

Basics of Java: Statements

Nathaniel Osgood

MIT 15.879

April 11, 2012

Recall: Methods

- Methods are “functions” associated with a class
- Methods can do either or both of
 - Return a value (doing computation as required)
 - Note that this value could be a reference to a value collection
 - Performing actions
 - Printing items
 - Displaying things
 - Changing the state of items
- Best practices
 - A method should do *one* of the above, not both!
 - Should either be a “query” (return values) or a “command” (perform action)
 - Methods should be *well named* (communicate intention)

Method Elements: 2 Pieces

- Header: Specifies what
 - “Types”
 - Expects as “arguments” (formal parameters – values given to the function)
 - Returned Value
 - “Exceptions” that can be thrown
- Body: Describes the algorithm (code) to do the work (the “implementation”)
- Best Practice: A well-documented (specified) method is has a *contract* that specifies *what* it does
 - we give it parameters with certain characteristics, and it does a certain job for us
 - We don’t have to worry about details of *how* it works
- The name & “header” of the function collectively give much hints as to the contract

Method Bodies & “Statements”

- Method bodies consist of
 - Comments (mostly ignored by “build”)
 - Variable Declarations
 - Statements (most involving “Expressions”)
- We discuss each of these below

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 left sidebar shows the project structure for 'Elephant', including parameters like 'drinkingPeriod: 100' and 'smokingInitiationRateByAgeAn', plain variables, statecharts, and functions.

The bottom panel shows the 'Elephant - Active Object Class' with 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 ) );  
}
```

Method Bodies & “Statements”

- Method bodies consist of
 - √ Comments (mostly ignored by “build”)
 - Statements (most involving “Expressions”)

Java Statements

- In contrast to Java Expressions (which calculate a value), Java “statements” *do* something – they *effect some change (to “program state”)*
- Statements are “commands” that, for example
 - Change the value of a variable or a field (this is an assignment expression)
 - Return a value (computed by an expression!) from the function
 - Call a method (call being in an expression)
 - Perform another sequence of statements a certain number of times (given by an expression) , or until a condition (given by an expression) is true
 - Based on some condition (given by an expression), perform one or another sequence of statements

When AnyLogic seeks *action* code (e.g. as a handler), we can give it a statement (or, typically a sequence of one or more statements).

Common Java Statements

- *if*
- *for*
- *while* or *do-while*
- *Try-Catch-Finally*
- *Throw* (Trigger) exception
- An expression (typically side-effecting – should be terminated by a “;”)
 - Assignment
 - Call to a function
- Composite statement block (multiple statements enclosed in a “{}”)

For statements

Note variable declaration.
This variable can then be used within the statement itself

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

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

Iterates over all integers from 0 to 99 (inclusive), with *i* bound to each integer in turn

```
for (Agent a : collection)  
    statement
```

Iterates over all of the agents in *Collection* (with *a* bound to each element of *collection* in turn)

Heading Towards Resource

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows a model named 'Wandering Elephants*' with an 'Elephant' agent. The main workspace shows a state transition diagram with states 'thirsty', 'NewDir', and 'GoToWater'. Transitions include 'GotThirsty', 'DrinkWater', and 'GoToWater'. A red text box is overlaid on the diagram with the text: "Determining current position & Searching for quickest way to find water from that position." Below this, a red arrow points to the 'headingToWater' function code in the console window. The code is as follows:

```
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 ) );
```

A red text box is overlaid on the code editor with the text: "(should be in separate function!)".

If Statements

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

This can be any expression that evaluates to a boolean (true or false) value

if (condition) consequent or if (condition) alternative
else

alternative

“falls through” to later code if condition is false. This is like having an “empty” (blank)

alternative

Handling of Movement Logic

The screenshot displays the AnyLogic Advanced interface for an 'Elephant' agent. On the left, a project tree shows the 'Elephant' agent with various components like Parameters, Statecharts, and Functions. The main workspace shows a statechart with states: 'FreeWandering' (yellow circle), 'GotThirsty' (yellow circle), and 'GoToWater' (yellow rectangle). Transitions include 'NewDir' (self-loop on FreeWandering), 'GotThirsty' (from FreeWandering to GotThirsty), and 'DrinkWater' (from GotThirsty to GoToWater). A 'behavior' entry point is also shown.

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

- Red text: "Handling the case of reaching water when thirsty" with a red arrow pointing to the 'if(thirsty && m.altitude[c][r] < 0)' line.
- Blue text: "Distinguishing the case of many & few trees" with a blue arrow pointing to the 'if(m.vegetation[c][r] > 10000)' line.

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 text annotation with an arrow points to a specific line in the code below.

A more complex condition (should really place condition in 1-2 functions that returns a boolean, and just call the functions! – can reuse elsewhere)

```
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 transition 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'. Other elements include a state 'thirsty', a function 'headingRandom', and a state 'headingToWater'. The 'headingRandom' function is highlighted in the diagram.

The 'headingRandom - Function' properties window 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 Explorer on the left with a tree view of the model structure, including 'Elephant', 'Parameters', 'Plain Variables', 'Statecharts', and 'Functions'. The Properties and Console panels are visible at the bottom, and a Palette on the right contains various modeling elements like Parameter, Flow Aux Variable, Stock Variable, Event, Dynamic Event, Plain Variable, Collection Variable, Function, Table Function, Port, Connector, Entry Point, Entry Point, State, Transition, Initial State Pointer, Branch, History State, Final State, and Environment.

Switch/Case

- A “Switch” statement and its associated “case” clauses are a form of conditional somewhat like a multi-way “if” statement
- Contrast:
 - **If statement:** Is provided with a *boolean* value, and has one clause for the case where this is *true*, and (optionally) another for case where it is *false*
 - **Switch statement:** This is provided with a more general value (int, Enum, char, short, byte, character, in Java 7 a String), and has an arbitrary number of “case” clauses, each to handle different possible concrete values

Example Switch Statement

Properties  Console

Main - Active Object Class

General

Name: Ignore

Agent Generic

Startup code:

```
switch (networkFileType)
{
    case Pajek:
        establishNetworkTransitionsAndPopulationsFromPajekNetworkFile(networkFilePathAndName);
        break;
    case ConnectivityMatrix:
        establishNetworkTransitionsAndPopulationsFromConnectivityMatrixFile(networkFilePathAndName);
        break;
    default:
        throw new RuntimeException("Unexpected networkFileType " + networkFileType);
}
environment.applyLayout(); // now that established connectivity, perform layout
```

Destroy code:

Composite Statements (“Blocks”) (Delineated by “{ }”)

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows the 'Elephant' model with various components like parameters, statecharts, and behaviors. The main workspace shows a state transition diagram with states 'FreeWandering' and 'GoToWater', and transitions 'GotThirsty' and 'DrinkWater'. A red text box is overlaid on the diagram, stating: 'Innermost “{ }” is not currently needed, because only one statement – could remove “{ }” and the statement inside would still be within the “if” “consequent”. But it is safer to have a block, in case further statements are added later'. Below the diagram, the 'Elephant - Active Object Class' code editor shows a code block for avoiding bounds and water. A blue arrow points from the text 'Variables declared inside block “disappear” after leaving the block' to the local variable declarations in the code. A red arrow points from the text 'Innermost “{ }” is not currently needed...' to the curly braces of the code block.

Innermost “{ }” is not currently needed, because only one statement – could remove “{ }” and the statement inside would still be within the “if” “consequent”. But it is safer to have a block, in case further statements are added later

Variables declared inside block “disappear” after leaving the block

```
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/10)][(int)(ytry/10)] > 0 );
    //and start moving in the new direction to a virtual distant target - this will be stopped when reach the target
    moveTo( x + 1000*cos( heading ), y + 1000*sin( heading ) );
}
```

Composite Statements and Variables

- Variables can be declared within a composite statement
- The region of the variable's visibility (i.e. the scope of the variable) is from there to the end of the enclosing statement
- The entire body of a method is a compound statement (hence the “{ }” surrounding it)

Recall: Variable Declarations

- Variables in Java are associated with “types” and can contain values
- When we “declare” a variable, we indicate its name & type – and possibly an initial value

Variable Declaration Statement

The screenshot displays the AnyLogic Advanced interface. On the left, a project tree shows the 'Elephant' model structure. The main workspace shows a statechart with a 'FreeWandering' state and a 'GoToWater' state. A red arrow points from the statechart to the code in the Properties window.

Statechart Diagram:

- State: FreeWandering (yellow circle)
- Initial State: FreeWandering (black dot)
- Transitions:
 - GotThirsty (from FreeWandering to GoToWater)
 - DrinkWater (from GoToWater to FreeWandering)
 - Self-loop: NewDir (from FreeWandering to FreeWandering)
- State: GoToWater (yellow circle)

Code in Properties Window (Elephant - Active Object Class):

```
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;
```


Expression Statements

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 ) );
}
```

Exceptions

- Not uncommonly, things may “go wrong” during execution of code
- We frequently want a way to signal that something has gone wrong
 - Stop normal processing of the code
 - Go “up” to a context where we know how to deal with (handle) the error
 - Up is defined in terms of the “call stack” – we wish to return to successive callers until one handles this condition
- To signal such exceptional conditions, java uses *Exceptions*
- Exceptions in Java are *thrown* where they occur & *caught* in “handlers” where we wish to handle them

Try-Catch Statements

try

```
{ try-block }
```

Exceptions thrown in **this block (a compound statement)** that are (most particularly) of **this exception type** are then handled by running **this block**

```
catch (ExceptionType1 e)
```

```
{ catch-block1 }
```

Exceptions thrown in the “try-block” that are of **this exception type** are then handled by running **this block**

```
catch (ExceptionType2 e)
```

```
{ catch-block2 }
```

...

```
catch (ExceptionType $n$  e)
```

```
{ catch-block $n$  }
```

Example Applications of “Try-Catch”

The screenshot shows the AnyLogic University IDE with a Java code editor. The code defines a function `parseAndProcessPajekEdgeDeclaration` that uses a try-catch structure to handle parsing errors. Annotations in red, green, and blue explain the purpose of different parts of the code.

Red Annotations:

- Here we
- 1) Try to parse the line so as to extract two integers from it
- 2) Connect the corresponding individuals in the population

Green Annotation:

- Handles cases where we can't find the expected numbers in the string

Blue Annotation:

- Handles cases where specified indices are out of bounds

Code Snippet:

```
try
{
    int iPajekEdgeNumber1 = scanner.nextInt();
    int iPajekEdgeNumber2 = scanner.nextInt();

    // the indices here are expressed at 0 offset (not 1 offset as in pajek

    if (iPajekEdgeNumber1 != iPajekEdgeNumber2) // ignore self edges
        population.get(iPajekEdgeNumber1 - 1).connectTo(population.get(iPajekEdgeNumber2 - 1));
}
catch (NumberFormatException e)
{
    System.err.println("Could not parse line " + strLine + "; two numbers separated by whitespace were expected (indicating nodes);");
    e.printStackTrace(System.err);
    throw e;
}
catch (IndexOutOfBoundsException e)
{
    System.err.println("Could not parse line " + strLine + "; one or both of the individuals specified could not be found within");
    e.printStackTrace(System.err);
    throw e;
}
```