

Basics of Java: Expressions & Statements

Nathaniel Osgood

CMPT 858

February 15, 2011

Java as a Formal Language

- Java supports many “constructs” that serve different functions
 - Class & Interface declarations
 - Importing references to classes from other code libraries
 - Defining methods

Methods

- Methods are “functions” associated with a class
- Methods can do either or both of
 - Computing values
 - Performing actions
 - Printing items
 - Displaying things
 - Changing the state of items
- Consist of two pieces
 - Header: Says what “types” the method expects as arguments and returns as values, and exceptions that can be thrown
 - Body: Describes the algorithm (code) to do the work (the “implementation”)

Method Bodies

- Method bodies consist of
 - Variable Declarations
 - Statements
- Statements are “commands” that *do* something (effect some change), for example
 - Change the value of a variable or a field
 - Return a value from the function
 - Call a method
 - Perform another set of statements a set of times
 - Based on some condition, perform one or another set of statements

Variable Declarations

- Variables in Java are associated with “types” and can contain values
 - The types describe the sort of values that a variable can contain (the set of possible values)
 - E.g.
 - double: Double precision floating point numbers
 - int: (positive & negative): Integer values within some range
 - String: A (reference to a) text sequence
 - boolean: A dichotomous value, holding “true”, or “false”
- When we “declare” a variable, we indicate its name & type – and possibly an initial value

Example Variable Declarations

The screenshot displays the AnyLogic Advanced interface. On the left is a project tree for an 'Elephant' model, showing parameters like 'drinkingPeriod' and 'smokingInitiationRateByAgeAndSmokingStatus', and statecharts including 'behavior' and 'FreeWandering'. The main workspace shows a statechart with states 'FreeWandering' and 'GoToWater', and transitions 'GotThirsty' and 'DrinkWater'. The 'Properties' pane shows the 'Elephant - Active Object Class' with the following code:

```
On Step:  
if( ! isMoving() )  
    error( "Not moving!" );  
Main m = get_Main();  
  
//where am I?  
double x = getX();  
double y = getY();  
int c = min( max( 0, (int) (x/5) ), 99 );  
int r = min( max( 0, (int) (y/5) ), 99 );  
  
//drink if thirsty if in water  
if( thirsty && m.altitude[c][r] < 0 )  
    behavior.receiveMessage( "Drink" );  
  
//demolish trees at current cell, if any  
if( m.vegetation[c][r] > 10000 )  
    m.vegetation[c][r] -= 10000;
```

Annotations with arrows point to specific lines of code:

- Red arrow: Declares a variable "m" that initially contains a reference to the "Main" object
- Blue arrow: Declares double-precision variables x & y
- Green arrow: Declares integer values c & r, and sets equal to the column & row for this elephant in the vegetation array

Finding location
in continuous space
(x,y) & in terms of
Discrete vegetation
Space (c,r).

Poor style -- Should be In
separate function

Common Java Statements

- if
- for
- while / do-while
- Try-Catch-Finally
- Throw (Trigger) exception
- An expression (typically side-effecting)
 - Assignment
 - Call to a function
- Composite statement block (multiple statements enclosed in a “{}”)

Common Java Expressions

- Literal (3.5, 1, “my string”, null)
- Causes changes Side effecting
 - Assignment (a=b) *Left hand side is some location (variable, field, etc.)*
- Comparison (a>b,a==b)
- Mathematical Operators (+,-,/,*) *Can be “overloaded” to mean other things (e.g. + as concatenation)*
- Method call (function call): this.get_Main()
- “Dereferencing”: Looking up field or value *b* in an object expression *a*: (a.b)
- Ternary operator: (predicate ? a : b)

Comments

- Comments in Java are indicated in two different ways
 - Arbitrarily long: Begun with `/*` and ended with `*/`
 - These can span many lines
 - Within a line: after a `//`
- Use comments to describe your intentions!

Rerouting Around Barriers (Boundaries & Water)

Poor Style – entire logic, conditions (checks on boundaries, whether water) & rerouting Logic should all be in separate functions from this & from each other). Remove constants

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a statechart for an elephant agent. The statechart includes a state named 'FreeWandering' with a self-loop labeled 'NewDir'. Transitions from 'FreeWandering' include 'GotThirsty' leading to a state 'GoToWater' and 'DrinkWater' leading back to 'FreeWandering'. The 'GoToWater' state is also shown as a separate state in the diagram.

The 'Elephant - Active Object Class' window shows the following Java code:

```
m.vegetation[c][r] -= 10000;  
  
//avoid bounds and water, change direction if needed  
if( x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0 ) {  
    stop();  
    //try new heading until find a valid one  
    double heading;  
    double xtry, ytry;  
    int count = 0;  
    do {  
        if( count >= 100 ) {  
            error( "Count not find way out!" );  
        }  
        heading = uniform( -Math.PI, Math.PI );  
        xtry = x + 10 * cos( heading );  
        ytry = y + 10 * sin( heading );  
        count++;  
    } while( xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int)(xtry/  
//and start moving in the new direction to a virtual distant target - this will be st  
    moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );  
}
```

For statements

- “For” statements “iterate”, repeatedly executing some inner statement many times
- Several variants are available

```
for (int i = 0; i < 100; i++)
```

statement

```
for (int i : collection)
```

statement

Heading Towards Resource

The screenshot displays the AnyLogic Advanced software interface. On the left, a project tree shows a model named 'Wandering Elephants*' with an 'Elephant' agent. The main workspace contains a state transition diagram for an 'Elephant' agent. It features a state 'thirsty' (yellow circle) with a transition 'GotThirsty' leading to a state 'NewDir' (yellow circle). From 'NewDir', a transition 'DrinkWater' leads to a state 'GoToWater' (yellow circle). A function 'headingToWater' (blue circle) is associated with the 'GoToWater' state. A red arrow points from the text 'Determining current position & Searching for quickest way to find water from that position. (should be in separate function!)' to the 'headingToWater' function in the diagram and to the corresponding code block in the editor.

Determining current position & Searching for quickest way to find water from that position.
(should be in separate function!)

```
Function body:  
stop();  
double x = getX();  
double y = getY();  
  
//find nearest water and set heading there  
double dmin = Double.POSITIVE_INFINITY;  
double heading = 0;  
for( double a = 0; a < 2 * Math.PI; a += Math.PI / 16 ) { // try 16 directions  
    for ( double d = 0; d < 750; d += 5 ) {  
        if ( d >= dmin )  
            break; // we know better direction  
        int c = (int) ( ( x + d * cos( a ) ) / 5 );  
        int r = (int) ( ( y + d * sin( a ) ) / 5 );  
        if ( c < 0 || 100 <= c || r < 0 || 100 <= r )  
            break; // this is outside the area  
        if( get_Main().altitude[c][r] < 0 ) {  
            dmin = d;  
            heading = a;  
            break;  
        }  
    }  
}  
  
//fixed high velocity  
setVelocity( 5 );  
//and start moving in the new direction to a virtual distant target - this will be stoppe  
moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );
```


If Statements

- With an if statement, one tests a condition (“predicate”), and – based on the result – either executes one statement or another (possibly empty) statement

```
if (condition)
    true-statement
else
    false-statement
```

```
if (condition)
    true-statement
or
    true-statement
```

“falls through” to later code if condition is false



Handling of Movement Logic

The screenshot displays the AnyLogic Advanced interface for an 'Elephant' agent. On the left, a project tree shows the agent's structure, including parameters like 'drinkingPeriod' and 'smokingInitiationRateByAgeAndSmokingStatus', and statecharts for 'behavior' and 'FreeWandering'. The main workspace shows a statechart with states 'FreeWandering' and 'GoToWater', and transitions 'GotThirsty' and 'DrinkWater'. The 'Properties' window shows the 'Elephant - Active Object Class' with code for movement logic.

```
On Step:  
  
if( ! isMoving() )  
    error( "Not moving!" );  
  
Main m = get_Main();  
  
//where am I?  
double x = getX();  
double y = getY();  
int c = min( max( 0, (int) (x/5) ), 99 );  
int r = min( max( 0, (int) (y/5) ), 99 );  
  
//drink if thirsty if in water  
if( thirsty && m.altitude[c][r] < 0 )  
    behavior.receiveMessage( "Drink" );  
  
//demolish trees at current cell, if any  
if( m.vegetation[c][r] > 10000 )  
    m.vegetation[c][r] -= 10000;
```

Handling the case of reaching water when thirsty

Distinguishing the case of many & few trees

Finding location in continuous space (x,y) & in terms of Discrete vegetation Space (c,r).

Poor style -- Should be In separate function

Rerouting Around Barriers (Boundaries & Water)

Poor Style – entire logic, conditions (checks on boundaries, whether water) & rerouting Logic should all be in separate functions from this & from each other). Remove constants

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows the 'Elephant' model structure, including parameters like 'drinkingPeriod' and 'smokingInitiationRateByAgeAn', statecharts like 'behavior', and functions like 'GoToWater'. The main workspace shows a statechart with states 'FreeWandering' and 'GoToWater', and transitions 'GotThirsty' and 'DrinkWater'. A red arrow points from a complex condition in the code to the 'GotThirsty' transition.

Elephant - Active Object Class

```
m.vegetation[c][r] -= 10000;

//avoid bounds and water, change direction if needed
if( x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0 ) {
    stop();
    //try new heading until find a valid one
    double heading;
    double xtry, ytry;
    int count = 0;
    do {
        if( count >= 100 ) {
            error( "Count not find way out!" );
        }
        heading = uniform( -Math.PI, Math.PI );
        xtry = x + 10 * cos( heading );
        ytry = y + 10 * sin( heading );
        count++;
    } while( xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int) (xtry/
//and start moving in the new direction to a virtual distant target - this will be st
moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );
}
```

New Direction Change Function Info

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a state machine diagram for an elephant's behavior. The diagram includes a state named 'FreeWandering' with a self-loop labeled 'NewDir'. Transitions from 'FreeWandering' include 'GotThirsty' leading to a state 'GoToWater', and 'DrinkWater' leading back to 'FreeWandering'. The diagram also shows variables like 'drinkingPeriod', 'thirsty', 'headingRandom', and 'headingToWater', and a function 'smokingInitiationRateByAgeAndSmokingStatus'.

The 'headingRandom - Function' properties panel is open, showing the following details:

- Name:** headingRandom
- Access:** default
- Return Type:** void
- Function arguments:** (empty table)

Name	Type

New Direction Change: Function “Body”

The screenshot displays the AnyLogic Advanced software interface. On the left, a project tree shows the model structure for 'Wandering Elephants*'. The main workspace shows a statechart with states like 'FreeWandering' and 'GoToWater', and transitions such as 'GotThirsty' and 'DrinkWater'. A red arrow points from the statechart to the 'headingRandom - Function' code editor at the bottom. The code editor shows the following code:

```
Function body:  
stop();  
//new velocity (note that 12 is the length of time until stop moving in this direction; we'  
setVelocity( get_Main().DistrDisplacement.get() / 12 );  
//new heading  
double heading = getHeading();  
heading += get_Main().DistrAngle.get() * ( randomTrue( 0.5 ) ? 1 : -1 );  
//move  
moveTo( getX() + 1000*cos( heading ), getY() + 1000*sin( heading ) );
```

Annotations in the image include:

- A red arrow pointing to the `setVelocity` line with the text: "Setting Agent Speed (set so as to reach target in fixed time until next target shift)".
- A blue arrow pointing to the `moveTo` line with the text: "Initiates movement towards (randomly chosen) destination".

“While”/“Do while” loop

- Executes a statement as long as some condition is true
- The classic “While” loop has the test at the beginning
- The “do while” has the test at the end of the loop

While loops

The screenshot displays the AnyLogic Advanced [EDUCATIONAL USE ONLY] interface. The main workspace shows a statechart for an elephant's behavior. The statechart includes a state named 'FreeWandering' with a self-loop labeled 'NewDir'. Transitions from 'FreeWandering' include 'GotThirsty' leading to a state 'GoToWater' and 'DrinkWater' leading back to 'FreeWandering'. The 'GoToWater' state is also shown as a separate state.

The 'Elephant - Active Object Class' code editor shows the following code:

```
m.vegetation[c][r] -= 10000;  
  
//avoid bounds and water, change direction if needed  
if( x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0 ) {  
    stop();  
    //try new heading until find a valid one  
    double heading;  
    double xtry, ytry;  
    int count = 0;  
    do {  
        if( count >= 100 ) {  
            error( "Count not find way out!" );  
        }  
        heading = uniform( -Math.PI, Math.PI );  
        xtry = x + 10 * cos( heading );  
        ytry = y + 10 * sin( heading );  
        count++;  
    } while( xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int)(xtry/  
//and start moving in the new direction to a virtual distant target - this will be st  
moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );  
}
```

The interface also shows a project tree on the left, a palette on the right, and a console window at the bottom.

Compound Statements (Delineated by “{ }”)

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows the 'Elephant' model structure. The main workspace contains a state transition diagram with states 'FreeWandering' and 'GoToWater', and transitions 'GotThirsty' and 'DrinkWater'. A red text overlay on the diagram states: 'Innermost is not actually needed, because Only one statement – could remove “{ }” and the statement inside would still be within The “if” “consequent”'. Below the diagram, the 'Elephant - Active Object Class' code is shown. A red arrow points from the text to a nested 'if' statement within a 'do' loop in the code.

```
m.vegetation[c][r] -= 10000;  
  
//avoid bounds and water, change direction if needed  
if( x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0 ) {  
    stop();  
    //try new heading until find a valid one  
    double heading;  
    double xtry, ytry;  
    int count = 0;  
    do {  
        if( count >= 100 ) {  
            error( "Could not find way out!" );  
        }  
        heading = uniform( -Math.PI, Math.PI );  
        xtry = x + 10 * cos( heading );  
        ytry = y + 10 * sin( heading );  
        count++;  
    } while( xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int) xtry/  
//and start moving in the new direction to a virtual distant target - this will be st  
    moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );  
}
```

Expression Statements

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows the 'Elephant' model structure, including parameters like 'drinkingPeriod' and 'smokingInitiationRateByAgeAndSmokingStatus', and states like 'FreeWandering' and 'GoToWater'. The main workspace shows a statechart with a 'FreeWandering' state containing a 'NewDir' transition and a 'GoToWater' state. A red arrow points from the statechart to the code editor.

Assignment expressions as an expression statements (including "count++", which is equivalent to "count=count+1")

Method call Expression as an expression statement

```
m.vegetation[c][r] -= 10000;  
  
//avoid bounds and water, change direction if needed  
if( x < 0 || x >= 500 || y < 0 || y >= 500 || m.altitude[c][r] < 0 ) {  
    stop();  
    //try new heading until find a valid one  
    double heading;  
    double xtry, ytry;  
    int count = 0;  
    do {  
        if( count >= 100 ) {  
            error( "Count not find way out!" );  
        }  
        heading = uniform( -Math.PI, Math.PI );  
        xtry = x + 10 * cos( heading );  
        ytry = y + 10 * sin( heading );  
        count++;  
    } while( xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[(int) xtry/  
    //and start moving in the new direction to a virtual distant target - this will be st  
    moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );  
}
```