

Programming

Problem ▶ Design ▶ Implementation ▶ Results

Make a
tic-tac-toe
demo



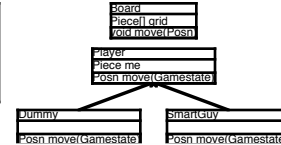
```
class Board {
    Piece[] grid;
    Board( int size ) {
        this.grid = new Piece[size][size];
        for( int i = 0; i < size; i++)
            for( int j = 0; j < size; j++)
                grid[i][j] = new EmptyPiece();
    }
}
```

X|O|X
O|O|X
O|X|O

Programming

Problem ▶ Design ▶ Implementation ▶ Results

Make a
tic-tac-toe
demo



```
class Board {
    Piece[] grid;
    Board( int size ) {
        this.grid = new Piece[size][size];
        for( int i = 0; i < size; i++)
            for( int j = 0; j < size; j++)
                grid[i][j] = new EmptyPiece();
    }
}
```

X|O|X
O|O|X
O|X|O

Programming

Problem ▶ Design ▶ Implementation ▶ Results

Make a
tic-tac-toe
demo



```
class Board {
    Piece[] grid;
    Board( int size ) {
        this.grid = new Piece[size][size];
        for( int i = 0; i < size; i++)
            for( int j = 0; j < size; j++)
                grid[i][j] = new EmptyPiece();
    }
}
```

X|O|X
O|O|X
O|X|O

Programming

Problem ▶ Design ▶ Implementation ▶ Results

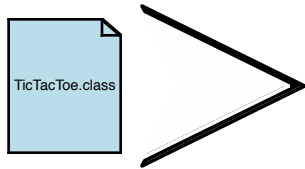
Make a
tic-tac-toe
demo



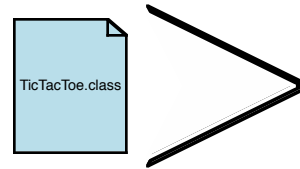
```
class Board {
    Piece[] grid;
    Board( int size ) {
        this.grid = new Piece[size][size];
        for( int i = 0; i < size; i++)
            for( int j = 0; j < size; j++)
                grid[i][j] = new EmptyPiece();
    }
}
```

X|O|X
O|O|X
O|X|O

From code to result

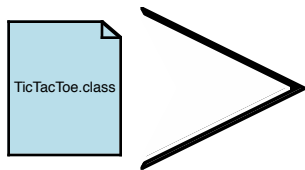


From code to result



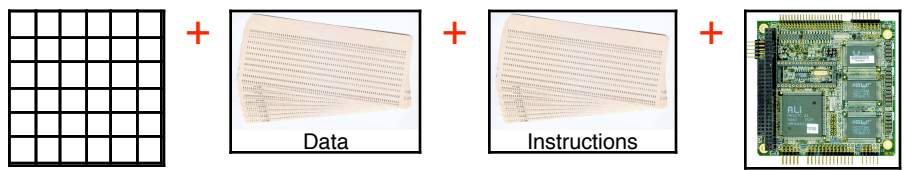
```
ld 345 r3
ld 4 r2
add r2 r3 r1
ld r1 r4
add r2 r4 r3
st r3 r1
```

From code to result



```
x|o|x
x|o|o
o|x|o
```

What is the machine?



Machine language

- `add r1 r2 r3`
- `ld r1 r2`
- `st r1 r2`
- `jmp r1`
- `if0 r1 r2 r3`

Memory

Correlating the machine to code

- `new Fish(10)`

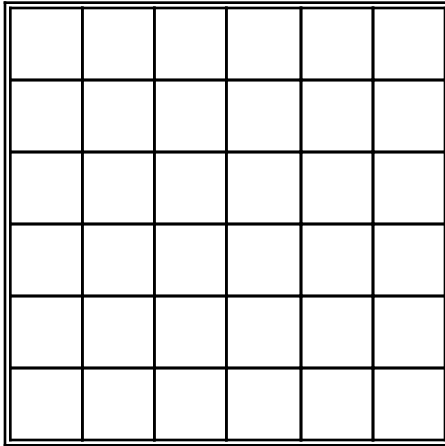
Correlating the machine to code

- `new Fish(10)`

<input type="text" value="10"/>					

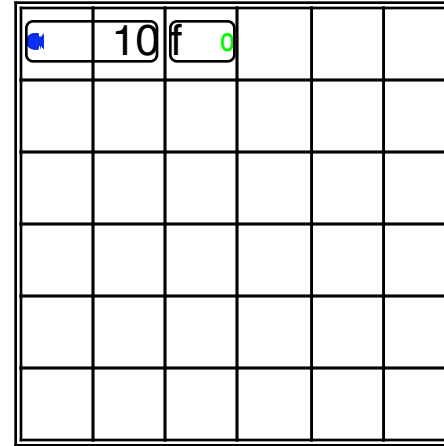
Correlating the machine to code

- Fish f = new Fish(10);



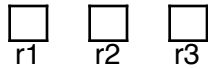
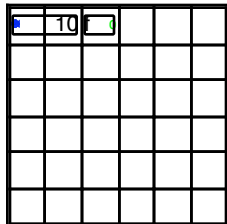
Correlating the machine to code

- Fish f = new Fish(10);



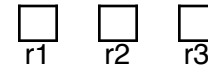
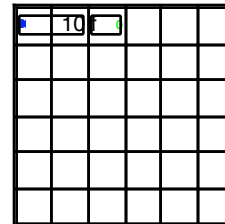
Correlating the machine to code

- f.weight + 3



Correlating the machine to code

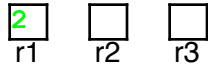
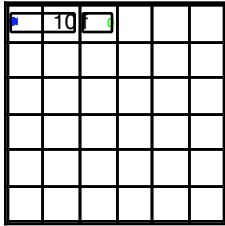
- f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

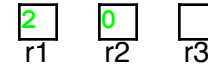
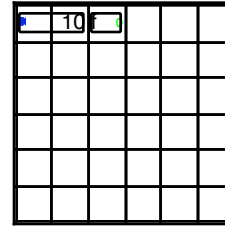
• f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

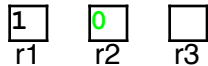
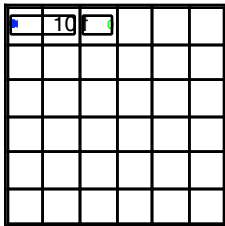
• f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

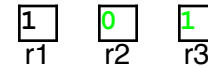
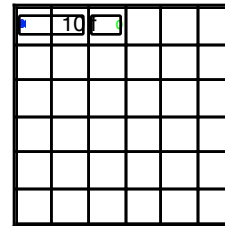
• f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

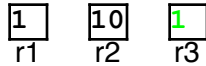
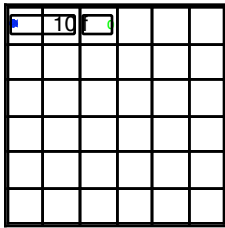
• f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

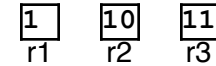
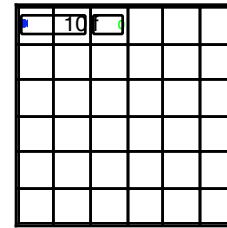
- f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Correlating the machine to code

- f.weight + 3



- ld f r1
- ld r1 r2
- ld 1 r1
- add r1 r2 r3
- ld r3 r2
- add r1 r2 r3

Assembly programming

- People used to write all programs in this kind of language
 - Some still do
- Similarities for programming:
 - Design data: structure and layout
 - Design communication
 - Reason about the problem, tackle what before how
- Differences
 - Java provides assistance in reasoning
 - Less screaming and crying

Other languages

- First major programming language: Fortran 1957
- Numerous languages in use today:
 - Fortran (still)
 - C
 - C++
 - Python
 - Smalltalk
 - Lisp
 - Scheme
 - ML
 - Haskell
 - ... list keeps going for awhile

Looking at a few languages: C

- Procedural language; close to the machine
- Data is stored in records
- Syntax similar to Java
- Procedures similar to methods
 - Not in a class
 - `int f(x) { return x + 1; }`

Looking at a few languages: C

- Procedural language; close to the machine
- Data is stored in records
- Syntax similar to Java
- Records similar to the data portion of a class
 - ```
struct posn {
 int x;
 int y;
};
struct linkedList {
 posn item;
 struct linkedList* prev;
 struct linkedList* next;
};
```

## Some details about C

- Programmer manages memory
- Supports arrays, chars, ints, floats ...
- Has for loops, while loops, recursion
- Statically checks types
- Does not dynamically check array bounds or casts

## Programming in C

- Analyze problem/data and organize data into structures
- Begin thinking about procedure: with purpose and header
- Examples
- Template: list the data available and remind yourself to use procedures for subdata
- Develop the body
- Test

## Fish in C

```
typedef struct {
 int weight;
} fish;
typedef fish *Fish;

Fish f = (Fish)malloc(sizeof(fish));
f->weight = 4;

//Create a new fish from oldF bigger by w
Fish feedFish(Fish oldF, int w) {
 // ... oldF->weight ... w
 Fish newF = (Fish)malloc(sizeof(fish));
 newF->weight = oldF->weight + w;
 return newF;
}
```

## Loops & arrays in C

```
//The current game board, and a scratch board
int Board[14];
int nextBoard[14];
//Note: modifies Board
//Initializes Board to start state
void setStartBoard() {
 //... Board ... nextBoard ...
 int i;
 for(i = 0; i < 14; i++)
 Board[i] = 0;
}
//Set Board to given board: used to progress game
void setBoard(int* b) {
 // b... Board ... nextBoard ...
 int i;
 for(int i = 0; i < 14; i++)
 Board[i] = b[i];
}
// oppositeOf(4) == 10
//Returns the slot in the board on the opposite side
int oppositeOf(int slot) {
 // slot ... Board ... nextBoard
 return 14-slot;
}
```

## Looking at a few languages: C++

- Combines object-oriented and procedural programming
- Data can be stored in objects or structs
- Programmer manages memory
- Statically checks types
- Does not check array bounds or casts
- Syntactically similar to C (and therefore Java)

## A class in C++

```
class Fish
{
public:
 Fish(int);
 // To create a new Fish larger by w;
 Fish feedFish(int w);
private:
 int weight;
};

Fish::Fish(int w) {
 weight = w;
}
new Fish(4)->feedFish(4) ==> Fish(8)
Fish Fish::feedFish(int w) {
 // ... weight ... w...
 return new Fish(weight + w);
}
```

## Programming in C++

- Analyze problem/data and organize into classes and structures
  - Reason about what functionality is tied to data
- Purpose/header
- Examples, template, body, test

## Looking at a few languages: Scheme

- A functional language
- Uses records to store data
- Not statically typed

## A functional language?

Functions are like objects, numbers, booleans

- A function can be parameterized over a function
  - `comparesTo( x, y, comp) = comp(x,y)`
  - `comparesTo(1,2,>)] => >(1,2) => 1 > 2 => false`
- A function can be produced by another function
  - `adder( amt ) = (func( x ) = amt + x)`
  - `adder(1) == A function that adds one to any number`

## Scheme syntax

Not like Java

- `1 + 2 --> (+ 1 2)`
- `add1(x) { return x + 1; } --> (define (add1 x) (+ x 1))`
- `add1(3) --> (add1 3)`

## Programming in Scheme

- Analyze data and build into structures
- Purpose and contract -- comment specifying the types
- Examples
- Template: list the data available and remind yourself to use functions for subdata
- Develop the body
- Test

## A Fish in Scheme

```
; (make-fish integer)
(define-struct fish (weight))

; Data example: (make-fish 4)

; To create fish, new-weight bigger than old-fish
; feed-fish: fish int -> fish
(define (feed-fish old-fish new-weight)
 ; ... (fish-weight old-fish) ... new-weight
 (make-fish (+ (fish-weight old-fish)
 new-weight)))
; (feed-fish (make-fish 4) 4) -> (make-fish 8)
```

## Recursion in Scheme

```
(cons 1 (cons 2 (cons 3 null)))
(list 1 2 3)
(list (make-fish 1) (make-fish 2) (make-fish 3))

; To add up all of the weights in f-list
; add-weights: fish-list -> int
(define (add-weights f-list)
 (cond
 ((null? empty) 0)
 ; ... (fish-function (car f-list))
 ; ... (list-function (cdr f-list))
 ((cons? f-list) (+ (fish-weight (car f-list))
 (add-weights (cdr f-list)))))
; (add-weights (list (make-fish 1) (make-fish 2) (make-fish 3))) = 6
```

## Practical Java Knowledge

### Testing

- A common testing package : junit
  - [www.junit.org](http://www.junit.org)
- Similar to the testing package we used

## A Testclass

- Extend the class TestCase
- Provide a constructor that accepts a String
  - Pass this String to the super class
- Write methods that begin with the word test and return void
- Use methods assertTrue and assertEquals to determine if tests succeed
  - assertTrue(boolean)
  - assertEquals( int, int ) // Think compareExacts
  - assertEquals( Object, Object ) // Think compareObjects
  - etc.

## Writing a simple test

```
import junit.framework.*;
class FishTest extends TestCase {
 FishTest(String name) {
 super(name);
 }

 void testWeightOK() {
 Fish curFish = new Fish(11);
 assertTrue(curFish.weightOK());
 }

 void testFeedFish() {
 Fish curFish = new Fish(10);
 //equals must have been overridden
 assertEquals(curFish.feedFish(10), new Fish(20));
 }
}
```

## Running a simple test

- Put the following in main:
  - `TestResult result= (new FishTest("testWeightOK")).run();`

## Making a Suite of tests

- A Suite of tests allows you to run tests for multiple classes at once
- Allows you to fully test your program
- Create a runAllTests type class
- In it's main:
  - `TestSuite suite = new TestSuite();`  
`suite.add(new FishTest("testWeightOK");`  
`suite.add(new FishTest("testFeedFish");`  
`suite.add(new TireTest("testValuable");`  
`TestResult = suite.run();`

## Graphical Test Running

- junit provides a graphical display of how your tests are doing
- To use the graphical, add the following method to FishTest

```
public static Test suite() {
 TestSuite suite = new TestSuite();
 suite.addTest(new FishTest("testWeightOK"));
 suite.addTest(new MoneyTest("testSimpleAdd"));
 return suite;
}
```