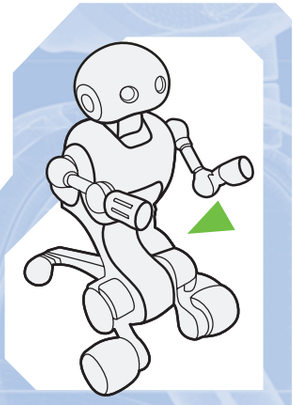
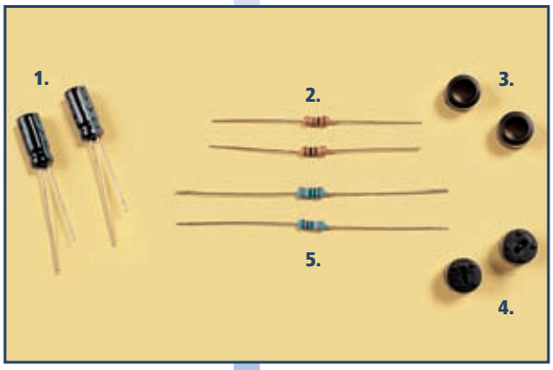
I CONDENSATORI E I RESISTORI DEL KIT



Gli elementi allegati a questo fascicolo fanno parte del kit per breadboard dei sensori a infrarossi: si tratta, soprattutto, dei due condensatori e dei quattro resistori.

Con questo fascicolo inizia la collezione degli elementi del kit per la breadboard, che realizzerà un sistema di rilevamento degli ostacoli tramite raggi infrarossi. In attesa, quindi, di completare la pinza con la scheda Hand, potrai presto montare sulla breadboard il nuovo sistema sensoriale, che doterà I-Droid01 di una 'vista' laterale. I primi elementi del kit sono due condensatori, quattro resistori e quattro elementi in materiale plastico. I condensatori serviranno, assieme a una coppia di resistori, a 'proteggere' i ricevitori a infrarossi da eventuali disturbi nella loro alimentazione. Gli altri due resistori, invece, avranno il compito di regolare la tensione fornita ai trasmettitori. Le due coppie di resistori si distinguono per la resistenza che le caratterizza: i resistori collegati ai ricevitori (con i condensatori) hanno una resistenza nominale di 100 ohm; quelli per l'alimentazione dei trasmettitori hanno resistenza nominale di 390 ohm. Infine, completano l'insieme degli elementi allegati quattro pezzi in materiale plastico. Due di essi sono i cappucci per i ricevitori, grazie ai quali i trasmettitori potranno concentrare i propri 'raggi' in una determinata direzione. Gli altri due elementi, invece, sono cilindri 'di supporto', che verranno connessi ai cappucci e consentiranno di mantenere questi ultimi in posizione.

COMPONENTI

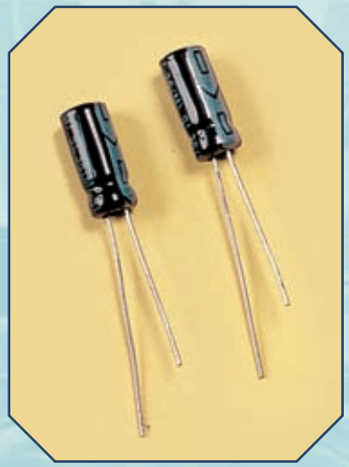


1. 2 condensatori
2. 2 resistori da 100 ohm
3. 2 cappucci
4. 2 cilindri per i cappucci
5. 2 resistori da 390 ohm

I CONDENSATORI

DATI

I condensatori allegati a questo fascicolo sono identici tra loro per forma e caratteristiche. Di tipo elettrolitico, entrambi presentano una capacità di 4,7 μF (ossia 4,7 microfarad, dove farad è l'unità di misura della capacità elettrica). Esteticamente, essi si presentano come cilindri dotati di due contatti metallici. Tali contatti non sono uguali (a destra): la loro lunghezza ne indica la funzione di anodo (contatto lungo) e catodo (morsetto corto). L'inserimento sulla breadboard è importante: un condensatore collegato al contrario potrebbe danneggiarsi in modo irreparabile e recare danni anche agli altri componenti.



CONTROLLARE LE BRACCIA IN JAVA

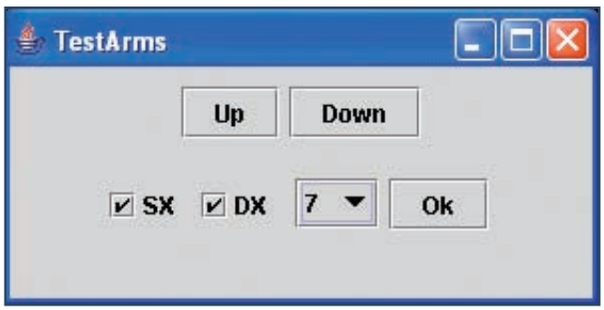
Prosegue la carrellata dei programmi di esempio scritti in Java. Questa volta il codice è relativo al controllo dei movimenti delle due braccia di I-D01, che può essere effettuato attraverso un'interfaccia grafica, simile a quella mostrata per la gestione dei LED della testa, presentata nel fascicolo 78.

Dopo aver visto come realizzare un pannello grafico di gestione, da PC, dell'accensione e spegnimento dei LED inclusi nella testa di I-D01 (fascicolo 78, pagina 11 e seguenti), continuiamo la presentazione di alcuni semplici programmi in Java. Stavolta viene mostrato il codice relativo a un pannello grafico di controllo dei movimenti delle braccia del robot, agendo, in particolare, sui motori posti nelle spalle di I-Droid01. Il pannello è realizzato con lo stesso stile di quello per i LED: ancora una volta, quindi, la gestione da parte dell'utente avviene utilizzando elementi grafici come pulsanti, menu a tendina e box di selezione. Inoltre, in modo ancora una volta del tutto analogo a quanto già fatto per i LED, anche in questo caso il programma è organizzato in due classi Java, **ArmsControl** e **TestArms**. La prima fornisce i metodi di controllo dei movimenti sia del braccio sinistro sia del braccio destro, che a loro volta richiamano le apposite procedure fornite dalle librerie di supporto. La seconda, invece, modella il pannello di controllo e si occupa della gestione dell'interfaccia grafica visualizzata sul monitor del PC, oltre che delle procedure di connessione e disconnessione tra PC e I-Droid01. I metodi presentati, soprattutto quelli della classe **ArmsControl**, possono essere facilmente riutilizzati, eventualmente dopo averli personalizzati secondo le proprie esigenze, nelle applicazioni che debbano gestire i movimenti delle braccia. La compilazione e l'esecuzione, ancora una volta, possono essere effettuate come già illustrato nei fascicoli scorsi. Infine, va ricordato che i codici delle due classi vanno memorizzati in file dai nomi a esse coerenti, e cioè, in questo caso, **ArmsControl.java** e **TestArms.java**. Vediamo ora più da vicino come sono state implementate le due classi.

'CONTROLLORE' E 'TESTER'

La struttura del programma Java per il controllo dei motori delle braccia è stata volutamente scelta per essere coerente con quella che caratterizza l'applicazione di gestione dei LED. In questo modo, infatti, risulta più semplice la comprensione dei diversi metodi impiegati e il loro riutilizzo futuro. Le due classi, quindi, modellizzano l'oggetto 'controllore braccia' (classe **ArmsControl**) e 'tester per le braccia' (**TestArms**). Il primo permette l'accesso alle funzioni di basso livello che riguardano i motori; il secondo invece fornisce l'intermediazione tra i comandi dell'utente e il controllore.

Sotto, il pannello grafico di controllo dei motori delle braccia, dotato di tre pulsanti, un menu a tendina e due box di selezione. Più in basso, invece, la schermata del prompt dei comandi nel quale viene lanciata l'esecuzione del programma.



ARMSCONTROL

Come per tutte le altre classi, anche il codice di **ArmsControl** comincia con alcune dichiarazioni preliminari alla classe stessa. Più precisamente, la prima riga di codice della classe consiste nella solita dichiarazione **package communication.examples**, seguita da tre **import**, relativi alle librerie di appoggio **InternalHandler**, **ArmsRegister** e **InternalModule**, tutte appartenenti alla directory **communication.handler.internal**. La seconda delle librerie citate (**ArmsRegister**) permette di accedere ai registri di controllo dei motori delle braccia di I-Droid01: il loro controllo, infatti, come già quello dei LED della testa, viene realizzato accedendo in lettura e scrittura ad appositi registri del robot. Subito dopo le righe dedicate alle importazioni, inizia il codice della classe vera e propria, con la dichiarazione del 'solito' attributo **internal**, oggetto di tipo **InternalHandler**, già incontrato sia in **LedControl**, sia nella classe di esempio per la connessione e disconnessione mostrata nel fascicolo 77. L'oggetto **internal** è l'unico attributo della classe **ArmsControl**; dopo la sua dichiarazione il codice della classe prosegue con i metodi. Il primo di essi è il costruttore omonimo della classe, che si limita a inizializzare l'attributo **internal**, in modo del tutto identico a quanto fatto dal costruttore della classe **LedControl** (si veda anche fascicolo 78, pagina 12). Gli altri metodi realizzano nel concreto i movimenti delle braccia. A tale proposito, va detto che i movimenti che possono essere effettuati sono di due tipologie. La classe qui mostrata, infatti, permette di posizionare le braccia in una postura precisa oppure di spostarle in modo relativo rispetto alla posizione precedentemente assunta. Così, i metodi 'di movimento' della classe risultano quattro: **leftArm** (posizionamento del braccio sinistro), **rightArm** (posizionamento del braccio destro), **leftArmRel** (spostamento relativo del braccio sinistro), **rightArmRel** (spostamento relativo del braccio destro). I primi due utilizzano come parametro la posizione desiderata per il braccio comandato, mentre i rimanenti ricevono come input il numero di 'passi' da far compiere al motore della spalla. Per entrambe le tipologie, i parametri di ingresso sono numeri interi: sono previste, infatti, 16 posizioni diverse che le braccia possono assumere. Così, se si vuole che il braccio sinistro sia posizionato nella posizione '12', si dovrà invocare il metodo **leftArm** passandogli come parametro il numero 12. Se, invece, si desidera spostare di tre posizioni in alto il braccio destro, si dovrà invocare il metodo **rightArmRel** con parametro pari a 3, mentre se si vuole spostare lo stesso braccio verso il basso di tre 'passi', si invocherà lo stesso metodo con parametro pari a –3. Non solo i movimenti desiderati, ma anche la posizione correntemente assunta dalle braccia può essere ricavata attraverso opportuni registri. I metodi della classe **ArmsControl** dapprima verificano che i motori delle braccia siano pronti (**internal.readRegister** con parametri **InternalModule.ARMS**, **ArmsRegister** con indicazione di LEFT o RIGHT e **buf**, variabile dove viene memorizzato il contenuto del registro). Poi leggono la posizione attuale (**internal.readRegister** con parametri **InternalModule.ARMS**, 13 o 14, rispettivamente per il braccio sinistro oppure il destro, e **buf**). Infine viene calcolata la nuova posizione, che viene memorizzata nel registro

ESEMPI DI PROGRAMMAZIONE

Esempio di codice Java per l'implementazione di una classe di controllo per le braccia di I-Droid01. Vengono mostrate solo le dichiarazioni iniziali di pacchetto, di importazione e dell'attributo internal di tipo InternalHandler. I metodi della classe (il costruttore 'ArmsControl', 'leftArm', 'rightArm', 'leftArmRel', 'rightArmRel') vengono solo citati. Il codice che ne implementa il funzionamento viene mostrato nelle prossime pagine.

CLASSE ARMSCONTROL

```
package communication.examples;

import communication.handler.internal.InternalHandler;
import communication.handler.internal.ArmsRegister;
import communication.handler.internal.InternalModule;

public class ArmsControl {

    private InternalHandler internal;

    public ArmsControl (InternalHandler internal) {
        this.internal = internal;
    } // ArmsControl

    public boolean leftArm(int pos) { [...] }

    public boolean rightArm(int pos) { [...] }

    public boolean leftArmRel(int step) { [...] }

    public boolean rightArmRel(int step) { [...] }

} // class ArmsControl
```

ESEMPI DI PROGRAMMAZIONE

Esempio di codice Java per l'implementazione di un pannello grafico di gestione delle braccia di I-Droid01. Sono mostrate le dichiarazioni di pacchetto, di importazione e relative agli attributi della classe. Viene mostrata anche la dichiarazione della 'sotto-classe' ArmsButtonHandler. Essa, come anche i metodi della classe TestArms ('main', il costruttore 'TestArms', 'createArmsPanel', 'connetti' e 'disconnetti'), viene solo citata. Il codice relativo è mostrato nelle prossime pagine. I metodi 'connetti' e 'disconnetti' hanno codice identico a quello mostrato nel fascicolo 77, pagine 13 e 14.

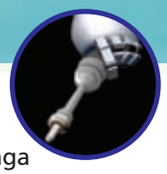
CLASSE TESTARMS

```
package communication.examples;

import communication.handler.ProtocolHandler;
import communication.handler.internal.InternalHandler;
import communication.transport.ConnectionProvider;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TestArms extends JFrame {

    private JPanel armsPanel;
    private JButton armUpB;
    private JButton armDownB;
    private JButton armAbsB;
    private JComboBox armPosCB;
    private JCheckBox armSxCB;
    private JCheckBox armDxCB;
    private String armValue[] =
        {"0","1","2","3","4","5","6","7","8","9","10","11","12","13","14","15"};
    private static String portName;
    private ProtocolHandler protocol;
    private InternalHandler internal;
    private ArmsControl arms;

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

    public TestArms () { [...] }

    JPanel createArmsPanel() { [...] }

    private class ArmsButtonHandler implements ActionListener {
        public void actionPerformed(ActionEvent e) { [...] }
    }

    private boolean connetti() { [...] }

    private void disconnetti() { [...] }

}
```

relativo (**internal.writeRegister** con parametri **InternalModule.ARMS**, **ArmsRegister** con indicazione LEFT o RIGHT e **buf**, dove è stata preventivamente memorizzata la nuova posizione).

TESTARMS

La classe **TestArms** non è molto dissimile, anche nel codice, da **TestLed**, presentata nel fascicolo 78. Anche in questo caso, si comincia con l'elencare la dichiarazione di pacchetto e quelle di importazione. Come nel caso di **TestLed**, queste ultime includono (oltre alle solite **ProtocolHandler**, **InternalHandler** e **ConnectionHandler** destinate alla connessione tra PC e robot) i pacchetti **awt**, **awt.event** e **swing**, relativi agli oggetti del pannello grafico di controllo. Il codice vero e proprio della classe inizia con le dichiarazioni degli attributi. Molti di questi ultimi sono istanze di oggetti grafici. Così, è presente un pannello (**armsPanel**), tre pulsanti (**armUpB**, **armDownB**, **armAbsB**), un menu a tendina (**armPosCB**) e due box di selezione (**armSxCB** e **armDxCB**). Gli altri attributi, invece, non hanno a che fare con la grafica: **portName**, **protocol** e **internal** hanno lo stesso ruolo già illustrato a proposito del programma di controllo dei LED; **arms** richiama la classe **ArmsControl**; **armValue**, infine, memorizza le possibili posizioni delle braccia, includendo in pratica i numeri interi compresi tra 0 e 15 (compresi). Nella classe **TestArms** è contenuto il metodo **main** del programma. Esso non fa altro se non controllare che sia stata indicata da parte dell'utente una porta di connessione, per poi lanciare il metodo costruttore. Quest'ultimo gestisce la chiusura del pannello e la terminazione del programma (invocando il metodo **disconnetti**), inizializza il pannello grafico (grazie a **createArmsPanel**), avvia la connessione (metodo **connetti**) e inizializza un oggetto 'controllore' (**new ArmsControl**). Come detto, il metodo **createArmsPanel** si occupa della realizzazione del pannello, istanziando i vari oggetti grafici al suo interno e associandovi un opportuno 'ascoltatore', ossia un oggetto di tipo **ArmsButtonHandler**, che ha il compito di 'rendersi conto' delle azioni compiute dall'utente sull'interfaccia. Analogamente a quanto fatto nella classe **TestLed**, anche nel caso di **TestArms** l'oggetto 'ascoltatore' viene descritto da una 'mini-classe' interna. Grazie al metodo **actionPerformed**, vengono monitorati gli oggetti dell'interfaccia, per poi invocare gli opportuni metodi della classe **ArmsControl**. Ancora una volta, quindi, tale metodo rappresenta il 'ponte' tra interfaccia e funzioni di controllo. Esso verifica se sono selezionate le box per la scelta del braccio da controllare (sinistro, destro o entrambi), per poi memorizzare la posizione scelta dall'utente attraverso il menu a tendina



(**armPosCB.getSelectedIndex**). Se viene premuto il tasto Up o quello Down (test **if** con variabile **e.getSource() == armUpB** oppure **e.getSource() == armDownB**), vengono invocati gli opportuni metodi della classe **ArmsControl** che implementano gli spostamenti relativi. Nel caso del programma mostrato, quando si premono i pulsanti Up e Down viene fatto alzare o abbassare il braccio selezionato di una posizione (con un 'passo' preimpostato, perciò, pari a 1): modificando opportunamente il codice si può fare in modo che l'utente possa scegliere il numero di 'passi' da far compiere alle braccia. Nel caso invece venga premuto il tasto Ok (**e.getSource() == armAbsB**), vengono invocati i metodi di posizionamento assoluto, facendo così in modo che i bracci scelti siano posti nella posizione definita dall'utente tramite il menu a tendina. I soliti metodi **connetti** e **disconnetti**, infine, forniscono la possibilità di collegare e scollegare PC e robot, in modo identico a quanto già mostrato all'interno degli esempi di programmi Java inseriti nei precedenti fascicoli.

ESEMPI DI PROGRAMMAZIONE

Codice di implementazione dei metodi della classe ArmsControl per il movimento delle braccia di I-Droid01. Viene mostrato il codice dei metodi di posizionamento assoluto e spostamento relativo del braccio sinistro ('leftArms' e 'leftArmsRel'). Il codice dei metodi per il braccio destro ('rightArms' e 'rightArmsRel') si ottiene sostituendo le formule in grassetto alle corrispettive formule dei metodi per il braccio sinistro.

METODI DI MOVIMENTO DELLE BRACCIA

```
public boolean leftArm(int pos) {
public boolean rightArm(int pos) {

    if ((pos < 0) || (pos > 15)) return false;

    try {
        boolean ready = false;
        int buf[] = new int[1];

        do {
            internal.readRegister(InternalModule.ARMS,
ArmsRegister.LEFT, buf);
            internal.readRegister(InternalModule.ARMS,
ArmsRegister.RIGHT, buf);
            if (buf[0]==1) {
                ready = false;
            } else {
                ready = true;
            }
        } while (!ready);

        // leggi la posizione
        internal.readRegister(InternalModule.ARMS, 13, buf);
        internal.readRegister(InternalModule.ARMS, 14, buf);

        int step = pos - buf[0];
        int temp = 0;
        int cmd;
        if (step < 0) {
            step = step * (-1);
            // Movimento verso il basso
            cmd = 3;
        } else if (step > 0) {
            // Movimento verso l'alto
            cmd = 1;
        } else return true;

        temp = step << 3;
        buf[0]= temp | cmd;
        internal.writeRegister(InternalModule.ARMS,
ArmsRegister.LEFT, buf);
        internal.writeRegister(InternalModule.ARMS,
ArmsRegister.RIGHT, buf);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;

} // leftArm


public boolean leftArmRel(int step) {
public boolean rightArmRel(int step) {

    int pos;

    try {
        boolean ready = false;
        int buf[] = new int[1];

        do {
            internal.readRegister(InternalModule.ARMS,
ArmsRegister.LEFT, buf);
            internal.readRegister(InternalModule.ARMS,
ArmsRegister.RIGHT, buf);
            if (buf[0]==1) {
                ready = false;
            } else {
                ready = true;
            }
        } while (!ready);

        // leggi la posizione
        internal.readRegister(InternalModule.ARMS, 13, buf);
        internal.readRegister(InternalModule.ARMS, 14, buf);

        pos = buf[0]+ step;
        if ((pos < 0) || (pos > 15)) return false;
        step = step << 3;
        if (step > 0) {
            buf[0]= step | 1;
        } else if (step < 0) {
            buf[0]= ((-1)*step) | 3;
        } else return false;
        internal.writeRegister(InternalModule.ARMS,
ArmsRegister.LEFT, buf);
        internal.writeRegister(InternalModule.ARMS,
ArmsRegister.RIGHT, buf);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }

    return true;

} // leftArmRel
```

ESEMPI DI PROGRAMMAZIONE

Codice di implementazione dei metodi della classe TestArms. Essa realizza il pannello grafico di controllo dei movimenti delle braccia e, in più, gestisce la connessione e disconnessione tra PC e I-Droid01 (tramite i metodi 'connetti' e 'disconnetti', il cui codice è identico a quello presentato nel fascicolo 77). Essa contiene l'implementazione della 'sotto-classe' ArmsButtonHandler, demandata al controllo dello stato degli elementi del pannello (pulsanti, menu e box di selezione), così da verificare la presenza di eventuali comandi da parte dell'utente.

METODI DELLA CLASSE TESTARMS

```
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];

    TestArms test = new TestArms();

}


public TestArms () {

    super("TestArms");

    addWindowListener(
        new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                disconnetti();
                System.exit(0);
            }
    });

    setBounds(0,0,300,150);
    armsPanel = createArmsPanel();
    getContentPane().add(armsPanel);
    setVisible(true);

    if (!connetti()) System.exit(1);

    arms = new ArmsControl(internal);

} // TestArms


JPanel createArmsPanel() {

    JPanel panel = new JPanel();
    JPanel panel1 = new JPanel();
    JPanel panel2 = new JPanel();
    armUpB = new JButton("Up");
    armDownB = new JButton("Down");
    armAbsB = new JButton("Ok");
    armPosCB = new JComboBox(armValue);
    armSxCB = new JCheckBox("SX");
    armDxCB = new JCheckBox("DX");
    ArmsButtonHandler abh = new ArmsButtonHandler();
    armUpB.addActionListener(abh);
    armDownB.addActionListener(abh);
    armAbsB.addActionListener(abh);
    JPanel abs = new JPanel();
    abs.add(armPosCB);
    abs.add(armAbsB);
    panel1.add(armUpB);
    panel1.add(armDownB);
    panel2.add(armSxCB);
    panel2.add(armDxCB);
    panel2.add(abs);
    panel.add(panel1);
    panel.add(panel2);

    return panel;

} // createArmsPanel


private class ArmsButtonHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        boolean sx,dx;

        sx = armSxCB.isSelected();
        dx = armDxCB.isSelected();

        int pos = armPosCB.getSelectedIndex();

        // Posizione relativa
        if (e.getSource() == armUpB){
            if(sx&&dx) {
                arms.leftArmsRel(1);
                arms.rightArmsRel(1);
                return;
            }
            if (sx) {
                arms.leftArmsRel(1);
                return;
            }
            if (dx) {
                arms.rightArmsRel(1);
                return;
            }
        }

        if (e.getSource() == armDownB){
            if(sx&&dx) {
                arms.leftArmsRel(-1);
                arms.rightArmsRel(-1);
                return;
            }
            if (sx) {
                arms.leftArmsRel(-1);
                return;
            }
            if (dx) {
                arms.rightArmsRel(-1);
                return;
            }
        }

        if (e.getSource() == armStopB){}

        // Posizione assoluta
        if (e.getSource() == armAbsB) {
            if(sx&&dx) {
                arms.leftArms(pos);
                arms.rightArms(pos);
                return;
            }
            if (sx) {
                arms.leftArms(pos);
                return;
            }
            if (dx) {
                arms.rightArms(pos);
                return;
            }
        }
    }

} // class ArmsButtonHandler
```