Chapter 11. Detecting User Input

A program can respond to the keyboard and mouse by utilizing Java's numerous java.io input stream and reader classes, or the key and mouse listeners in java.awt.event. So, what's the need for J/Invoke?

The crucial limitation is that key and mouse data can only be collected if the Java application is the *active* window. There's (almost) no way for a Java program to listen to input being directed towards other windows or to the Desktop.

This ability to monitor (or if you prefer, 'snoop on') user activity is very useful for a wide range of applications, including contextual help, screensavers, and logging tools.

Hacks 11-1 and 11-2 are concerned with monitoring the *position* of the mouse and moving it, no matter where it is on screen. This functionality is supported by Java's MouseInfo and Robot classes. We start using J/Invoke in Hack 11-3 to poll for mouse and keyboard *state* information. Hack 11-4 is about detecting the *absence* of user activity (i.e. no mouse clicks, moves, or keypresses).

The Hack 11-3 technique isn't ideal due to the difficulty of determining a good polling interval. Hacks 11-5 and 11-6 introduce Win32 keyboard and mouse *hooks*, which trigger callback functions when a key is pressed or the mouse state changes. Hack 11-6 also shows how a clicked mouse position can be converted into more useful information: the name of the window activated by the click.

The last two hacks are concerned with low-level *signal handling* triggered by the user pressing ctrl-c or ctrl-break: Hack 11-7 discusses how termination signals can be caught, while Hack 11-8 looks at modifying the meaning of a ctrl-break signal.

Hack 11.1 Where's the Mouse (low level)

Use a thread to poll the mouse position.

Java's MouseInfo class can return the current screen position of the mouse in a PointerInfo object. However, the object is only filled with data when MouseInfo.getPointerInfo() is called. This requirement makes it natural to utilize polling in a thread to keep up with the mouse's travels.

The run() method given below is from CursorWin.java. It reports the mouse's position relative to the screen, it's position relative to the CursorWin JFrame window, and determines whether the mouse is currently inside that window.

```
// global
private Rectangle winRect;    // same size as the window

public void run()
{
  Point pos;
  PointerInfo info;
  while (true) {
    info = MouseInfo.getPointerInfo();   // must be called repeatedly
    pos = info.getLocation();
    System.out.println("Screen pos: (" + pos.x + ", " + pos.y + ")");
```



```
    SwingUtilities.convertPointFromScreen(pos, this);
    System.out.println("Window pos: (" + pos.x + ", " + pos.y +
                "); inside: " + winRect.contains(pos.x, pos.y));

    try {
      Thread.sleep(200);   // 0.2 secs sleep
    }
    catch(InterruptedException e) {}
  }
}  // end of run()
```

Typical output from CursorWin:

```
> java CursorWin
Screen pos: (724, 371)
Window pos: (65, -39); inside: false
Screen pos: (698, 373)
Window pos: (39, -37); inside: false
Screen pos: (709, 402)
Window pos: (50, -8); inside: false
Screen pos: (691, 430)
Window pos: (32, 20); inside: true
Screen pos: (693, 435)
Window pos: (34, 25); inside: true
Screen pos: (700, 431)
    :
```

The Point data stored in the PositionInfo object uses screen coordinates (i.e. the top-left corner of the screen is at (0, 0)), but these can be easily converted into window coordinates with SwingUtilities.convertPointFromScreen(). Window coordinates start at (0, 0) in the top-left corner of the window, which includes the title bar.

Once window-specific coordinates are obtained, the cursor is tested against a Rectangle object representing the size of the window. The rectangle is generated one-time only in the CursorWin constructor since the window cannot be resized:

```
// in CursorWin()
winRect = new Rectangle( getWidth(), getHeight());
```

MouseInfo's main drawback is that it can't observe the mouse's state (i.e. which buttons are being pressed).

Hack 11.2 Moving the Mouse Cursor (low level)

Prevent the mouse from entering a window by moving it's cursor away if it gets too close.

The CursorRepel.java application described in this hack was written a few days after April Fools Day, which may explain its rather silly functionality. The window contains a "Click Me" button (see Figure 11-1), but it's virtually impossible to click on the button. If the mouse cursor gets too close, it's pushed back to the nearest window edge.

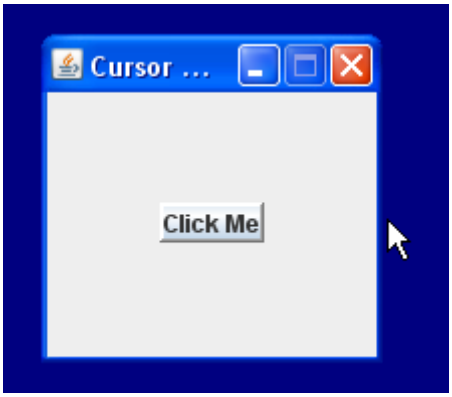

Figure 11-1. Repelling the Mouse.

The repellent nature of the window extends over its complete surface, so its also very difficult to click on the close box to terminate the application. The easier option is to close the program via its icon on the task bar.

Although silly, the program illustrates how MouseInfo can get the position of the mouse, and how Java's Robot class can move the mouse cursor.

The Robot class has three main areas of functionality: it can control the mouse cursor (which includes moving it, generating press and release events, and mouse wheel changes), it can activate the keyboard (i.e. generate press and release key events), and it can taking screenshots of the desktop. Unfortunately, Robot doesn't have any way of reading the current state of the mouse or keyboard.

Serious uses of the Robot class include animated program demonstrations (Hack #90 in *Swing Hacks* by Joshua Marinacci and Chris Adamson), creating non-rectangular/transparent windows (Hack #41, *Swing Hacks*), and color picking from screen pixels (Hack #59, *Swing Hacks*).

The run() method in CursorRepel.java is essentially an extension of the one used in CursorWin.java in Hack 11.1:

```
// global
private Rectangle winRect;   // same size as window

public void run()
// keep the cursor outside the window
{
  Point pos;
  while (true) {
    pos = MouseInfo.getPointerInfo().getLocation();
    SwingUtilities.convertPointFromScreen(pos, this);
    if (winRect.contains(pos.x, pos.y))
      repelCursor(pos);
    try {
      Thread.sleep(200);
    }
    catch(InterruptedException e) {}
  }
}  // end of run()
```


repelCursor() calculates the distance of the cursor from the four edges of the window, selects the shortest, and moves the cursor to that edge using Robot.mouseMove().

```
// global
private Robot robot;    // for moving the cursor

private void repelCursor(Point pos)
// move cursor to nearest window edge
{
  // get the distances from the four edges
  int topDist = pos.y;
  int bottomDist = winRect.height - pos.y;
  int leftDist = pos.x;
  int rightDist = winRect.width - pos.x;

  // find the smallest distance, and calculate the edge position
  Point edgePos = . . . // calculation not shown;

  robot.mouseMove(edgePos.x, edgePos.y);
}  // end of repelCursor()
```

The calculation of the new edge position consists of a series of if-tests to determine the shortest distance, followed by the calculation of a new Point object. Interested readers can see the details in CursorRepel.java.

The Robot object is created in CursorRepel's constructor:

```
// in CursorRepel()
try {
  robot = new Robot();
}
catch(AWTException e) {}
```

Hack 11.3 Polling the Mouse and Keyboard (medium level)

Use J/Invoke to obtain mouse and keyboard state information.

The drawback of the MouseInfo and Robot classes is that they don't allow the keyboard or mouse state to be read. Of course, such state information is available to a Java application through input streams, readers, and listeners, but only if the application is the active window. If we want to monitor user activity when the window focus is elsewhere, then the Win32 API is needed.

J/Invoke can obtain the mouse and keyboard states with a single Win32 method: GetAsyncKeyState() from user32.dll (documented at http://msdn2.microsoft.com/en-us/library/ms646293(VS.85).aspx). GetAsyncKeyState() is called with the virtual key code of the key (or button) that we want to check, and the function returns whether the key (or button) was pressed since the last call to GetAsyncKeyState(), and whether it's currently up or down.

Our PollingWin.java example utilizes polling inside a thread:

```
// globals
private int hwnd;   // handle for the window
private Rectangle winRect;    // same size as the window
```



```
public void run()
{
  Point pos = new Point();    // a JInvoke Point object;
  while (true) {
    User32.GetCursorPos(pos);
    System.out.println("Screen pos: (" + pos.x + ", " + pos.y + ")");
    User32.ScreenToClient(hwnd, pos);

    System.out.println("Window pos: (" + pos.x + ", " + pos.y +
                "); inside: " + winRect.contains(pos.x, pos.y));

    System.out.println("Mouse state [L,M,R]: [" +
              isPressed(WinConstants.VK_LBUTTON) + "," +
              isPressed(WinConstants.VK_MBUTTON) + "," +
              isPressed(WinConstants.VK_RBUTTON) + "]");
    int ch = getLetter();
    if (ch != 0)
      System.out.println("Letter: " + (char)ch);

    if (isFinished())
      System.exit(0);

    try {
      Thread.sleep(200);
    }
    catch(InterruptedException e) {}
  }
}  // end of run()
```

run() prints out the mouse coordinates relative to the screen and window, as in Hack 11-1. However, we've used the Win32 functions GetCursorPos() and ScreenToClient() from user32.dll to do it here.

Typical output from PollingWin consists of:

```
> java -cp d:\jinvoke\jinvoke.jar;. PollingWin
Screen pos: (153, 422)
Window pos: (-509, -17); inside: false
Mouse state [L,M,R]: [false,false,false]
Letter: g
Screen pos: (153, 422)
Window pos: (-509, -17); inside: false
Mouse state [L,M,R]: [false,false,false]
Letter: s
Screen pos: (153, 422)
Window pos: (-509, -17); inside: false
Mouse state [L,M,R]: [true,false,false]
Screen pos: (150, 420)
Window pos: (-512, -19); inside: false
Mouse state [L,M,R]: [false,false,true]
Letter: G
Screen pos: (150, 420)
Window pos: (-512, -19); inside: false
Mouse state [L,M,R]: [false,false,false]
  :
```


The "Screen pos" and "Window pos" lines are similar to the output from CursorWin.java in Hack 11-1. There's also additional information on the state of the left, middle, and right-hand mouse buttons (the [L,M,R] lines), and the letter if a key is pressed.

There's a subtle difference between Java's and Win32's screen-to-window coordinates conversion. Java's SwingUtilities.convertPointFromScreen() locates (0,0) at the top-left corner of the window (including the title bar), whereas User32.ScreenToClient() puts (0,0) at the top-left corner of the window's panel (*excluding* the title bar).

The mouse button state is obtained using isPressed() which calls GetAsyncKeyState() from user32.dll:

```
private boolean isPressed(int key)
// is key (or mouse button) pressed down?
{ return ((User32.GetAsyncKeyState(key) & 0x8000) == 0x8000);  }
```

GetAsyncKeyState() returns an integer holding information on whether the supplied key is up or down, and whether it's been pressed since the previous call to GetAsyncKeyState(). However, the Win32 documentation warns us against relying on the correctness of the "was pressed since" data, which is stored in the least significant bit of the integer. Consequently, isPressed() only checks the "key is up or down" information stored in the most significant bit (0x8000).

Obtaining a letter is a little bothersome since we must check all the letter keys *and* the shift key. If the shift key *isn't* pressed then the letter key is converted into its lower-case letter ASCII value.

```
private int getLetter()
// return a letter (in its ASCII form), or 0
{
  boolean isShifted = isShifted();
  for (int key = 65; key <= 90; key++) {   // ASCII range of A-Z
    if (isPressed(key)) {
      if (isShifted)
        return key;
      else  // not shifted
        return (key+32);    // convert key to ASCII range a-z
    }
  }
  return 0;
}  // end of getLetter()


private boolean isShifted()
// check if the left or right hand shift key has been pressed
{
  return (isPressed(WinConstants.VK_SHIFT) ||
          isPressed(WinConstants.VK_RSHIFT));
}
```

isShifted() checks both shift keys on the keyboard.

The isFinished() method is a convenient way of stopping PollingWin without having to click on its close-box. isFinished() returns true if it detects that ctrl-c, ESC, or END have been pressed, which makes the thread exit.

```
private boolean isFinished()
// check for ctrl-c, ESC, and END to indicate things are finished
{
  if (isPressed(WinConstants.VK_CONTROL) && isPressed(67)) {
    System.out.println("ctrl-c");            // 67 == C key
    return true;
  }
  if(isPressed(WinConstants.VK_ESCAPE)) {
    System.out.println("ESC");
    return true;
  }
  if(isPressed(WinConstants.VK_END)) {
    System.out.println("END");
    return true;
  }
  return false;   // not finished
}  // end of isFinished()
```

The trickiest aspect of isFinished() are the names of the virtual key constants. They are listed at http://msdn2.microsoft.com/en-us/library/ms927178.aspx

One problem with GetAsyncKeyState() is that it only reports on mouse button presses and releases; there's no way to obtain rotation information from the mouse wheel.

Another problem is the polling rate in run() (key tests followed by a sleep of 200ms), which is a bit slow to catch rapidly pressed keys. Decreasing the sleep time has the drawback of increasing the polling workload.

The solution for both issues is to employ Win32 *hooks*, which are the topic of Hacks 11-5 and 11-6.

Hack 11.4 Detecting No Activity (low level)

Determine how long it is since the user pressed any keys or touched the mouse.

Detecting the absence of user activity is a useful way of triggering system utilities, such as screensavers and alarms, and initiating background jobs which are better left until the machine is idle, such as virus checking and defragmentation.

The trick is to utilize Win32's LASTINPUTINFO structure (see http://msdn2.microsoft.com/en-us/library/ms646272.aspx), which holds the tick count when the last user input event was received. The tick count is the number of milliseconds that have elapsed since the system was started, up to a maximum of 49.7 days. By periodically obtaining a new LASTINPUTINFO structure, and comparing its tick count to the current time, we can determine how long the user has been idle.

Idler.java reports the user's idle time: a cumulative value (in ms) since the user last touched the keyboard or mouse. A report is printed to stdout roughly every second. An example is shown below:


```
> java -cp d:\jinvoke\jinvoke.jar;. Idler
User idle for: 266 ms
User idle for: 1266 ms
User idle for: 2266 ms
User idle for: 3266 ms
User idle for: 4266 ms
User idle for: 5266 ms
User idle for: 797 ms
User idle for: 1797 ms
User idle for: 2797 ms
User idle for: 3797 ms
User idle for: 4797 ms
   :
```

The output shows that the user did nothing for about 5 seconds (5266 ms), and then either pressed a key or moved the mouse, causing the cumulative idle time to be reset.

Idler.java is another polling application, this time of the LASTINPUTINFO structure. However, since Idler has no GUI interface, the test-sleep loop is implemented inside main():

```
public static void main(String[] args)
{
  JInvoke.initialize();
  while (true) {
    System.out.println("User idle for: " + getIdleTime() + " ms");
    try {
      Thread.sleep(1000); // 1 sec
    }
    catch(InterruptedException e) {}
  }
} // end of main()
```

getIdlerTime() calculate the idle time as the difference between the last user input time and the current tick time.

The LASTINPUTINFO structure is retrieved using GetLastInputInfo() from User32.dll (see http://msdn2.microsoft.com/en-us/library/ms646302.aspx). Unfortunately, this isn't supplied in the User32.java helper class in the version of J/Invoke that we're using, so it must be defined, as must the Java version of LASTINPUTINFO.

The LASTINPUTINFO struct is defined at http://msdn2.microsoft.com/en-us/library/ms646272.aspx as:

```
typedef struct LASTINPUTINFO {
  UINT cbSize;    // size of structure
  DWORD dwTime;   // time of last input
} LASTINPUTINFO;
```

This is mapped easily into a LASTINPUTINFO class:

```
@NativeStruct
public class LASTINPUTINFO {
  public int cbSize;
```



```
  public int dwTime;
}
```

GetLastInputInfo() expects a call-by-reference argument to an empty LASTINPUTINFO struct, which it fills in. Fortunately, J/Invoke handles struct arguments using call-by-reference by default. The resulting code is located in getIdleTime():

```
// global
@NativeImport(library="User32")
public static native boolean GetLastInputInfo(LASTINPUTINFO[] info);


private static int getIdleTime()
{
  LASTINPUTINFO info = new LASTINPUTINFO();
  info.cbSize = Util.getStructSize(LASTINPUTINFO.class);

  int dwTime = 0;
  if (GetLastInputInfo(info))  // fill in info
    dwTime = info.dwTime;
  else
    System.out.println("GetLastInputInfo() failed");

  return (Kernel32.GetTickCount()- dwTime);  // curr. time–last input
}  // end of getIdleTime()
```

The Util.getStructSize() utility method is used to fill in the cbSize field of the empty LASTINPUTINFO struct, and the call to GetLastInputInfo() stores the last input time in its dwTime field.

The cumulative idle time is calculated by subtracting the LASTINPUTINFO time from the current tick count value, obtained with GetTickCount() from kernel32.dll (GetTickCount() is described at http://msdn2.microsoft.com/en-us/library/ms724408.aspx).

Hack 11.5 Being Woken by the Keyboard (medium level)

Use a low-level keyboard hook and callback function to detect keyboard activity.

We've been utilizing polling to listen to the user up to now, a technique that can consume lots of resources if the polling frequency is high, and which may miss input if the frequency is too low. An alternative solution is to use Win32 *hooks* (overviewed at http://msdn2.microsoft.com/en-us/library/ms632589(VS.85).aspx).

Win32 supports many types of hooks to access (and manipulate) different kinds of system messages. For example, an application can use the WH_MOUSE hook to monitor mouse input.

The OS maintains a separate *hook chain* for each kind of hook. When a message arrives that's associated with a particular hook type, the message is sent to each *hook procedure* in the hook chain, one after the other.

A new hook procedure can be added to the front of a hook chain by calling SetWindowsHookEx() from user32.dll. There are two main kinds of procedure: a *global hook procedure* monitors messages being sent to all threads associated with the same desktop as the hook thread. A *thread-specific hook procedure* only monitors messages aimed at that thread.

We don't need to utilize thread-specific hooks, because Java listeners do much the same thing, in a more high-level manner. Our need is for global hooks which can listen to all keyboard and mouse input irrespective of where it's being sent.

Global hook procedures are rather difficult animals to work with in Win32: the procedure must be placed in a DLL separate from the application installing it. The procedure's addition to the hook chain is complicated since the DLL must be injected into every application process. This means that the global hook code will probably adversely affect system performance. It's also tricky to cleanly remove the procedure from the chain, since it will be referenced by multiple processes.

Fortunately, we can avoid all of these hassles by using *low-level hooks* to hold our global keyboard and mouse hook procedures.

A procedure for a low-level keyboard hook (the WH_KEYBOARD_LL hook) doesn't need to be defined inside a DLL since it's not injected into other processes at installation time. Instead, the code is called in its original thread context, which may make it less of a system hog (although there's still the overhead of thread switching), and definitely makes it easy to uninstall the hook using UnhookWindowsHookEx() from user32.dll

The low-level mouse hook (the WH_MOUSE_LL hook) has the same useful properties.

In this hack, we'll be using the WH_KEYBOARD_LL hook, and Hack 11-6 will employ the WH_MOUSE_LL hook.

CatchKeys.java sets up the global low-level keyboard hook to call the keysProcessing() hook procedure/method whenever a key is pressed in any window or on the desktop. The method converts the keycode value to an ASCII character if possible, and prints it to standard output. After about 15 seconds, the hook procedure is automatically uninstalled and the program terminates.

Typical output from CatchKeys is shown below:

```
> java -cp d:\jinvoke\jinvoke.jar;. CatchKeys
Hook installed
Virtual key: 160
F
J
G
o
9
u
8
Virtual key: 118
Virtual key: 119
Virtual key: 116
9
Virtual key: 112
```



```
Virtual key: 113
4
Posted 'quit' to message pump thread
Hook thread has finished.
```

The “Virtual key” outputs are for keycodes without an ASCII representation, such as function keys.

The main() function of CatchKeys.java sets up the hook procedure, waits for 15 seconds, then removes it:

```
public static void main(String[] args)
{
  JInvoke.initialize();

  setHook();
  try {
    Thread.sleep(15000);   // wait while user presses the keyboard
  }
  catch (InterruptedException e) {}
  unHook();
}  // end of main()
```

Setting the Hook Procedure

setHook() creates a new thread specifically for the keyboard hook. The hook procedure will be called in the context of that thread when a key is pressed. The call is made by sending messages to that thread, which must therefore contain an implementation of a Win32 message loop. The message loop (sometimes called a message pump) reads the messages and indirectly calls the hook procedure through message dispatching.

The message loop will be kept very busy, so it makes sense to implement it in a separate thread rather than in CatchKeys’ main application thread.

```
// globals
@NativeImport(library = "user32")
public native static int SetWindowsHookEx(int idHook,
                Callback hookProc, int hModule, int dwThreadId);

private static final int WH_KEYBOARD_LL = 13; //low-level keybrd type

private static int keyboardHook;  // ID (handle) of hook procedure
private static int hookThreadId;  // ID of thread


private static void setHook()
{
  Thread hookThread = new Thread( new Runnable() {
    public void run()
    {
      int hInst = Kernel32.GetModuleHandle(null);

      // create keyboard hook callback function
      Callback kbCallback = new Callback(CatchKeys.class,
                                          "keysProcessing");
```



```
      // add hook function to OS; get back hook ID (handle)
      keyboardHook = SetWindowsHookEx(WH_KEYBOARD_LL, kbCallback,
                                                      hInst, 0);
      // store the ID of the thread managing the hook
      hookThreadId = Kernel32.GetCurrentThreadId();

      // message dispatch loop (message pump)
      System.out.println("Hook installed");
      Msg msg = new Msg();
      while (User32.GetMessage(msg, 0, 0, 0)) {
        User32.TranslateMessage(msg);
        User32.DispatchMessage(msg);
      }
      System.out.println("Hook thread has finished.");
    }
  });

  hookThread.start();
}  // end of setHook()
```

The hook procedure, keyProcessing(), which is defined later in the CatchKeys class, is converted into a J/Invoke callback object, and supplied to SetWindowsHookEx() along with the hook type (WH_KEYBOARD_LL), and the handle for this application.

SetWindowsHookEx() from user32.dll is described at http://msdn2.microsoft.com/en-us/library/ms644990(VS.85).aspx, and is prototyped in CatchKeys since it isn't part of the User32 helper class. SetWindowsHookEx() returns a handle to the hook procedure, which is stored globally along with the ID of the thread for use later in unHook().

The message loop code is a common sight in C and VB Window programs. It's possible to do various things to the message before sending it on its way with User32.DispatchMessage(), but we don't bother in this example.

Removing the Hook Procedure

unHook() stops the thread's message loop (which allows the thread to finish), and removes the hook procedure from the hook chain.

```
// globals
@NativeImport(library = "user32")
public native static int UnhookWindowsHookEx(int idHook);


private static void unHook()
{
  User32.PostThreadMessage(hookThreadId, WinConstants.WM_QUIT, 0, 0);
  System.out.println("Posted 'quit' to message pump thread");
  try {
    Thread.sleep(200);   // wait a bit while pump stops
  }
  catch (InterruptedException e) {}

  if (UnhookWindowsHookEx(keyboardHook) > 0)  // remove hook proc.
    printLastError("Keyboard Hook");
```



```
}  // end of unHook()
```

The message loop is stopped by sending it a WM_QUIT message, which is directed to the correct loop by including the thread ID in the call to PostThreadMessage(). unHook() briefly sleeps to give the thread time to complete, and then UnhookWindowsHookEx() is passed the hook procedure handle to remove it from the chain. UnhookWindowsHookEx() from user32.dll is described at http://msdn2.microsoft.com/en-us/library/ms644993(VS.85).aspx.

The Hook Procedure

The hook procedure is keyProcessing() inside the CatchKeys class (as specified by setHook()). The signature of the method must correspond to the LowLevelKeyboardProc() hook procedure format explained at http://msdn2.microsoft.com/en-us/library/ms644985(VS.85).aspx:

```
LRESULT CALLBACK LowLevelKeyboardProc(
                    int nCode, WPARAM wParam, LPARAM lParam);
```

If nCode is less than zero then there's a problem and the procedure should hand on the message to the CallNextHookEx() function without further processing. This passes the message to the next hook procedure in the hook chain. CallNextHookEx() is documented at http://msdn2.microsoft.com/en-us/library/ms644974(VS.85).aspx.

wParam is a constant identifying the general keyboard action of the message, including if it's a key press or release (represented by WM_KEYDOWN and WM_KEYUP).

lParam is a pointer to a KBDLLHOOKSTRUCT structure which holds more details about the key. It's described at http://msdn2.microsoft.com/en-us/library/ms644967(VS.85).aspx.

keyProcessing() uses its input information to print the key in a nice format, either as an ASCII Value or a keycode constant:

```
// globals
@NativeImport(library = "user32")
public native static int CallNextHookEx(int idHook, int nCode,
                                         int wParam, int lParam);


public static int keysProcessing(int nCode, int wParam, int lParam)
{
  if (nCode < 0)  // pass message to next hook procedure
    return CallNextHookEx(keyboardHook, nCode, wParam, lParam);

  if((nCode == WinConstants.HC_ACTION) &&
     (wParam == WinConstants.WM_KEYDOWN)) {    // a key was pressed
    KBDllHookStruct keyInfo =
                Util.ptrToStruct(lParam, KBDllHookStruct.class);
                      // convert lParam into keyboard info.

    // try to print the key nicely
    boolean wasPrinted = printSpecialKey(keyInfo.vkCode);
    if (!wasPrinted)
```



```
      wasPrinted = printASCIIKey(keyInfo);
    if (!wasPrinted)
      System.out.println("Virtual key: " + keyInfo.vkCode);
  }

  // pass message onto next hook proc. for further processing
  return CallNextHookEx(keyboardHook, nCode, wParam, lParam);
}  // end of keysProcessing()
```

The general form of all hook procedures is much the same. The method starts by checking nCode, and passes the message onto the next hook procedure if nCode is less than 0. At the end of the procedure, the message is also sent on to the next hook procedure, to ensure that it will eventually be delivered to its intended application.

The middle part of keysProcessing() checks the nCode and wParam constants to respond only to a key press, and the wParam pointer (a 32 bit integer) is converted into a KBDLLHOOKSTRUCT struct using Util.ptrToStruct().

printASCIIKey() examines the complete keyboard state so that the effects of <shift>, <ctrl>, and other modifier keys are correctly applied to the virtual key in the message. For example, if <shift> and <a> were pressed together, then the ASCII 'A' character should be printed.

```
private static boolean printASCIIKey(KBDllHookStruct keyInfo)
{
  User32.GetKeyState(0);   // triggers update of GetKeyboardState()

  byte[] keyState = new byte[256];
  User32.GetKeyboardState(keyState);
    // use keyboard state to deal with shift and other modifier keys

  short[] wBuf = new short[1];
  int numChars = User32.ToAscii(keyInfo.vkCode, keyInfo.scanCode,
                                    keyState, wBuf, 0);
        // translates virtual key and keyboard state to char(s)
  if (numChars == 1) {
    System.out.println( ((char)wBuf[0]) );  // ASCII value
    return true;
  }
  return false;
}  // end of printASCIIKey
```

The keyboard state (a byte array) is initialized with User32.GetKeyboardState(), and passed to User32.ToAscii() along with the virtual key details. The resulting character is accessible via the wBuf pointer (which is represented as a single element array in J/Invoke). GetKeyboardState() and ToAscii() are explained at http://msdn2.microsoft.com/en-us/library/ms646299(VS.85).aspx and http://msdn2.microsoft.com/en-us/library/ms646316(VS.85).aspx.

Hack 11.6 Being Woken by the Mouse (medium level)

Use a low-level mouse hook and callback function to detect mouse activity.

The steps for using a low-level mouse hook are almost the same as those for a low-level keyboard hook described in Hack 11.5. I won't explain everything again, so if you've come straight to this hack, you should read Hack 11.5 first.

Setting up a global mouse hook is quite complicated due to Win32's insistence that it be placed in a DLL, and then be injected into every process running on the desktop. Fortunately, we can avoid these problems by using a low-level mouse hook (the WH_MOUSE_LL hook).

The CatchMouse.java example sets up a low-level mouse hook procedure called mouseProcessing(), which is triggered whenever the mouse is used. The method reports on the button states (pressed or not), the cursor position, and the rotation of the mouse wheel (a feature not supported by User32.GetAsyncKeyState() in Hack 11.3).

CatchMouse.java is very similar to CatchKeys.java in Hack 11.5 except for the implementation details of mouseProcessing().

Sample output from CatchMouse.java is shown below:

```
> java -cp d:\jinvoke\jinvoke.jar;. CatchMouse
Hook installed
Mouse pt: (113, 113) Mouse moved
Mouse pt: (112, 112) Mouse moved
Mouse pt: (111, 111) Mouse moved
Mouse pt: (111, 111) Left button down
Mouse pt: (111, 111) Left button up
Mouse pt: (111, 111) Left button down
Mouse pt: (111, 111) Left button up
Mouse pt: (111, 111) Left button down
Mouse pt: (111, 111) Mouse moved
Mouse pt: (110, 110) Mouse moved
Mouse pt: (109, 109) Mouse moved
Mouse pt: (108, 108) Mouse moved
Mouse pt: (107, 107) Mouse moved
Mouse pt: (106, 106) Mouse moved
Mouse pt: (105, 105) Mouse moved
Mouse pt: (103, 103) Mouse moved
Mouse pt: (100, 100) Mouse moved
Mouse pt: (100, 100) Mouse wheel rotated up 1 click(s)
Mouse pt: (97, 97) Mouse moved
Mouse pt: (95, 95) Mouse moved
Mouse pt: (93, 93) Mouse moved
Mouse pt: (93, 93) Mouse wheel rotated up 1 click(s)
Mouse pt: (92, 92) Mouse moved
Mouse pt: (92, 92) Mouse wheel rotated up 1 click(s)
Mouse pt: (92, 92) Mouse wheel rotated up 1 click(s)
Mouse pt: (92, 92) Mouse moved
Posted 'quit' to message pump thread
Hook thread has finished.
```

CatchMouse automatically stops after about 15 seconds.

Hooking and Unhooking the Procedure

The setHook() method in the CatchMouse class is almost identical to the same-named method in CatchKeys in Hack 11.5, the only differences being the name of the hook procedure and the hook type.


```
// global
private static final int WH_MOUSE_LL = 14; //low-level mouse type


private static void setHook()
{
  Thread hookThread = new Thread( new Runnable() {
    public void run()
    {
      int hInst = Kernel32.GetModuleHandle(null);

      // create mouse hook callback function
      Callback mouseCallback =
               new Callback(CatchMouse.class, "mouseProcessing");

      // add hook function to OS; get back hook ID (handle)
      mouseHook = SetWindowsHookEx(WH_MOUSE_LL, mouseCallback,
                                                    hInst, 0);
      // store the ID of the thread managing the hook
      hookThreadId = Kernel32.GetCurrentThreadId();

      // message dispatch loop (message pump)
      System.out.println("Hook installed");
      Msg msg = new Msg();
      while (User32.GetMessage(msg, 0, 0, 0)) {
        User32.TranslateMessage(msg);
        User32.DispatchMessage(msg);
      }
      System.out.println("Hook thread has finished.");
    }
  });
  hookThread.start();
}  // end of setHook()
```

The mouseProcessing() hook procedure will be called in the context of the thread that installed it, and is triggered by a mouse message being sent to the thread. This means that the thread must have a message dispatch loop.

The unHook() function is identical to the one in CatckKeys.java, so won't be shown again.

Processing the Mouse

The mouseProcessing() hook procedure has the same general form as the keyProcessing() method in CatchKeys(). It begins by checking the nCode value, and passing the message onto the next hook in the chain if there's an error. The method also passes the message on when it's finished.

The differences lie in how mouseProcessing() treats the lParam input argument: here the argument represents a pointer to a MSLLHOOKSTRUCT structure which holds mouse details. The wParam field specifies the identifier of the mouse message, which may be one of WM_LBUTTONDOWN, WM_LBUTTONUP, WM_MOUSEMOVE, WM_MOUSEWHEEL, WM_MOUSEHWHEEL (the horizontal scroll wheel), WM_RBUTTONDOWN, or WM_RBUTTONUP. The required signature for a LowLevelMouseProc function is spelled out at http://msdn2.microsoft.com/en-us/library/ms644986(VS.85).aspx.


```
// global
private static final int WM_MOUSEHWHEEL = 0x020E;
      // mouse message ID missing from WinConstants class


public static int mouseProcessing(int nCode, int wParam, int lParam)
{
  if (nCode < 0)   // pass message to next hook procedure
    return CallNextHookEx(mouseHook, nCode, wParam, lParam);

  if (nCode == WinConstants.HC_ACTION) {
    MSLLHOOKSTRUCT mInfo =
         Util.ptrToStruct(lParam, MSLLHOOKSTRUCT.class);
               // convert lParam input into a mouse struct
    String message = "Mouse pt: (" + mInfo.pt.x + ", " +
                                     mInfo.pt.x + ") ";
    switch (wParam) {
      case WinConstants.WM_LBUTTONDOWN:
        message += "Left button down";
        break;
      case WinConstants.WM_LBUTTONUP:
        message += "Left button up";
        break;
      case WinConstants.WM_MOUSEMOVE:
        message += "Mouse moved";
        break;
      case WinConstants.WM_MBUTTONDOWN:
        message += "Middle button down";
        break;
      case WinConstants.WM_MBUTTONUP:
        message += "Middle button up";
        break;
      case WinConstants.WM_MOUSEWHEEL: {
        message += "Mouse wheel rotated";
        int wheelDelta = mInfo.mouseData >> 16;
                  // the high-order part is the wheel delta
        int numClicks = wheelDelta / WinConstants.WHEEL_DELTA;
        if (numClicks > 0)
          message += " up " + numClicks + " click(s)";
        else   // negative means rotation down
          message += " down " + -numClicks + " click(s)";
        break;
      }
      case WM_MOUSEHWHEEL:    // not in WinConstants
        message += "Horizontal mouse wheel moved";
        break;
      case WinConstants.WM_RBUTTONDOWN:
        message += "Right button down";
        break;
      case WinConstants.WM_RBUTTONUP:
        message += "Right button up";
        break;
    }
    System.out.println(message);
  }
  // pass message onto next hook proc. for further processing
  return CallNextHookEx(mouseHook, nCode, wParam, lParam);
}  // end of mouseProcessing()
```


The MSLLHOOKSTRUCT structure isn't defined in a J/Invoke helper class, but it's easy enough to create a Java definition. The struct is defined at http://msdn2.microsoft.com/en-us/library/ms644970(VS.85).aspx:

```
typedef struct {
    POINT pt;
    DWORD mouseData;
    DWORD flags;
    DWORD time;
    ULONG_PTR dwExtraInfo;
} MSLLHOOKSTRUCT
```

It becomes the Java class:

```
@NativeStruct
public class MSLLHOOKSTRUCT
{
  public Point pt = new Point();
  public int mouseData;
  public int flags;
  public int time;
  public int dwExtraInfo;
}
```

The only unusual aspect is that the Point field in the struct is represented by a com.jinvoke.win32.structs.Point object.

The mouseData field is most commonly accessed when the message concerns the mouse wheel. In that case, the field's high-order word holds the wheel delta. A positive value indicates that the wheel was rotated forward, away from the user, and a negative value that the wheel was rotated backward, toward the user. The delta can be converted into a number of clicks by dividing it by the WHEEL_DELTA constant.

Hacking the Hack: Detecting Window Selection

CatchMouse provides lots of useful mouse state information, but often we need more than that. For example, we may want to know the name of the window where the mouse was clicked.

The secret is to convert the mouse's screen coordinates into a window handle using ChildWindowFromPoint() from user32.dll (documented at http://msdn2.microsoft.com/en-us/library/ms632676.aspx). It determines which, if any, of the child windows belonging to the parent window (the desktop in our case) contains the specified coordinate.

The CatchWinClick.java example illustrates the idea: when the left mouse button is clicked, the name of the window directly under the cursor is printed. An example of the output:

```
> java -cp d:\jinvoke\jinvoke.jar;. CatchWinClick
Hook installed
Clicked on [JI-11-v1.doc - Microsoft Word]
Clicked on [Program Manager]
Clicked on [JI-11-v1.doc - Microsoft Word]
```



```
Clicked on [Program Manager]
Clicked on [EditPlus - [F:\Ch 11 Hooks\Ch 11 Hooks Code\Mouse
Hooks\run.bat]]
Clicked on [C:\WINDOWS\system32\CMD.EXE - run CatchWinClick]
Posted 'quit' to message pump thread
Hook thread has finished.
```

As the user switches between Word, a text editor, and a console window, the titles of those windows are reported. Clicking on the desktop is reported as "Program Manager".

CatchWinClick.java is coded in a similar manner to CatchMouse.java except for the hook procedure. In setHook(), the callback function is mouseWinProcessing():

```
Callback mouseCallback =
      new Callback(CatchWinClick.class, "mouseWinProcessing");
```

mouseWinProcessing() is a simpler version of the mouseProcessing() hook procedure in CatchMouse.java since it only needs to react to a left mouse button press:

```
public static int mouseWinProcessing(int nCode,
                                      int wParam, int lParam)
{
  if (nCode < 0)   // pass message to next hook procedure
    return CallNextHookEx(mouseHook, nCode, wParam, lParam);

  if (nCode == WinConstants.HC_ACTION) {
    MSLLHOOKSTRUCT mInfo =
         Util.ptrToStruct(lParam, MSLLHOOKSTRUCT.class);
               // convert lParam input into a mouse struct
    if (wParam == WinConstants.WM_LBUTTONDOWN) // left button down
      reportTitle(mInfo.pt);
  }

  // pass message onto next hook proc. for further processing
  return CallNextHookEx(mouseHook, nCode, wParam, lParam);
}  // end of mouseWinProcessing()
```

The translation of the mouse coordinates into a window handle, and the reporting of the window's title is done by reportTitle():

```
private static void reportTitle(Point pt)
{
  int hDesktop = User32.GetDesktopWindow();

  // find handle of active, visible child window on desktop at pt
  int hWnd = User32.ChildWindowFromPointEx(hDesktop, pt.x, pt.y,
         WinConstants.CWP_SKIPINVISIBLE |
         WinConstants.CWP_SKIPDISABLED  |
         WinConstants.CWP_SKIPTRANSPARENT );

  int titleLength = User32.GetWindowTextLength(hWnd);
  if (titleLength == 0)    // the window doesn't have a title
    return;

  if (User32.GetParent(hWnd) != 0)
```



```
    return;   // window has no parent, so ignore it

  // get window title
  StringBuffer sb = new StringBuffer(titleLength+1);
  User32.GetWindowText(hWnd, sb, sb.capacity());
  System.out.println("Clicked on [" + sb + "]");
} // end of reportTitle()
```

User32.ChildWindowFromPointEx() requires the handle of the parent window and the mouse coordinates. The flags filter the matching windows to only include those that are active and fully visible. ChildWindowFromPointEx() is documented at http://msdn.microsoft.com/en-us/library/ms632677(VS.85).aspx. The desktop handle is obtained with User32.GetDesktopWindow().

Once we have a window handle, we can do almost anything we like to the window (e.g. close it, resize it, copy its text). We use User32.GetWindowText() to retrieve its title, but first check that the window actually has a title.

Hack 11.7 Refusing to Die (medium level)

Make it harder to terminate a program by having it ignore its close box and ctrl-c's. J/Invoke isn't used.

The novel aspect of this hack is the discarding of ctrl-c's, which really amounts to the application ignoring POSIX-style SIGINT signals. The bad news is that there's no *standard interface* for signal handling in Java. We'll be using the sun.misc package, a proprietary Sun library that might change or even be removed in future versions of the JDK. It also means that our code won't work with IBM's Java, or other vendor's standard libraries.

The range of signals supported by the sun.misc.Signal class varies from one platform to another. The Window's MSDN documentation for signal handling at http://msdn.microsoft.com/en-us/library/xdkz3x12.aspx lists:

- SIGABRT. Abnormal termination.
- SIGFPE. Floating-point error.
- SIGILL. Illegal instruction.
- SIGINT. Ctrl-c signal.
- SIGSEGV. Illegal storage access.
- SIGTERM. Termination request.

However, SIGINT isn't officially supported by any Win32 applications, and SIGILL and SIGTERM are never generated by Windows. Absent from the list is SIGKILL; Windows employs the TerminateProcess() function instead, which unconditionally causes a process and all of its threads to exit (see http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx).

The SignalsWin example looks like a standard Java GUI application (see Figure 11-2).

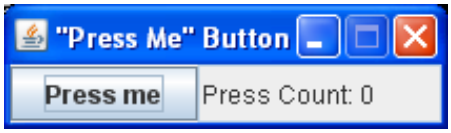

Figure 11-2. The "hard-to-kill" SignalsWin Application.

Appearances can be deceptive however, since SignalsWin is quite hard to kill. Clicking on its close box produces the dialog box in Figure 11-3, and the application carries on executing. The same thing happens when the application is minimized, and the user right clicks on the application icon's "close" menu item.

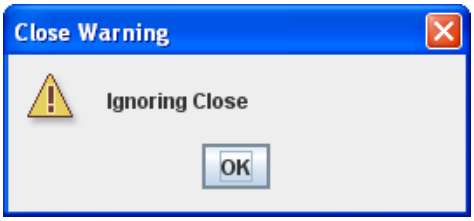

Figure 11-3. The Close Message from SignalsWin.

Typing ctrl-c in the console window where SignalsWin was invoked produces the message shown in Figure 11-4, and the application keeps going. The JVM converts the ctrl-c into a SIGINT signal sent to SignalsWin, which it catches and discards.

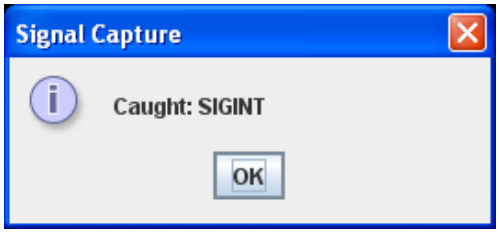

Figure 11-4. The SIGINT Message from SignalsWin.

SignalsWin may be tough, but it's no match for Window's Task Manager. If the user closes SignalsWin via the Task Manager's "Applications" tab, the close message from Figure 11-3 will appear initially. However, the Task Manager follows an ignored close request by a TerminateProcess() call (after a brief time-out), which *does* stop the program. Depending on how Windows is configured, it may display a "End Program" dialog box before calling TerminateProcess().

It's also possible to kill SignalsWin by closing the console window where SignalsWin was started, or by terminating the java.exe process running SignalsWin via the Task Manager's "Processes" tab. Both approaches use TerminateProcess().

Ignoring the Close Box

Making SignalsWin ignore close requests is quite simple: the JFrame's setDefaultCloseOperation() must be set to JFrame.DO_NOTHING_ON_CLOSE. The dialog box shown in Figure 11-3 is displayed by having the JFrame listen for a

window closing event, and then call JOptionPane.showMessageDialog(). All this is done in the SignalsWin constructor:

```
public SignalsWin()
{
  super("\"Press Me\" Button");

  setHandlers();
  makeGUI();

  // ignore the close box, and tell the user
  setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {  JOptionPane.showMessageDialog(null, "Ignoring Close",
          "Close Warning", JOptionPane.WARNING_MESSAGE);
    }
  });

  pack();
  setResizable(false);
  setLocationRelativeTo(null);
  setVisible(true);
} // end of SignalsWin()
```

Handling Ctrl-c and other Signals

SignalsWin sets up its signal handling inside setHandlers(). A series of sun.misc.Signal objects are created, and a sun.misc.SignalHandler object is associated with each one. SignalHandler contains a handle() method which will be called when the associated signal arrives at the application.

```
private void setHandlers()
{
  // create a signal handler that reports to the user
  SignalHandler handler = new SignalHandler () {
    public void handle(Signal sig)
    { JOptionPane.showMessageDialog(null, "Caught: " + sig,
            "Signal Capture", JOptionPane.INFORMATION_MESSAGE);
    }
  };

  String[] signals = {"INT", "KILL", "BREAK", "TERM", "ABRT"};
                     // signals we want to process
  for(String sig : signals) {
    try {
      Signal.handle(new Signal(sig), handler);
        // we're using the same handler to respond to every signal
    }
    catch(IllegalArgumentException e)
    {  System.out.println(e);  }
          /* signal unsupported on this platform or
             processed by the JVM as currently configured */
    catch(Throwable e)
    {  System.out.println(e); }  // missing classes or changed API
  }
```



```
}  // end of setHandlers()
```

Signal.handle() can fail with an IllegalArgumentException exception if it's asked to associated an unknown signal with a handler, or if the signal is already being dealt with by the JVM. It's also possible that the SignalsWin.java may compile, but fail when run at a later date since the sun.misc package has changed (remember: it's not a standard library). This second problem is covered by catching a Throwable exception.

When SignalsWin is started, the following exceptions are reported:

```
> java SignalsWin
java.lang.IllegalArgumentException: Unknown signal: KILL
java.lang.IllegalArgumentException: Signal already used by
VM: SIGBREAK
```

The SIGKILL signal isn't recognized, since Windows uses TerminateProcess(). SIGBREAK is rejected, not because it's unknown to the OS, but because it's already handled by the JVM. When a SIGBREAK is sent to java.exe (by the user typing ctrl-break), the JVM generates a thread dump showing the status of all the active threads. We'll look at modifying this functionality in Hack 11-8.

An alert reader may notice that we've included SIGTERM in the list of signals caught by SignalsWin. Although SIGTERM is never generated by Windows, a dialog box like the one shown in Figure 11-4 is sometimes momentarily displayed on screen, containing "Caught: SIGTERM". This behaviour seems to be associated with the OS terminating the JVM using TerminateProcess().

java.exe can be called with the "-Xrs" argument, which affects SignalsWin:

```
> java -Xrs SignalsWin
java.lang.IllegalArgumentException: Signal already used by VM: SIGINT
java.lang.IllegalArgumentException: Unknown signal: KILL
java.lang.IllegalArgumentException: Signal already used by VM:
SIGBREAK
java.lang.IllegalArgumentException: Signal already used by VM:
SIGTERM
```

"-Xrs" reduces the usage of signals by the JVM and its application, leading to exceptions being raised for all the signals that setHandlers() tries to utilize. When a ctrl-c is typed in the console window, the OS default behavior is carried out, and the JVM and application are terminated as normally.

Hack 11.8 Simpler, Different Thread Dumps (medium level)

Use J/Invoke to modify the meaning of ctrl-break, to generate different kinds of thread dumps.

Although signal handling is supported in Sun's version of Java, it's not a standard feature. The alternative is to use Win32's signal handling capabilities, which are standard, albeit only across versions of Windows.

An advantage of Win32 signal handling is that it can redefine the meaning of a ctrl-break signal. This isn't possible with sun.misc.Signal since the JVM uses ctrl-break to trigger a thread dump.

The GUI functionality of this hack's example, SignalsJIWin.java, is the same as SignalsWin.java in Hack 11.7 (see Figure 11-2). SignalsJIWin duplicates SignalsWin's "hard-to-kill" behavior, ignoring its close box and ctrl-c's, and displaying dialog boxes like those in Figures 11-3 and 11-4. Thankfully, SignalsJIWin can be terminated using the Task Manager, in the same ways as SignalsWin.

The new feature of SignalsJIWin is it's response to a ctrl-break typed in the console window. Instead of the JVM printing a large thread dump of all the active threads, SignalsJIWin's response can be customized inside its signal handler.

Ignoring the Close Box

SignalsJIWin ignores close requests using the same Java techniques as SignalsWin: JFrame's setDefaultCloseOperation() is set to JFrame.DO_NOTHING_ON_CLOSE, and a listener waits for window closing events. All this is done in the SignalsJIWin constructor:

```
// globals
private long mainThreadId;   // thread ID of main application
private int counter = 1;


public SignalsJIWin()
{
  super("\"Press Me\" Button");

  JInvoke.initialize();

  mainThreadId = Thread.currentThread().getId();
  setSignalsHook();

  makeGUI();

  // ignore the close box, and tell the user (as in SignalsWin.java)
  setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
  addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    {  JOptionPane.showMessageDialog(null, "Ignoring Close",
                "Close Warning", JOptionPane.WARNING_MESSAGE);
    }
  });

  pack();
  setResizable(false);
  setLocationRelativeTo(null);
  setVisible(true);

  // do something to keep the main application thread going
  while (true) {
    counter++;
    try {
      Thread.sleep(1000);
    }
```



```
    catch (InterruptedException e) {}
  }
} // end of SignalsJIWin()
```

The SignalsJIWin constructor is different from the one for SignalsWin in several ways. The Win32 signal handling is set up by setSignalsHook(), and there's no use of the sun.misc package. The thread ID for the main application is stored globally, and the thread is prevented from exiting by going into an infinite loop which periodically increments a counter. This unusual coding has nothing to do with Win32 signal handling, but allows us to illustrate different kinds of thread dumps later in this hack.

Handling Ctrl-c, Ctrl-Break, and other Signals

setSignalsHook() installs a function as a signal handler using SetConsoleCtrlHandler() from kernel32.dll (described at http://msdn2.microsoft.com/en-us/library/ms686016(VS.85).aspx). J/Invoke represents the handler as a callback function object.

```
private void setSignalsHook()
{
  // create the signals callback handler for this object
  Callback cb = new Callback(this, "signalsHandler");

  boolean res = Kernel32.SetConsoleCtrlHandler(cb, true);
       // add handler to the list of handlers for the process
  if (!res) {
    System.out.println("Handler setup failed");
    System.exit(0);
  }
}  // end of setSignalsHook()
```

The callback is called signalsHandler(), and is associated with the SignalsJIWin object.

The required signature for a signal handler function is explained at http://msdn2.microsoft.com/en-us/library/ms683242(VS.85).aspx. The function receives a single input argument representing the signal that triggered it, which can be:

- CTRL_C_EVENT: when the user presses ctrl-c
- CTRL_BREAK_EVENT: when the user presses ctrl-break
- CTRL_CLOSE_EVENT: when the program's console is closed
- CTRL_LOGOFF_EVENT: when the user is logged off
- CTRL_SHUTDOWN_EVENT: when the system is shutdown

The handler must return true or false upon finishing. False means that the handler has not processed the signal, and it should be passed to the next suitable handler. If true is returned then the signal has been processed and no other handlers need examine the signal.

Usually a handler catches CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT and CTRL_SHUTDOWN_EVENT so it can perform some specialized cleanup before the program exits. The Windows OS has timeouts associated with these three events: typically five seconds for CTRL_CLOSE_EVENT, and 20 seconds for the other two, but these times can be adjusted. If the program doesn't exit within the timeout period, then Windows displays the "End Program" dialog box to the user (or perhaps terminates the program automatically depending on its configuration). Any cleanup should be finished well before the timeout period has expired.

The signalsHandler() method:

```
public boolean signalsHandler(int signalType)
{
  Thread.currentThread().setName("my signals handler");
      // assign a name to this signal handler thread

  switch (signalType) {
    case WinConstants.CTRL_C_EVENT:
      showCapture("ctrl-c");
      return true;   // the signal has been handled; don't pass on

    case WinConstants.CTRL_BREAK_EVENT:
      printThreads();
      // printStackTraces();
      return true;   // the signal has been handled; don't pass on

    case WinConstants.CTRL_CLOSE_EVENT:  // closes the console
      showCapture("ctrl-close");
      return true;   // but appl still terminates eventually

    case WinConstants.CTRL_LOGOFF_EVENT:   // user logs off
      showCapture("ctrl-logoff");
      return true;   // still terminates

    case WinConstants.CTRL_SHUTDOWN_EVENT:  // system shutdown
      showCapture("ctrl-shutdown");
      return true;   // still terminates

    default:
      return false;    // pass the signal to next handler
  }
}  // end of signalsHandler()
```

signalsHandler() displays a dialog box by calling showCapture():

```
private void showCapture(String sig)
{ JOptionPane.showMessageDialog(null, "Caught: " + sig,
              "Signal Capture", JOptionPane.INFORMATION_MESSAGE);
}
```

In the case of ctrl-c, the dialog box looks like Figure 11-5, and the application carries on afterwards since signalsHandler() returns true.

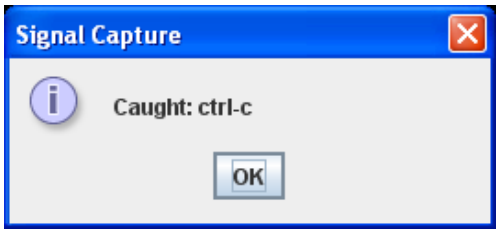

Figure 11-5. The ctrl-c Message from SignalsJIWin.

Similar messages are displayed for CTRL_CLOSE_EVENT, CTRL_LOGOFF_EVENT and CTRL_SHUTDOWN_EVENT, but the OS will still terminate the program with a TerminateProcess() call, perhaps after showing a "End Program" dialog box.

Modifying the Thread Dump

The default thread dump generated by the JVM when it detects a ctrl-break is quite extensive, and somewhat intimidating for naïve (and even intermediate) users. By catching the event in signalsHandler(), it's possible to display a simpler, modified version, which may be better suited to those programmers.

A simple thread dump which briefly lists the active threads is shown in the following execution of SignalsJIWin; the user has typed ctrl-break *three* times:

```
> java -cp d:\jinvoke\jinvoke.jar;. SignalsJIWin
---------------------- 5 active threads ----------------------
Current thread: my signals handler
0. "main" prio=5 Thread id=1 TIMED_WAITING
1. "AWT-Shutdown" prio=5 Thread id=11 WAITING
2. "AWT-Windows" daemon prio=6 Thread id=10 RUNNABLE
3. "AWT-EventQueue-0" prio=6 Thread id=13 WAITING
4. "my signals handler" prio=5 Thread id=14 RUNNABLE
---------------------- 6 active threads ----------------------
Current thread: my signals handler
0. "main" prio=5 Thread id=1 TIMED_WAITING
1. "AWT-Shutdown" prio=5 Thread id=11 WAITING
2. "AWT-Windows" daemon prio=6 Thread id=10 RUNNABLE
3. "AWT-EventQueue-0" prio=6 Thread id=13 WAITING
4. "my signals handler" prio=5 Thread id=14 RUNNABLE
5. "my signals handler" prio=5 Thread id=15 RUNNABLE
---------------------- 7 active threads ----------------------
Current thread: my signals handler
0. "main" prio=5 Thread id=1 TIMED_WAITING
1. "AWT-Shutdown" prio=5 Thread id=11 WAITING
2. "AWT-Windows" daemon prio=6 Thread id=10 RUNNABLE
3. "AWT-EventQueue-0" prio=6 Thread id=13 WAITING
4. "my signals handler" prio=5 Thread id=14 RUNNABLE
5. "my signals handler" prio=5 Thread id=15 RUNNABLE
6. "my signals handler" prio=5 Thread id=16 RUNNABLE
```

Each ctrl-break caught by signalsHandler() results in a call to printThreads(), which prints a dashed line, followed by a numbered list of active threads. Since three ctrl-

breaks were typed during the execution of SignalsJIWin, there are three lists shown in the example above.

printThreads() is defined as:

```
private void printThreads()
// print brief info on all the active threads
{
  int activeNum = Thread.activeCount();
  System.out.println("-------------------- " + activeNum +
           " active threads -------------------------");

  System.out.println("Current thread: " +
                      Thread.currentThread().getName());

  // get array of active threads
  Thread ts[] = new Thread[activeNum];
  Thread.enumerate(ts);
  for (int i = 0; i < activeNum; i++)
    printThreadInfo(i, ts[i]);
}  // end of printThreads()
```

printThreads() uses Thread.enumerate() to fill the ts[] array with every active thread. printThreadInfo() prints a line about each of these threads:

```
private void printThreadInfo(int i, Thread t)
{
  System.out.println(i + ". \"" + t.getName() + "\"" +
     (t.isDaemon()?" daemon":"") + " prio=" + t.getPriority() +
      " Thread id=" + t.getId() + " " + t.getState());
}
```

A Zombie Threads Problem?

A close look at the output from SignalsJIWin reveals a problem with the thread that executes signalsHandler(). Each new ctrl-break shows that the signal handling threads for the previous ctrl-breaks are still in their RUNNING states. These threads are easy to identify since they're all called "my signal handler", a name assigned to the thread at the start of signalsHandler(). The old handler threads should not be present: a thread running signalsHandler() should terminate when the method returns.

I tried out several techniques for forcing the termination of the old handler threads, including Thread.interrupt(), the deprecated Thread.stop(), and the Win32-level method TerminateThread() from kernel32.dll; none of them had any effect.

Printing Stack Information

Another useful thread dump variant is to print stack information for all the live threads. The signalsHandler() method is edited to call printStackTraces() rather than printThreads(), and SignalsJIWin then outputs the following when ctrl-break is typed:

```
> java -cp d:\jinvoke\jinvoke.jar;. SignalsJIWin
-------------------- 10 stack traces -------------------------
0. "Attach Listener" daemon prio=5 Thread id=5 RUNNABLE
```



```
1. "my signals handler" prio=5 Thread id=14 RUNNABLE
        java.lang.Thread.dumpThreads(Native Method)
        java.lang.Thread.getAllStackTraces(Unknown Source)
        SignalsJIWin.printStackTraces(SignalsJIWin.java:185)
        SignalsJIWin.signalsHandler(SignalsJIWin.java:143)

// many more lines, edited out, and then...

9. "main" prio=5 Thread id=1 TIMED_WAITING
    main() thread -- counter: 7
        java.lang.Thread.sleep(Native Method)
        SignalsJIWin.<init>(SignalsJIWin.java:74)
        SignalsJIWin.main(SignalsJIWin.java:237)
```

Later, when ctrl-break is typed again, the output for the main application thread is:

```
10. "main" prio=5 Thread id=1 TIMED_WAITING
    main() thread -- counter: 140
        java.lang.Thread.sleep(Native Method)
        SignalsJIWin.<init>(SignalsJIWin.java:74)
        SignalsJIWin.main(SignalsJIWin.java:237)
```

The output for the main application thread includes the counter value, which is progressively increasing due to the loop inside SignalsJIWin().

printStackTraces() is defined as:

```
// globals
private long mainThreadId; // thread ID of main application
private int counter = 1;


private void printStackTraces()
{
  Map<Thread, StackTraceElement[]> traces =
                          Thread.getAllStackTraces();
       // map of all the live threads and their stack traces

  System.out.println("--------------------- " + traces.size() +
         " stack traces -------------------------");
  int i = 0;
  for (Map.Entry <Thread, StackTraceElement[]> entry:
                                      traces.entrySet()) {
    Thread t = entry.getKey();
    printThreadInfo(i, t);
    i++;

    if (mainThreadId == t.getId())   // report on main thread
      System.out.println("    main() thread -- counter: " + counter);

    // print stack trace for this thread
    StackTraceElement[] trace = entry.getValue();
    for (StackTraceElement line: trace)
      System.out.println("\t" + line);
    System.out.println();
  }
}  // end of printStackTraces()
```


printStackTraces() uses Thread.getAllStackTraces() to return a map of stack traces for all the live threads. The map keys are threads and each map value is an array of StackTraceElement objects that represents the stack dump of the corresponding thread. Each trace element represents a single stack frame by a file name, line number, class, and the method name of the executing line.

printStackTrace() checks for the main application thread, using it's thread ID, and prints out the current value of the counter variable.

This version of SignalsJIWin suffers from the same zombie handler thread problem as the earlier one. Each time ctrl-break is pressed the number of handler threads increases by one, since the old ones never terminate.

This SignalsJIWin reports on ten threads initially, while the early version started with just five. The difference is to do with the meanings of the Thread.getAllStackTraces() and Thread.enumerate() methods: the former returns every live thread, which includes additional JVM-level tasks, while the latter collects only the active threads in the current thread's thread group and subgroups.

Using the JPDA

The Java Platform Debugger Architecture (JPDA) is available for more advanced debugging instrumentation. It consists of three parts: the Java Virtual Machine Tools Interface (JVMTI) which defines the debugging services for a virtual machine (i.e. for the JVM), the Java Debug Wire Protocol (JDWP) for transferring data between the program being debugged and a debugger application, and the Java Debug Interface (JDI) which the debugger utilizes. This approach allows the debugger to be cleanly separated from the program being studied, a technique called *remote debugging*.

The architecture is explained at http://java.sun.com/javase/technologies/core/toolsapis/jpda/, which includes a FAQ and sample code. A brief introduction by Peter V. Mikhalenko can be found at http://articles.techrepublic.com.com/5100-3513_11-6139512.html.

