "Attenzione",
            JOptionPane.ERROR_MESSAGE);
        } else {
            try {
                ImageIO.write((RenderedImage)img,"jpg", filename);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    } // savePicture


    private boolean connetti() {
        ConnectionProvider provider = new
SerialJavaxCommProvider(portName,9600);
        protocol = new ProtocolHandler(provider, 9600);
        internal = new InternalHandler();
        transfer = new TransferHandler();
        protocol.addCapabilityHandler(internal);
        protocol.addCapabilityHandler(transfer);
        System.out.println("Connessione in corso...");
        try {
            if (!protocol.connect()) {
                System.err.println("Impossibile connettersi!");
                return false;
            }
            if (!protocol.setup(5000)) {
                System.err.println("Settaggio della connessione fallito!");
                protocol.disconnect(true);
                return false;
            }
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        System.out.println("Client connesso");
        return true;
    } // connetti


    private void disconnetti() {
        System.out.println("Disconnessione in corso...");
        try {
            protocol.disconnect(false);
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }
        System.out.println("Client disconnesso");
    } // disconnetti


} // PictureBrowser
```

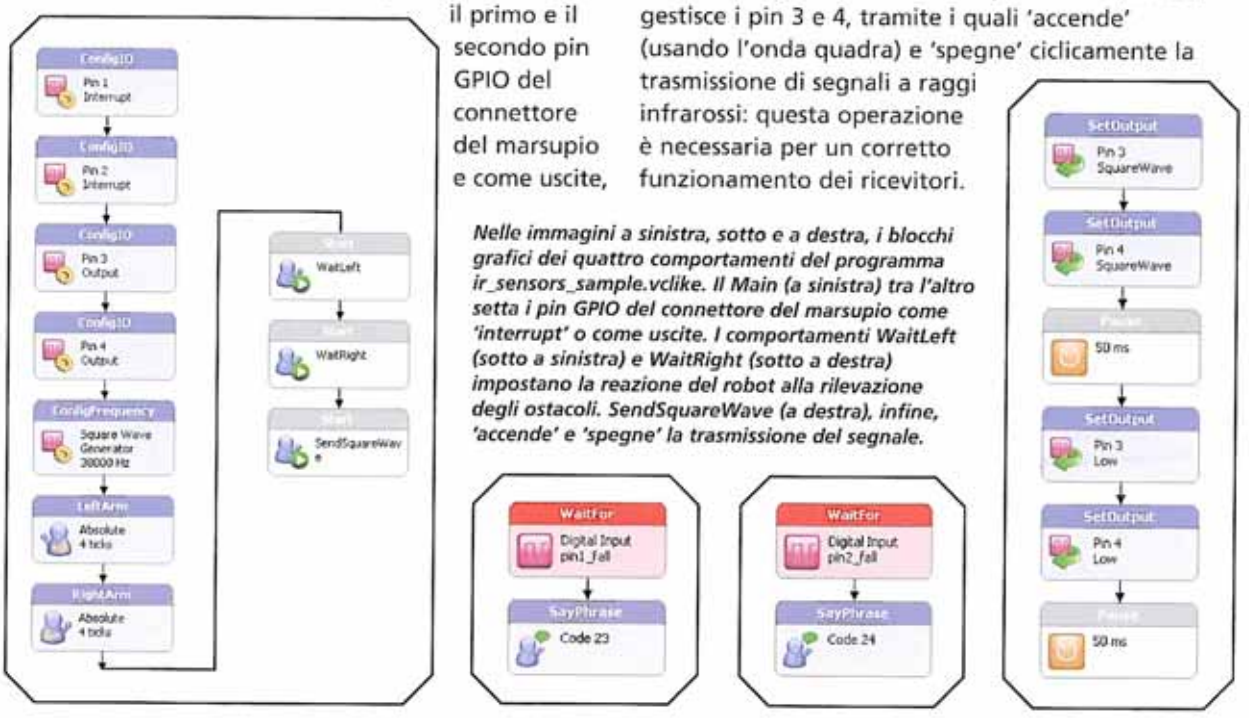
I SENSORI LATERALI IN VISUAL C-LIKE

Tra i programmi di esempio contenuti nel quarto CD-ROM di I-Droid01, ce n'è uno che mostra come sfruttare i sensori laterali a raggi infrarossi utilizzando la nuova versione del Visual C-like Editor. Diamo un'occhiata al codice.

Nel fascicolo 83, a pagina 16, abbiamo mostrato un codice di esempio in C-like per il test dei sensori a infrarossi montati sulla breadboard. Il programma consente di verificare il funzionamento dei sensori, facendo pronunciare a I-D01 la parola 'destra' se viene rilevato un ostacolo sulla destra del robot, 'sinistra' se l'ostacolo è 'visto' a sinistra. Con la nuova versione del Visual C-like Editor inclusa nel quarto CD, è possibile realizzare un programma dal funzionamento del tutto analogo, che sfrutta i blocchetti per la gestione dei GPIO.

I COMPORTAMENTI

Il programma **ir_sensors_sample.vclike** è accessibile dalla sezione SOFTWARE del CD, in particolare all'interno della voce relativa al Visual C-like Editor. I comportamenti che realizzano il programma sono quattro. **Main** configura come ingressi 'a interrupt' il primo e il secondo pin GPIO del connettore del marsupio e come uscite, invece, il terzo e quarto pin GPIO. In seguito, viene impostato un segnale 'a onda quadra' (ossia con la presenza di un livello 'alto' e di uno 'basso' che si alternano ciclicamente), in particolare definendone la frequenza (numero di volte al secondo in cui si alternano i livelli alto e basso), pari a 38.000 Hz: esso servirà per l'utilizzo delle uscite, in modo che i trasmettitori del kit generino un segnale a infrarosso caratterizzato da una frequenza (38 kHz) riconoscibile dai ricevitori. Per una guida all'utilizzo dei pin GPIO in Visual C-like, si veda comunque la guida all'Editor inclusa nel CD. Il **Main** prosegue collocando le braccia abbastanza in alto per non influire sui sensori, poi lancia i comportamenti **WaitLeft**, **WaitRight** e **SendSquareWave**. I primi due non fanno altro che aspettare che sui pin 1 e pin 2 venga rilevato il segnale emesso dai trasmettitori e riflesso dagli ostacoli. **SendSquareWave**, invece, gestisce i pin 3 e 4, tramite i quali 'accende' (usando l'onda quadra) e 'spegne' ciclicamente la trasmissione di segnali a raggi infrarossi: questa operazione è necessaria per un corretto funzionamento dei ricevitori.



Nelle immagini a sinistra, sotto e a destra, i blocchi grafici dei quattro comportamenti del programma ir_sensors_sample.vclike. Il Main (a sinistra) tra l'altro setta i pin GPIO del connettore del marsupio come 'interrupt' o come uscite. I comportamenti WaitLeft (sotto a sinistra) e WaitRight (sotto a destra) impostano la reazione del robot alla rilevazione degli ostacoli. SendSquareWave (a destra), infine, 'accende' e 'spegne' la trasmissione del segnale.