Today's Lecture

1. Programming with (Unix) timers

2. Java, from the bottom up . . .

Programming with Timers

- Timers are mechanisms that interrupt your program sometime in the future (*e.g.*, 10 seconds from now)
- Such mechanisms are useful (and often necessary) for building user interfaces
- Examples:
  - timing-out on password entry
  - updating a graphical clock
  - monitoring some external device
  - switching to another part of the program
- A common model for timers is to:
  1. Enable and initialize the timer to a time interval
  2. Bind the asynchronous timer event (*e.g.*, a Unix signal) to the handler function f
- It is also useful to know about timers since they will be part of your next assignment

Unix timers: `timer.c`


```
/* timer.c */
/* a simple demonstration of Unix timers */

#include <stdio.h>
#include <sys/time.h>
#include <signal.h>

/* the handler (installed with signal) */
void say_hello(sig, code, scp, addr)
int sig, code;
struct sigcontext *scp;
char *addr;
{
    /* do what's to do */
    printf("hello, world\n");

    /* reinstall handler */
    signal(SIGVTALRM, say_hello);
}

main(int argc, char *argv[])
{
    struct itimerval itv;
    int seconds;

    if (argc != 2) {
        fprintf(stderr, "usage: %s seconds\n", argv[0]);
        exit(1);
    }

    seconds = atoi(argv[1]);

    /* install a handler for SIGVTALRM */
    signal(SIGVTALRM, say_hello);
```



```
/**
    from the getitimer/setitimer man pages:

          ITIMER_REAL         Decrements in real time.  A SIGALRM sig-
                              nal is delivered when this timer
                              expires.

          ITIMER_VIRTUAL      Decrements in process virtual time.  It
                              runs only when the process is executing.
                              A SIGVTALRM signal is delivered when it
                              expires.

          ITIMER_PROF         Decrements both in process virtual time
                              and when the system is running on behalf
                              of the process.  It is designed to be
                              used by interpreters in statistically
                              profiling the execution of interpreted
                              programs.
**/

    /* set timer interval */
    itv.it_interval.tv_sec = seconds;
    itv.it_interval.tv_usec = 0;
    /* initialize timer value */
    itv.it_value.tv_sec = seconds;
    itv.it_value.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &itv, NULL);

    /* loop forever */
    while (1)
        ;
}

/* end of timer.c */
```

Java Datatypes: Numerics

Signed integer types:

`byte` – 8 bits

`short` – 2 bytes

`int` – 4 bytes

`long` – 8 bytes

Note: there are no unsigned integer types.

Floating-point types ("reals"):

`float` – 32 bits

`double` – 64 bits

Casting between the numeric types is permitted

Java Datatypes: `boolean` and `char`

Boolean type:

`boolean` – two values: `true` or `false`

Character type:

`char` – 16-bit Unicode

Some Unicode characters:

`\u0024`	`$`	dollar sign
`\u0041`–`\u005a`	`A-Z`	Latin capital letters
`\u005f`	`_`	underscore
`\u0061`–`\u007a`	`a-z`	Latin small letters
`\u00c0`–`\u00d6`		various Latin letters with diacritics
`\u3040`–`\u318f`		Hiragana, Katakana, Bopomofo, and Hangul
`\u3400`–`\u3d2d`		Korean Hangul Symbols
`\u4e00`–`\u9fff`		Han (Chinese, Japanese, Korean)

Java: Type Wrappers

Primitive Java types (*e.g.*, `boolean`) are not objects

Java does however provide classes to operate on the primitive types (*e.g.*, `java.lang.Boolean`)

Each type's class supplies methods for converting an instance of a primitive type (*e.g.*, `true`) to an object, and *vice versa*

Such classes for the primitive types are called `type wrappers`

For example, some declarations that use the `Boolean` type wrapper:

```
Boolean bObj = new Boolean(true);
boolean bPrim = bObj.booleanValue();
```

`bObj` can, for example, be passed to a method taking a parameter of type `Object`, whereas `bPrim` cannot since it is not an object

Alternately,

```
if (bObj) ···
```

is invalid since a conditional requires a `boolean` value, not an object

Java: Operators

Unary operators:

```
!    ~
++   --
```

Binary operators:

```
<    >    ==   <=   >=
||   &&
+    -    *    /    %
&    |    ^    <<   >>   >>>
=    +=   -=   *=   /=   %=
&=   |=   ^=   <<=  >>=  >>>=
instanceof
```

Ternary operator:

```
? :
```

Java: Differences with C/C++Types

Differences with C/C++:

- No pointers
- No unsigned numeric types
- Unicode bias (in lieu of ASCII)
- No `struct`s, `union`s
- No `sizeof` operator; why?

Java: Arrays

Arrays are objects

Java checks validity of array subscripts dynamically

Instance variable `length` holds an array's length

Some array declarations:

```
int counts[] = new int[10];
int[] counts = new int[10];
double[] twoD = new double[10][20];
char[] s = new char[10];
MyObj[] moArray = new MyObj[10];
```


Java: Strings

Strings are objects of the `String` class
(`char` arrays are not strings in Java)

Java *overloads* the operator '+' to mean string *concatenation*
(in addition to integer/real addition).

```
String HW = "Hello, " + "World" + "!";
```

Some things from the `String` class:

- Constructors

```
public String(String s)
public String(char value[])
public String(byte ascii[], int hibyte)
```

- Methods

```
public int length()
public char charAt(int i)
public boolean equals(Object o)
public String toLower()
```

The `StringBuffer` class provides *mutable* strings

Java: Control Flow

Conditionals:

`if` (*boolean expr*) *stmt* [`else` *stmt*]

`switch` (*expr*) {
    `case` *expr*: *stmt*
    ...
    `default`: *stmt*
}

Loops:

`for` (*expr*; *expr*; *expr*) *stmt*

`while` (*boolean expr*) *stmt*

`do` *stmt* `while` (*boolean expr*)

`break` [*label*]

`continue` [*label*]

No `goto`*!*

Java: Exceptions

With *structured exceptions*, Java programs can detect and handle (practically all) dynamic errors.

This obviates checking for many (rare) error conditions.

```
try {
    code that might fault
} catch (exception-name exception-var) {
    code to handle the exception
} catch (exception-name exception-var) {
    code to handle another exception
} catch ···
    ⋮
} finally {
    code that always executes before leaving try's scope
}
```


Java: Exceptions (cont'd)

Some Java defined exceptions:

- `ArithmeticException` *e*
- `NullPointerException` *e*
- `NegativeArraySizeException` *e*
- `ArrayIndexOutOfBoundsException` *e*

You define your own exceptions by extending the `Exception` class:

```
class MyException extends Exception { }
```

You explicitly invoke your exceptions by *throwing* them:

```
throw (new MyException());
```

An example:

```
try {
    a[i / j] = i * j;
} catch (ArithmeticException e1) {
    // code to handle a divide-by-zero here
    throw new MyException();
} catch (ArrayIndexOutOfBoundsException e2) {
    // code to handle a bad subscript here
    throw new MyException();
} finally {
    // code to always execute when leaving the try block
    i++;
}
```


Java: Miscellaneous

- Comments:

```
/* comment, just as in C */
// comment to the end-of-line, just as in C++

/** comment for automatically generated docs */
```


Java: Sample Program


```
/* IntSort.java */

class IntSort {
        private int Partition(int a[], int pivot, int low, int high) {
                int top = high;
                int bottom = low;

                while (bottom < top) {
                        while (bottom < high && a[bottom] < pivot) bottom++;
                        while (top > low && a[top] > pivot) top--;
                        if (bottom < top) {
                                int tmp = a[bottom];

                                a[bottom] = a[top];  a[top] = tmp;
                                bottom++;  top--;
                        }
                }
                return top;
        }

        private int[] Quick(int a[], int low, int high) {
                int middle;

                if (low >=  high) return a;
                middle = Partition(a, a[low], low, high);
                Quick(a, low, middle-1);
                Quick(a, middle+1, high);
                return a;
        }

        public int[] Sort(int a[]) {
                return Quick(a, 0, a.length-1);
        }
}
```

```
class Main {
        final static int ArraySz = 10;

        private static void Show(int a[]) {
                for (int i = 0; i < a.length; i++) {
                        System.out.print(a[i] + " ");
                }
                System.out.println();
        }

        public static void main(String args[]) {
                int[] a = new int[ArraySz];
                IntSort S = new IntSort();

                for (int i = 0; i < ArraySz; i++) {
                        a[i] = (int) (Math.random() * ArraySz);
                }
                Show(a);
                S.Sort (a);
                Show(a);
        }
}

/* end of IntSort.java */
```


Next Lecture

Guest: Olin Shivers, MIT

- Scheme, an advanced programming language
- `scsh`, Scheme as a shell

Team sign-up for in-class project proposals