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
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**

- “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).

Note variable declaration. This variable can then be used within the statement itself.
Determining current position & Searching for quickest way to find water from that position.

(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.

          if (condition)                         if (condition)
              consequent       or           alternative

    else

      alternative

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

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

Handling the case of reaching water when thirsty

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

Poor style -- Should be in separate function

Distinguishing the case of many & few trees
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

A more complex condition (should really place condition in 1-2 functions that returns a boolean, and just call the functions! – can reuse elsewhere)
New Direction Change Function Info
New Direction Change: Function “Body”

Setting Agent Speed (set so as to reach target in fixed time until next target shift)

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

```java
while (xtry < 0 || xtry >= 500 || ytry < 0 || ytry >= 500 || m.altitude[int](xtry) < 0) {
    // and start moving in the new direction to a virtual distant target - this will be at
    moveTo(x + 100*cos heading, y + 100*sin heading);
}
```
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

```java
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
```
Composite Statements ("Blocks")
(Delineated by "{}")

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.
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

This declares the variable y (using as its initial value the value returned by this.getY())
Expression Statements

Assignment expressions as an expression statement (including “count++”, which is equivalent to “count=count+1”)

Method call expression as an expression statement
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

```java
try {
    try-block
} catch (ExceptionType1 e) {
    catch-block1
} catch (ExceptionType2 e) {
    catch-block2
}...
catch (ExceptionType n e) {
    catch-blockn
}
```

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

Exceptions thrown in this block (a compound statement) that are (most particularly) of this exception type are then handled by running this block.
Example Applications of “Try-Catch”

Here we:

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

- Handles cases where specified indices are out of bounds
- Handles cases where we can’t find the expected numbers in the string