Programación con Karel

Ramón y Johan

Taller de Ciencia para Jóvenes
del 11 al 17 de Julio de 1999

1 El robot y su mundo

1.1 Introducción

En este curso pretendemos introducir varios conceptos básicos de programación, la meta no es aprender un lenguaje en particular, aunque en la práctica eso es lo que haremos. Se trata de aprender conceptos comúnes a la mayoría de los lenguajes, tales como: procedimientos, algoritmos, condicionales, etc., y, sobre todo, adquirir la habilidad de resolver el tipo de problemas que son "atacables" con los lenguajes de programación. Para esto se hace énfasis en dos aspectos al resolver los problemas: la planeación y la implementación. El trabajo durante el curso lo haremos usando un robot imaginario al que le daremos las órdenes para verificar si hemos resuelto adecuadamente los problemas que se nos presentan.

1.2 El mundo de Karel

Karel es un robot, en principio, un robot imaginario al cual podemos controlar por medio de un programa para que realice cierto trabajo.

El mundo de Karel consta de los siguentes elementos:

- Karel, el protagonista
- Calles (horizontales) y avenidas (verticales) numeradas.
- Esquinas y origen

- Paredes impenetrables
- Bipers removibles
- Bolsa de bipers

El manejo y limitaciones de cada uno de estos elementos es lo que pretendemos entender a lo largo de esta parte del curso.

La manera de comunicarse con Karel es por medio de un programa, el problema principal es que lo único que Karel puede hacer es seguir lo que le indiquemos "al pie de la letra". Karel no piensa y no puede darse cuenta de lo que queremos que haga si no sabemos como decírselo, para eso aprenderemos a comunicarnos adecuadamente con Karel.

Para comunicarnos con Karel lo hacemos por medio de un programa, y lo que aprenderemos es programación, es decir cual es la manera de hacer que Karel haga lo que nosotros queremos, para ello, tenemos que aprender su lenguaje, y este es un *lenguaje de programación*.

Lo que puede hacer Karel es, en primera instancia, muy limitado, de hecho comenzaremos con un conjunto muy reducido de instrucciones que podemos darle. Parte de la moraleja es ver cómo con un número reducido de instrucciones podemos llegar a hacer trabajos bastante sofisticados.

1.3 Situaciones y trabajos

En cada momento para describir la situación en la que se encuentra Karel necesitamos detallar todos los elementos que conforman el mundo de Karel, así, una tarea o trabajo específico de Karel consistirá en llevarlo de una situación *original* a una *final*.

1.4 Problemas

1. ¿En que direcciones puede Karel apuntar?
   - noreste
   - este
   - sur-sureste
   - norte
   - 164 grados

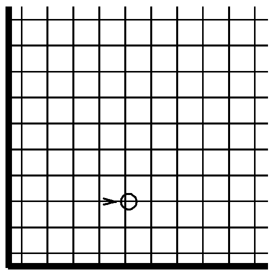
- vertical
- para abajo

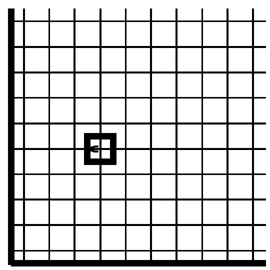
2. ¿Qué objetos además de Karel forman parte de su mundo?

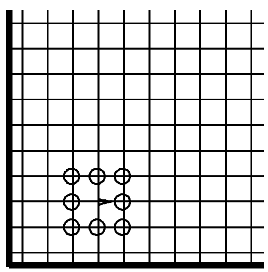
3. ¿Cuáles de estos objetos puede Karel manipular o cambiar?

4. ¿Qué puntos de referencia se pueden usar en el mundo de Karel para describir su posición exacta en el mundo?

5. Da la posición exacta y relativa de Karel en los siguentes mundos:







2 Primeras instrucciones y programas

Comenzaremos con las intrucciones básicas que Karel puede ejecutar. Cuando hablamos de ejecutar una instrucción nos referimos a la acción que Karel

realiza asociada con esa instrucción. Podemos tambien referirnos a ejecutar un programa, lo cual quiere decir llevar a cabo las instrucciones del programa.

- Cambio de posición:
  - `move` moverse en la dirección que está apuntando. (Puede causar error "apagón".)
  - `turnleft` girar a la izquierda (siempre se puede)

  No necesitamos `turnright`, ¿Por qué?

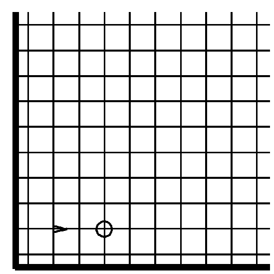
- Manipulando bipers
  - `pickbeeper` recoge un biper de la esquina dónde está parado
  - `putbeeper` deposita un biper

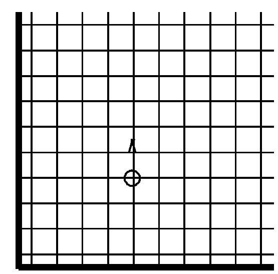
  Pueden ocasionar apagones.

- Terminado `turnoff`

2.1 Primer programa completo

Queremos que Karel siendo que se encuentra apuntando al este en la calle 2 avenida 2 realice la tarea de llevar el biper que se encuentra en la calle 2 avenida 4 a la calle 4 avenida 5, y debe moverse una cuadra más al norte antes de apagarse.





```
BEGINNING_OF_PROGRAM
  BEGINNING_OF_EXECUTION
    move;
    move;
    pickbepper;
    move;
    turnleft;
    move;
    move;
    putbeeper;
    move;
    turnoff;
  END_OF_EXECUTION
END_OF_PROGRAM
```

Al ejecutar el programa, Karel revisa que no tenga errores y luego procede a llevar a cabo cada una de las instrucciones entre las *palabras reservadas*: `BEGINNING_OF_EXECUTION` y `END_OF_EXECUTION`, hasta que se produce un apagón o encuentra un `turnoff`.

No basta con escribir el programa, hay que simularlo, para verificar que realiza lo que queremos.

2.1.1 Gramática

Es importante respetar las regals de puntuación y gramática para no confundir a Karel. Hay tres tipos de palabras o symbolos que Karel entiende, sin distinción de mayusculas, minusculas o indentación:

- puntuación (solo el "punto y coma")
- instrucciones
  - `move`, `turnleft`, `pickbeeper`, `putbeeper`, `turnoff`
- palabras reservadas (mayusculas por nuestra conveniencia)
  - `BEGINNING_OF_PROGRAM`, `END_OF_PROGRAM`
  - `BEGINNING_OF_EXECUTION`, `END_OF_EXECUTION`
  - `{`, `}`

  - `IF`, `THEN`, `ELSE`
  - `ITERATE`, `TIMES`
  - `WHILE`, `DO`
  - `DEFINE_NEW_INSTRUCTION`, `AS`
- pruebas
  - `front_is_clear`, `front_is_blocked`, `left_is_clear`, `left_is_blocked`, `right_is_clear`, `right_is_blocked`,
  - `next_to_a_beeper`, `not_next_to_a_beeper`,
  - `facing_north`, `not_facing_north`, `facing_south`, `not_facing_south`, `facing_east`, `not_facing_east`, `facing_west`, `not_facing_west`,

2.2 Errores

Hay varios tipos de dificultades con las que que podemos toparnos al escribir un programa. Una de las partes más complicadas de la programación es el detectar donde estan las fallas de nuestro programa.

2.2.1 Apagón

Karel se ve impedido de realizar alguna de sus instrucciones básicas, y se apaga automáticamente por sí mismo. Esto es un error imputable al que escribió el programa y es mejor apagarse que comenzar a hacer más tonterías. Las instrucciones que pueden ocasionar un apagón son: `move`, `pickbepper`.

2.2.2 Programación

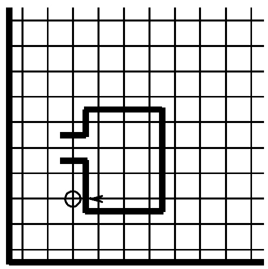
Hay varios tipos de errores de programación, estos los clasificaremos en:

- léxicos (palabras que no están en el vocabulario)
- sintácticos (se entiende, pero no está bien escrito)
- de ejecución (no puede hacer lo que dijimos, apagones)
- de intento (hace correctamente otra cosa)

2.3 Problemas

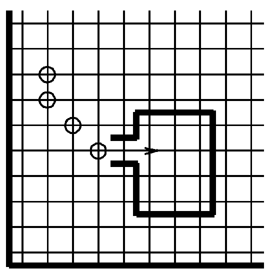
1. ¿Cuál es el programa más pequeño qué es correcto en su sintaxis?

2. Cada mañana Karel tiene que salir a recoger el periodico y regresar a la cama a leerlo.



3. Karel tiene que escalar una montaña y plantar su bandera de alpinista.

4. A Karel se le caen algunos víveres cuando se rompe su bolsa del mandado al llegar a la casa y tiene que regresar a recogerlos.



3 Extendiendo el lenguaje de Karel

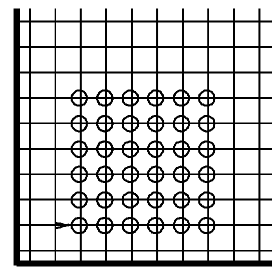
Tenemos la posiblidad de extender el limitado lenguaje de Karel definiendo instrucciones nuevas, la sintaxis es:


```
DEFINE_NEW_INSTRUCTION <nueva-instruccion> AS
<instruccion>
```

donde <instruccion> puede ser una sola instrucción o un bloque creado con {, }. Esto crea lo que se conoce como una entrada del diccionario de Karel, que es dónde está todo lo que Karel puede ejecutar. Las entradas del diccionario van separadas por ";" entre el BEGINNING_OF_PROGRAM y el BEGINNING_OF_EXECUTION, además siempre que se usa una instruccion debe haber sido definida entes.

Es bueno darles nombre a las instrucciones de acuerdo a la acción que realizan, para simplificarnos el poder entender los programas que escribimos.

Por ejemplo, para cosechar las zanahorias que Karel plantó hace tiempo



usamos definiciones de nuevas instrucciones tales como:

```
DEFINE_NEW_INSTRUCTION cosecha2filas AS
  {
    cosecha1fila;
    siguente_fila;
    cosecha1fila;
  }
```

sucesivamente definimos:

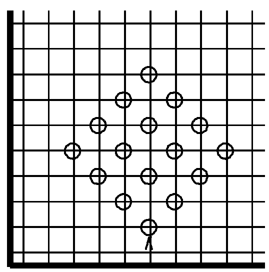
```
DEFINE_NEW_INSTRUCTION cosecha1fila AS
  {
    pickbeeper;
    move;
```

```
    pickbeeper;
    move;
    pickbeeper;
    move;
    pickbeeper;
    move;
    pickbeeper;
    move;
    pickbeeper;
  }
```

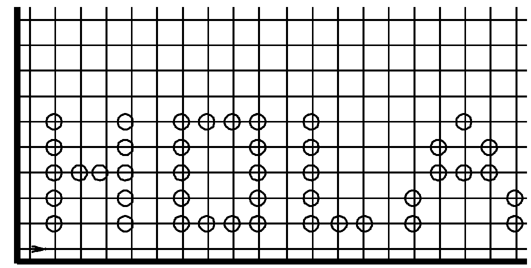
Siempre se puede escribir un programa sin instrucciones nuevas, pero resulta una larga secuencia de comandos básicos de Karel, y es muy complicado de entender, correjir, modificar, etc.

3.1 Problemas

1. El problema de Karel es ahora cosechar el siguente campo que plantó después de un partido de baseball.



2. Karel quiere mandar un mensaje a astrónomos de otros mundos para lo cual es necesario que arregle bipers de la siguente manera:



4 Ejecutando instrucciones condicionalmente

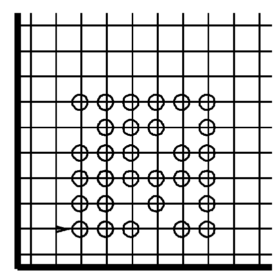
Karel necesita tener la posiblidad de decidir si va a realizar una instrucción, para eso hay dos tipos de condicionales, una es: `IF`, `THEN` le permite decidir si realiza una instrucción o no. La otra: `IF`, `THEN`, `ELSE`, le permite decidir cual debe realizar entre dos instrucciones.

4.1 La instrucción `if/then`

Por ejemplo para evitar la muerte Karel, puede checar si cada vez que intenta recoger un biper, realmente hay uno, y así definir `recoge_si_hay`. La sintaxis del `IF`, `THEN` es:

```
IF <prueba> THEN
  <instruccion>
```

Si en el problema de la cosecha de Karel, resulta que algunas de las zanahorias que plantó, no se dieron



Necesitamos modificar la definición de `cosecha1fila` sustituyendo `pickbeeper` por `recoge_si_hay`.

4.2 Ejemplos de `if/then`

A continuación tenemos dos ejemplos de como definir la instrucción: `voltea_al_norte`

```
DEFINE_NEW_INSTRUCTION voltea_al_norte AS
  {
    IF facing_east THEN
      turnleft;
    IF facing_south THEN
      {
        turnleft;
        turnleft;
      };
    IF facing_west THEN
      {
        turnleft;
        turnleft;
        turnleft;
      };
  }
```

En este caso respondemos a la pregunta ¿qué necesita hacer Karel dependiendo de en qué dirección está apuntando? Sin embargo, si comenzamos planteando la situación de otro modo, respondiendo a la pregunta ¿Necesita Karel girar a la izquierda? obetenemos:

```
DEFINE_NEW_INSTRUCTION voltea_al_norte AS
  {
    IF not_facing_north THEN
      turnleft;
    IF not_facing_north THEN
      turnleft;
    IF not_facing_north THEN
      turnleft;
  }
```


Las dos son correctas.

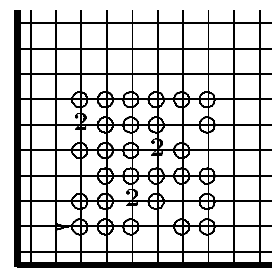
4.3 La instrucción `if/then/else`

Esta otra modalidad de condicional sirve para pode decidir entre dos posiblidades. La sintaxis es:

```
IF <prueba> THEN
  <instruccion1>
ELSE <instruccion2>
```

4.4 Instrucciones `if` anidadas

Supongamos ahora que el campo a cosechar de Karel tiene el problema de qué en algunas esquinas hay dos, en otras una y en otras ninguna zanahorias, y Karel quiere replantar el campo de tal manera que quede una sola zanahoria por esquina comenzando con una dotación suficiente de zanahorias (bipers).



La instrucción a ejecutar para la cosecha puede ser

```
DEFINE_NEW_INSTRUCTION replanta_uno AS
  {
    IF not_next_to_a_beeper THEN
      putbeeper;
    ELSE
      {
        pickbeeper;
        IF not_next_to_a_beeper THEN
```

```
          putbeeper;
        };
    }
```

En un IF, THEN, ELSE podemos invertir el orden de las instrucciones siempre y cuando la prueba que hagamos sea la opuesta. Además, si al final de ambos bloques de instrucciones tenemos una(s) instruccione(s) común(es) podemos *factorizar por abajo*, para factorizar por arriba debemos verificar que lo que factorizamos no afecte la prueba que estamos haciendo.

4.5 Problemas

1. Suponer que Karel debe apuntar al norte si está en una esquina con un biper y apuntar al sur si en la esquina hay dos bipers, y después, dejar los bipers tal y como estaban.

2. Ahora Karel debe poder decidir si está completamente rodeado de paredes y si así es, debe apagarse, en caso contrario, debe quedar funcionando y apuntando en la misma dirección que lo hacía originalmente.

3. ¿Qué es lo que hace la siguente instrucción misteriosa?

```
DEFINE_NEW_INSTRUCTION misteriosa AS
{
  IF facing_west THEN
    {
      move;
      turnright;
      IF facing_north THEN
        move;
      turnleft;
      turnleft;
    };
  ELSE
    {
      move;
      turnleft;
      move;
```



```
          turnleft;
          turnleft;
        };
    }
```

5 Instrucciones que se repiten

Con las instrucciones que producen ciclos o "loops" completamos todo el lenguaje de Karel, éstas permiten que Karel relice cierto número de veces determinada instrucción.

5.1 Las instrucciones `iterate`, `while`

En lugar de escribir una instrucción repetidas veces podemos decirle a Karel exactamente el número de veces que queremos que repita la instrucción con la siguente sintaxis:

```
ITERATE <numero> TIMES
  <instruccion>
```

Por ejemplo:

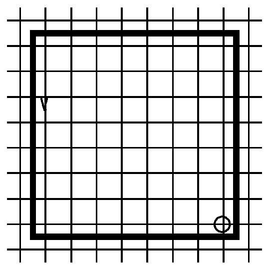
```
DEFINE_NEW_INSTRUCTION turnright AS
  {
    ITERATE 3 TIMES
      turnleft;
  }
```

Sin embargo, hay muchas situaciones en las que no sabemos el número de veces que necesitaremos ejecutar una instrucción, pero sabemos que debemos ejecutarla en tanto se cumple cierta *prueba*. Para esto tenemos:

```
WHILE <prueba> DO
  <instruccion>
```

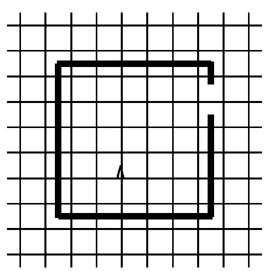
Lo que tenemos aquí es que Karel realiza la instrucción siempre y cuando se cumpla la prueba repetidas veces hasta que la prueba ya no se cumpla.

Con los elementos que tenemos, podemos resolver problemas, como el de buscar un biper junto a la pared, en un cuarto que no sabemos de que dimensiones es y dónde soló sabemos que, Karel tiene la pared a su derecha.



```
DEFINE_NEW_INSTRUCTION encuentra_biper AS
  {
    WHILE not_next_to_a_beeper DO
      {
        IF front_is_clear THEN
          move;
        ELSE turnleft;
      };
  }
```

El problema es ahora hacer que Karel encuentre la puerta de un cuarto y hacer que se asome a ella.



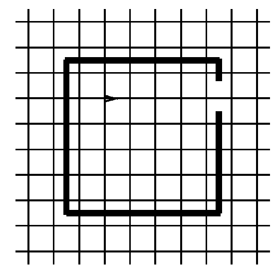
Mediante un programa del tipo:

```
BEGGINING_OF_EXECUTION
  busca_pared;
```

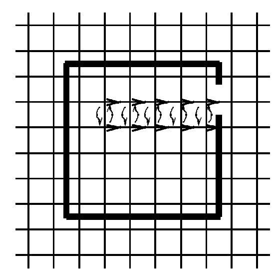


```
  turnleft;
  busca_salida;
  salte;
END_OF_EXECUTION
```

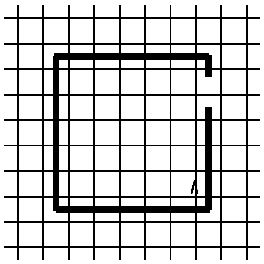
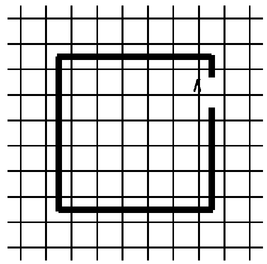
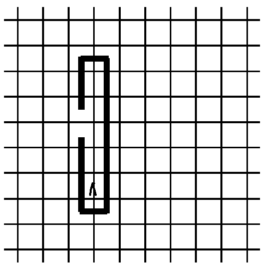
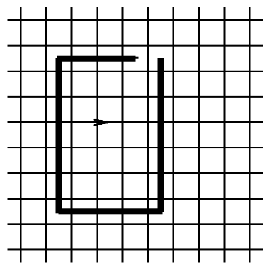
Y por medio de aproximaciones sucesivas, ir refinando el programa al probar, lo que hace Karel en la siguente situación inicial



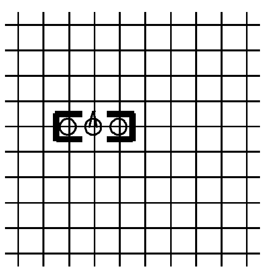
esto nos hace ver la necesidad de una instrucción `baila` como parte de `busca_pared`



después probarlo en las situaciones:

Otro problema es el del montón infinito de bipers del que Karel debe huir. Siguendo la pista que esta en el montón sobre el que está parado, Karel puede saber: el montón de la izquierda es el infinito, si el número de bipers en la pista es par y el de la derecha si es impar.



6 Técnicas avanzadas

Hay varias técnicas que hacen al lenguaje mucho más poderoso de lo que hasta ahora lo hemos usado. Mencionaremos recursión y búsqueda.

6.1 Recursión

Hemos visto otras dos versiones de la siguente instrucción:

```
DEFINE_NEW_INSTRUCTION voltea_al_norte AS
  {
    WHILE not_facing_north DO
      turnleft;
  }
```

a continuación tenemos una cuarta posiblidad

```
DEFINE_NEW_INSTRUCTION voltea_al_norte AS
  {
    IF not_facing_north THEN
      BEGIN
        turnleft;
        voltea_al_norte;
      };
  }
```

Este es un ejemplo de recursión.

Recursión sirve para resolver el siguente problema: Karel sabe que en la dirección este, de dónde se encuentra (no sabemos en que dirección está apuntando actualmente) hay un biper. Lo que debe hacer es encontrar ese biper y luego caminar hacia el norte el mismo número de pasos que caminó para encontrar el biper para así llegar a la mina perdida.

```
DEFINE_NEW_INSTRUCTION busca_mina AS
  {
    IF not_next_to_a_beeper THEN
      {
        move;
        busca_mina;
```

```
      move;
    };
  ELSE turnleft;
}
```

6.2 Búsqueda

Hay dos comandos que son fáciles de definir, y que tienen particular utilidad: `zig_left_up` y `zag_down_right` con movimientos de tipo zig-zag es posible encontrar la frontera oeste y la frontera sur respectivamente. Con esto y movimientos diagonales en el plano donde vive Karel podemos "barrer" todo el plano en busca de un biper por ejemplo.

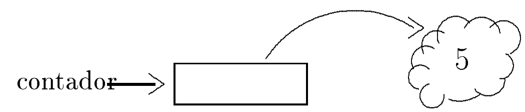
7 Java Applets

Antes de poder usar un lenguaje profesional de programación, necesitamos introducir dos conceptos más: una *variable* y un *objeto*.

7.1 Variables

Tomamos como punto de partida el biperbag de Karel. En cualquier momento se encuentra un número determinado de bipers en su biperbag. Será eficiente que Karel lleve un *registro* de cuantos hay. Con ese fin va a *apuntar* en algun lugar un número entero (positivo).

Podemos llamar este registro la *variable* contador cuyo valor es un número entero. Lo podriamos representar de la siguiente manera:



La caja indica el lugar físico donde Karel va a apuntar el número (puede ser un papel, algunos bits en una memoria electrónica, etc. para nosotros no importa la implementación específica). Llamamos la nube el *valor* de la variable.

En general restringimos los posibles valores que una variable puede tomar. En el ejemplo del biperbag, es evidente que el valor nunca puede ser 3.24, $\sqrt{23}$, "AE", etc. Llamamos *integer* la familia de todos los numeros naturales. Indicar a que familia partenece una variable (por ejemplo contador), lo hacemos a través de una frase como:

```
int contador;
```

Otras familias son `boolean` (dos valores: `false` (falso) y `true` (verdadero)), `real` (el valor es un número real) y `string` (el valores es una cadena de caracteres, por ejemplo "tgfgwefw", donde usamos para indicar que no nos referimos a la variable tgfgwefw). El siguiente ejemplo muestra el uso:

```
int contador; int contador2;
contador=7;
contador2=contador*2 + 3;
```

No olvides asignar un valor a una variable antes de usarla! No olvides declarar un variable antes de usarla !

El concepto *variable* permite definir operaciones que dependen de un parámetro. Por ejemplo, para putbeeper, podemos añadir una variable que indica cuantos bipers poner.

```
int cantidad;
cantidad=7;
putbeeper(cantidad);
putbeeper(cantidad+3);
```

7.2 Objetos

Hasta ahorita la manera de construir un programa fue siempre a través de la elaboración de todas las instruciones que Karel va a tener que ejecutar para resolver una cierta tarea (eventualemnte agrupadando las instrucciones en funciones para tener más claridad).

Existe otro enfoque. Con ese fin considera que el mundo de Karel consiste de *objetos* y cada uno partenece a una cierta familia: las paredes verticales y horizontales son miembros de la familia `Pared`, los bipers de la familia `Beeper`, karel es miembro de la familia `Robot`, etc. Para crear un *miembro* de una familia de objetos particular, usamos la siguiente estructura:

```
Karel = new Robot();
Clodomiro = new Robot();
pared1= new Pared;
```

A cada familia de objetos podemos atribuir ciertas características y operaciones (métodos). Por ejemplo, todos los miembros de la familia `Robot` pueden dar un paso adelante a través del comando `move` (igualmente un `turnleft`, etc.). Lo denotamos así:

```
Karel = new Robot();
Clodomiro = new Robot();
Karel.move();
Karel.turnleft();
Clodomiro.move();
```


Como se puede ver, contrario a variables, en la notación de funciones se usa siempre '()' al final.

Es muy frecuente que ordenemos las familias seguún los métodos que tienen en común. `Paredes` y `Beepers` (junto con `Fuentes`, `Luces`, etc.) son todos `Articulos`. Tienen en común un método para colocarlos en un lugar específico. Por otro lado, la clase `Beeper` tendrá un método para indicar el volumen del tono, `Pared` algo para cambiar su grosor o orientación, etc. El siguiente ejemplo muestra un caso hipotético:

```
public class Robot extends Maquinas {
   public void move() {
      /* aqui el codigo para establecer un paso adelante */
   };
   public void sleep(int x) {
      /* aqui el codigo para que duerme x segundos */
   };
   /* etc. etc. */
}
```

7.3 Applets

A continuación damos algunos ejemplos:

7.3.1 Escribir un texto y cambiar el color del fondo

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
import java.applet.Applet;

public class ColorHelloApplet extends Applet {
  Font f = new Font("TimesRoman",Font.BOLD,36);
  public void paint(Graphics g) {
    g.setFont(f);
    g.setColor(Color.red);
    g.drawString("Welcome to Java Programming!!!", 5, 50);
  }
}
```

7.3.2 El usuario elige el color de fondo 1

```
import java.awt.*;
import java.applet.*;

public class ChoiceTest extends Applet {
  Choice colorChoice = new Choice();
  public void init() {
    setBackground(Color.red);
    colorChoice.addItem("Red");
    colorChoice.addItem("Yellow");
    colorChoice.addItem("Green");
    colorChoice.addItem("Blue");
    colorChoice.addItem("White");
    colorChoice.addItem("Black");
    add(colorChoice);
  };
  public boolean action(Event evt, Object arg) {
    if (evt.target == colorChoice) {
      if (((String)arg).equals("Red"))
        setBackground(Color.red);
      else if (((String)arg).equals("Yellow"))
        setBackground(Color.yellow);
      else if (((String)arg).equals("Green"))
        setBackground(Color.green);
      else if (((String)arg).equals("Blue"))
        setBackground(Color.blue);
      else if (((String)arg).equals("White"))
        setBackground(Color.white);
      else
        setBackground(Color.black);
      repaint();
      return true;
    }
    return false;
  };
}
```


7.3.3 El usuario elige puede elegir el color de fondo 2

```
import java.applet.*;
import java.awt.*;

public class ButtonApplet5 extends Applet {

    Button redButton    = new Button("Red");
    Button yellowButton = new Button("Yellow");
    Button orangeButton = new Button("Orange");
    Button blackButton  = new Button("Black");
    Button blueButton   = new Button("Blue");
    Button whiteButton  = new Button("White");

    Panel colorPanel;
    Panel buttonPanel;

    public void init() {

        setLayout(new GridLayout(2, 1));  // Applet has
                                          // 2 row, 1 cols

        buttonPanel = new Panel();
        buttonPanel.setLayout(new GridLayout(2, 3));
        buttonPanel.add(redButton);
        buttonPanel.add(yellowButton);
        buttonPanel.add(orangeButton);
        buttonPanel.add(blackButton);
        buttonPanel.add(blueButton);
        buttonPanel.add(whiteButton);

        colorPanel = new Panel();
        colorPanel.setBackground(Color.white);

        add(colorPanel);      // add colorPanel to applet
        add(buttonPanel);     // add buttonPanel to applet


    }
```

```
    public boolean action(Event evt, Object arg) {
       if (evt.target instanceof Button) {
          if (evt.target == redButton)
             colorPanel.setBackground(Color.red);
          else if (evt.target == yellowButton)
             colorPanel.setBackground(Color.yellow);
          else if (evt.target == orangeButton)
             colorPanel.setBackground(Color.orange);
          else if (evt.target == blackButton)
             colorPanel.setBackground(Color.black);
          else if (evt.target == blueButton)
             colorPanel.setBackground(Color.blue);
          else
             colorPanel.setBackground(Color.white);
          colorPanel.repaint();
          return true;
       }
       return false;
    }
}
```

7.3.4 El usuario elige el fondo de un texto

```
import java.awt.*;
import java.applet.*;

public class RadiobuttonTest extends Applet {

  CheckboxGroup cbg = new CheckboxGroup();
  Checkbox small = new Checkbox("Small", cbg, true);
  Checkbox medium = new Checkbox("Medium", cbg, false);
  Checkbox large = new Checkbox("Large", cbg, false);
  Checkbox extraLarge = new Checkbox("Extra Large", cbg, false);
  int fontSize = 10;

  public void init() {
```



```
    small.setBackground(Color.yellow);
    medium.setBackground(Color.yellow);
    large.setBackground(Color.yellow);

    add(small);
    add(medium);
    add(large);
    add(extraLarge);
    extraLarge.setBackground(Color.yellow);
    setBackground(Color.yellow);
  }

  public boolean action(Event evt, Object arg) {
    if (evt.target instanceof Checkbox) {
       if (evt.target == small)
          fontSize = 10;
       else if (evt.target == medium)
          fontSize = 16;
       else if (evt.target == large)
          fontSize = 24;
       else
          fontSize = 30;
       repaint();
       return true;
    }
    return false;

  }

  public void paint(Graphics g) {

    Font f = new Font("TimesRoman", Font.BOLD, fontSize);
    g.setFont(f);
    g.drawString("I love Radiobuttons!!!!", 5, 95);
  }

}
```

7.3.5 Mover un caracter en la pantalla através de las flechas

```
import java.awt.*;
import java.applet.*;

public class Keys extends Applet {

  char currkey = 'Q';      // set initial value to Q
  int currx;
  int curry;

  public void init() {

    currx = size().width / 2;   // start character in center
    curry = size().height / 2;
    setBackground(Color.green);
    setFont(new Font("Helvetica",Font.BOLD,36));
    requestFocus();             // request the input focus
  }

  public boolean mouseDown (Event evt , int  x, int y) {
  curry = y;
  currx = x;
  return true;
  }


  public boolean keyDown(Event evt, int key) {
    switch (key) {
    case Event.DOWN:
      curry += 5;
      break;
    case Event.UP:
      curry -= 5;
```



```
      break;
    case Event.LEFT:
      currx -= 5;
      break;
    case Event.RIGHT:
      currx += 5;
      break;
    default:
      currkey = (char)key;
    }

    repaint();
    return true;
  }


  public void paint(Graphics g) {
      g.drawString(String.valueOf(currkey), currx,curry);
  }
}
```

7.3.6 Escribir y leer texto

```
import java.awt.*;
import java.applet.*;

public class TextTest extends Applet {


   Panel p;
   TextArea ta;
   TextField from;
   TextField to;
   Button replace = new Button("Replace");

   public void init() {
```

```
      setLayout(new BorderLayout());

      p = new Panel();
      p.setLayout(new FlowLayout());
      p.add(replace);
      from = new TextField(10);
      p.add(from);
      p.add(new Label("with"));
      to = new TextField(10);
      p.add(to);

      add("South", p);
      ta = new TextArea(8,40);
      add("Center", ta);
   }


   public boolean action(Event evt, Object arg) {
      if (arg.equals("Replace")) {
          String f = from.getText();
          int n = ta.getText().indexOf(f);
          if (n >= 0 && f.length() > 0)
          ta.replaceText(to.getText(), n, n + f.length());
          return true;
      }
      return false;
   }
}
```