

Methods and Functional Abstraction

Nathaniel Osgood

MIT 15.879

May 2, 2012

Building the Model Right: Some Principles of Software Engineering

Technical guidelines

- Try to avoid needless complexity
- Use abstraction & encapsulation to simplify reasoning & development
- Name things carefully
- Design & code for transparency & modifiability
- Document & create self-documenting results where possible
- Consider designing for flexibility
- Use defensive programming
- Use type-checking to advantage
 - Subtyping (and sometimes subclassing) to capture commonality
 - For unit checking (where possible)

Process guidelines

- Use peer reviews to review
 - Code
 - Design
 - Tests
- Perform simple tests to verify functionality
- Keep careful track of experiments
- Use tools for version control & documentation & referent integrity
- Do regular builds & system-wide “smoke” tests
- Integrate with others’ work frequently & in small steps
- Use discovery of bugs to find weaknesses in the Q & A process

The Challenges of Complexity

- Complexity of software development is a major barrier to effective delivery of model value
- Complexity leads to systems that are late, over budget, and of substandard quality
- Complexity has extensive impact in both human & technical spheres
- Achieving modularity in a model is key to reducing complexity of model development

Why Modularity?

- As a way of managing complexity: Allows decoupling of pieces of the system
 - “*Separation of Concerns*” in comprehension & reasoning
 - Example areas of benefit
 - Code creation
 - Modification
 - Testing
 - Review
 - Staff specialization
 - *Modularity allows ‘divide and conquer’ strategies to work*
- As a means to reuse

Abstraction: Key to Modularity

- Abstraction is the process of forgetting certain details in order to treat many particular circumstances as the same
- We can distinguish two key types of abstraction
 - *Abstraction by parameterization.* We seek generality by allowing the same mechanism to be adapted to many different contexts by providing it with information on that context
 - *Abstraction by specification.* We ignore the implementation details, and agree to treat as acceptable any implementation that adheres to the specification
 - [Liskov&Guttag 2001]

A Key Motivator for Abstraction: Risk of Change

- Abstraction by specification helps lessen the work required when we need to modify the program
- By choosing our abstractions *carefully*, we can gracefully handle anticipated changes
 - e.g. Choose abstracts that will hide the details of things that we anticipate changing frequently
 - When the changes occur, we only need to modify the implementations of those abstractions

Abstraction by Parameterization

- Major benefit: *Reuse*
 - Common needs identified
 - Elimination of need to separately
 - Develop
 - Test
 - Review
 - Debug
- Diverse forms
 - Functions: Formal parameters
 - Generics/Parameterized types
 - Cross cutting: Aspects (parameterized by pointcuts)

Parameterization

- We can parameterize functions, so that the values that they yield depends on the values passed to them as “arguments” by callers
 - This allows flexibly: A function can be used somewhat differently in different contexts
 - While parameters may differ, the *behavior of the function* will typically be the same

Examples of Parameterization

- We may build a function that identifies all people who have been smokers for more than n years
 - n here is a parameter! Different contexts, we might be interested in different n .
- We may wish to count the number of people of a certain sex
 - Rather than independently creating separate methods for Males and Females, we may create a method that is called `CountPopulationOfSex` that takes a parameter that specifies the sex of interest

Types of Abstraction in Java

- Functional abstraction: Action performed on data
 - We use functions (in OO, *methods*) to provide some functionality while hiding the implementation details

We are concentrating on this today
- Interface/Class-based abstraction: State & behaviour
 - We create “interfaces”/“classes” to capture behavioural similarity between sets of objects (e.g. agents)
 - The class provides a contract regarding
 - Nouns & adjectives: The characteristics (properties) of the objects, including state that changes over time
 - Verbs: How the objects do things (*methods*) or have things done to them

Functional Abstraction

- Functional abstraction provides methods to do some work (*what*) while hiding details of *how* this is done
- A method might
 - Compute a value (hiding the algorithm)
 - Test some condition (hiding all the details of exactly what is considered and how): e.g. ask if a person is susceptible
 - Perform some update on e.g. a person (e.g. infect a person, simulate the change of state resulting from a complex procedure, transmit infection to another)
 - Return some representation (e.g. a string) of or information about a person in the model

Encapsulation: Accompanies Abstraction

- *Separation of interface from implementation (allowing multiple implementations to satisfy the interface)* facilitates modularity
- Specifications specify expected behavior of anything providing the interface
- Types of benefits
 - *Locality*: Separation of implementation: Ability to build one piece without worrying about or modifying another
 - See earlier examples
 - *Modifiability*: Ability to change one piece of project without breaking other code
 - Some reuse opportunities: Abstract over mechanisms that differ in their details to only use one mechanism: e.g. Shared code using interface based polymorphism

Why Use Functional Abstraction?

- Easier modifiability: Only one place to update
- Transparency : What the code does is clearer
 - Can communicate intention from clear name
 - Reduced clutter throughout code: Don't have to look at all the gory details of the how every time want to undertake this task
- Easier later reuse
- Reduced complexity lowers risk of programming error

Using Functional Abstraction in AnyLogic

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a state transition diagram for a 'Person' model, divided into three functional blocks: Tuberculosis (green), Tobacco Use (grey), and Diabetes (cyan). The Tuberculosis block includes states like 'TbSusceptible', 'LTI', 'UndiagnosedActiveTb', and 'DiagnosedActiveTb'. The Tobacco Use block includes 'NewSmoker', 'CurrentSmoker', and 'FormerSmoker'. The Diabetes block includes states from 'NormalGlycemia' to 'DiabeticOnOralMedication'. A 'Death' state is a central hub for transitions from various states in the other blocks. The left sidebar shows a 'Project' tree with a 'Functions' folder highlighted in a red circle, listing various functions such as 'AgeCoefficientForSmokingInitiation', 'CirclePerimeterColorFromState', and 'getDegree'. The right sidebar shows a 'Palette' with various modeling elements like 'Parameter', 'Flow Aux Variable', and 'State'. The bottom of the interface shows a 'Properties' panel.

AnyLogic Advanced [EDUCATIONAL USE ONLY]
File Edit View Model Window Help
50% Get Support

Project Person

TBRiskFactors
Main
Person
Parameters
Plain Variables
Dynamic Variables
Statecharts
Functions
AgeCoefficientForSmokingInitiation
CirclePerimeterColorFromState
CirclePerimeterWidthFromState
CountContacts
CountSmokingContacts
FractionOfContactsThatSmoke
IsCurrentSmoker
ReactivationRateCoefficientForCKD
ReactivationRateCoefficientForSmokingStatus
ReactivationRateForSmokingStatus
SmokingInitiationHazardCoefficient
SmokingInitiationHazard
getDegree
Presentation
Simulation: Main

Model
Parameter
Flow Aux Variable
Stock Variable
Event
Dynamic Event
Plain Variable
Collection Variable
Function
Table Function
Port
Connector
Entry Point
State
Transition
Initial State Pointer
Branch
History State
Final State
Environment

Action
Analysis
Presentation
Connectivity
Enterprise Library

Using Functional Abstraction in AnyLogic: Example Functions

- Functions
 - AgeCoefficientForSmokingInitiation
 - CirclePerimeterColorFromState
 - CirclePerimeterWidthFromState
 - CountContacts
 - CountSmokingContacts
 - FractionOfContactsThatSmoke
 - IsCurrentSmoker
 - ReactivationRateCoefficientForCKDStage
 - ReactivationRateCoefficientForSmokingStatus
 - ReactivationRateForSmokingStatusAndCKDStage
 - SmokingInitiationHazardCoefficientAsAFunctionOfFractionOfContactsThatSmoke
 - SmokingInitiationHazard
 - getDegree



Hands on Model Use Ahead



Load Sample Model:

ABMModelWithBirthDeath

(Via “Sample Models” under “Help” Menu)

A Function's Definition

The screenshot displays a software development environment with a class diagram and a properties window. The class diagram shows two classes, **NonPregnant** and **Pregnant**, both represented by yellow rounded rectangles. Arrows indicate a bidirectional relationship between them. To the right of the diagram, a list of functions is shown, with **PerformBirth** highlighted in blue. Below the diagram, the **Properties** window is open, showing the configuration for the **PerformBirth - Function**.

PerformBirth - Function

General

Name: Show Name Ignore Public Show At Runtime

Code

Description

Access: Static

Return Type: void boolean int double String Other:

Function arguments:

Name	Type	

Selection

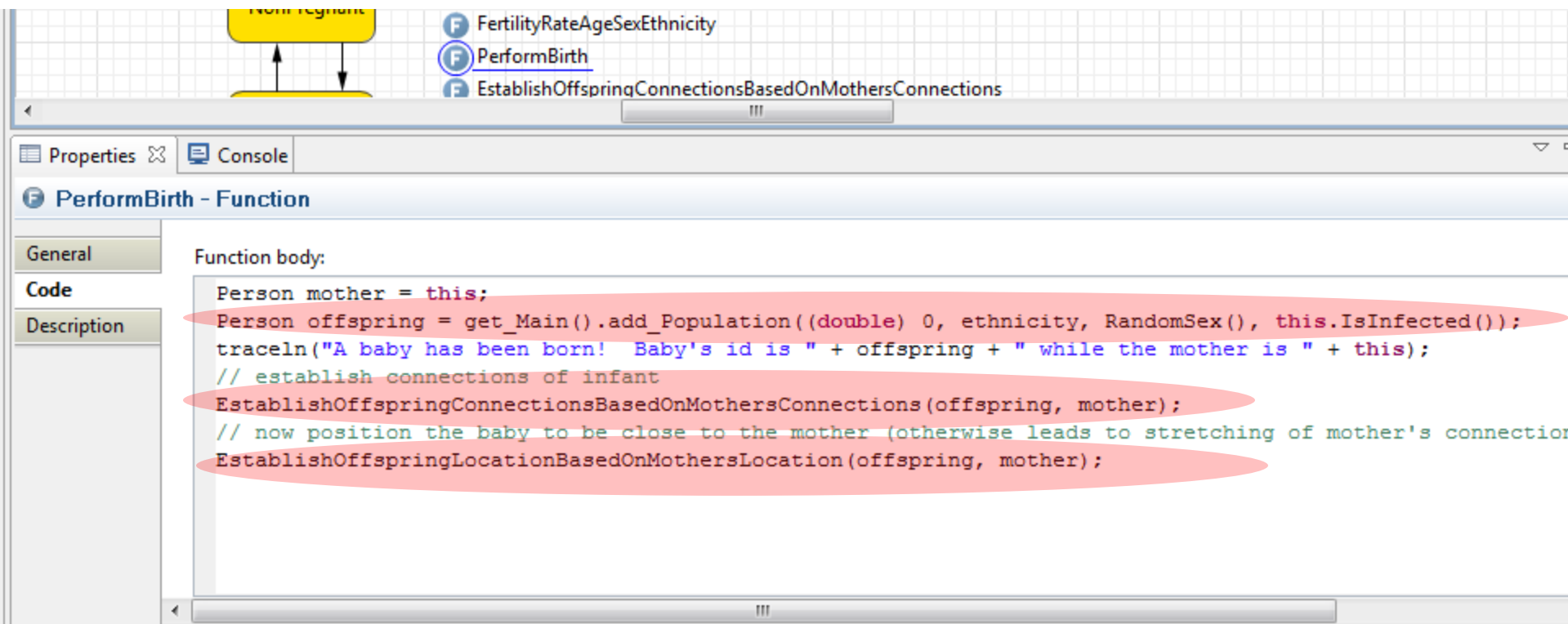
Another Example

The screenshot displays the AnyLogic Advanced software interface. The main workspace shows a state transition diagram with two states: 'NonPregnant' and 'Pregnant'. A transition labeled 'PregnancyStatus' leads from 'NonPregnant' to 'Pregnant', and another transition leads from 'Pregnant' back to 'NonPregnant'. The 'PerformBirth' function is highlighted in the diagram.

The 'PerformBirth' function is defined in the Properties window as follows:

```
Function body:  
Person mother = this;  
Person offspring = get_Main().add_Population((double) 0, ethnicity, RandomSex(), this  
println("A baby has been born! Baby's id is " + offspring + " while the mother is "  
// establish connections of infant  
EstablishOffspringConnectionsBasedOnMothersConnections(offspring, mother);  
// now position the baby to be close to the mother (otherwise leads to stretching of  
EstablishOffspringLocationBasedOnMothersLocation(offspring, mother);
```

A Closer Look at the Code...



The screenshot shows the AnyLogic IDE interface. At the top, a diagram shows a yellow box labeled 'Non-Infected' with arrows pointing to and from another yellow box below it. To the right, a list of functions is visible: 'F FertilityRateAgeSexEthnicity', 'F PerformBirth', and 'F EstablishOffspringConnectionsBasedOnMothersConnections'. Below this, the 'PerformBirth - Function' is selected, and its code is displayed in the 'Code' tab. The code is as follows:

```
Function body:
Person mother = this;
Person offspring = get_Main().add_Population((double) 0, ethnicity, RandomSex(), this.IsInfected());
println("A baby has been born! Baby's id is " + offspring + " while the mother is " + this);
// establish connections of infant
EstablishOffspringConnectionsBasedOnMothersConnections(offspring, mother);
// now position the baby to be close to the mother (otherwise leads to stretching of mother's connection
EstablishOffspringLocationBasedOnMothersLocation(offspring, mother);
```

What is called a “function” in AnyLogic is classically called a “Method” in Object Oriented Programming

Methods

- Methods are “functions” you can call on an object
- 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
- Recall: 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

Method “Parameters” or “Arguments”

- Parameters passed to the method are accessible within the method body
- These parameters are only available inside the method
 - Once the method exits, these parameters are no longer available
 - The fact that we have a parameter named “a” does not change any value of “a” outside the method
 - If we refer to “a” within the method, we will be referring to the value of the parameter
 - After we leave the method, “a” will refer to whatever it did before the method call
- In most (i.e. non-static) “Methods”, “this” is passed implicitly as a parameter – to tell the method on what object we have invoked this method

Pass by Value & Modifying Parameters

- In Java, changing the *value* of the parameters does not modify the values of variables passed to this method
 - Note, however, that if a parameter in the method is a reference to the same *object* as something outside of method, a change made within method will still be visible after return
- For example, if we had

```
void MyMethod(double a) { a = 5.0; }
```

And we had this use of it

```
double b;
```

```
b = 2.0;
```

```
MyMethod(b);
```

```
// b would still be 2.0 here
```

Reminder

The screenshot displays the AnyLogic University software interface. The main workspace shows a state transition diagram for a 'Person' agent. The states are represented by yellow rounded rectangles: 'Susceptible', 'Infective', 'NonPregnant', and 'Pregnant'. Transitions are indicated by arrows, some with associated function names like 'PerformBirth' and 'InfectionStateChange'. The diagram also includes various attributes and functions for the person, such as 'color', 'circleSize', 'isInitiallyInfected', 'sex', 'ethnicity', 'mother', 'appearanceTime', 'InitialAge', 'CurrentAge', 'FinalizeDeath', 'FertilityRateAgeSexEthnicity', 'EstablishOffspringConnectionsBasedOnMothersConnections', and 'EstablishOffspringLocationBasedOnMothersLocation'. The left sidebar shows a project tree with folders for 'Cell', 'Main', 'Simulation: Main', 'Wandering Elephants', 'Elephant', 'Schelling Segregation', 'SIR Agent Based Calibration', 'ABMModelWithBirthDeath', and 'Person'. The bottom panel shows the 'Properties' window for the 'PerformBirth' function, with fields for Name, Access, Return type, and Function arguments.

Person State Transition Diagram:

- States:** Susceptible, Infective, NonPregnant, Pregnant, Death.
- Transitions:**
 - Susceptible to Infective: InfectionStateChange
 - Infective to Susceptible: InfectionStateChange
 - Infective to Death: (unlabeled)
 - NonPregnant to Pregnant: PerformBirth
 - Pregnant to NonPregnant: (unlabeled)
 - NonPregnant to Death: (unlabeled)
 - Pregnant to Death: (unlabeled)
- Attributes:** color, circleSize, isInitiallyInfected, sex, ethnicity, mother, appearanceTime, InitialAge, CurrentAge, FinalizeDeath, FertilityRateAgeSexEthnicity, EstablishOffspringConnectionsBasedOnMothersConnections, EstablishOffspringLocationBasedOnMothersLocation.
- Functions:** reportChildren, children, reportLastTimeInfectedForPersons, dictLastTimeInfectedForPerson, dictLastTimeExposedForPerson, dictAllTimesInfectedForPerson, getAndIncrementNextIdForNewPerson, nextIdForNewPerson, getPersonName, strName, causeError, PregnancyStatus, reportChildren, children, FertilityRateAgeSexEthnicity, PerformBirth, EstablishOffspringConnectionsBasedOnMothersConnections, EstablishOffspringLocationBasedOnMothersLocation.

PerformBirth - Function Properties:

- Name: PerformBirth
- Show name:
- Ignore:
- Show at runtime:
- Access: default
- Static:
- Return type: void
- Use Units:
- Unit:
- Function arguments: (empty table)

EstablishOffspringConnectionsBasedOnMothersConnections

The screenshot displays the AnyLogic software interface. The main workspace shows a state transition diagram for a person agent. The states are Susceptible, Infective, NonPregnant, and Pregnant. Transitions include InfectionStateChange, PregnancyStatus, and actions like reportChildren, children, FertilityRateAgeSexEthnicity, PerformBirth, and EstablishOffspringConnectionsBasedOnMothersConnections. The function definition for EstablishOffspringConnectionsBasedOnMothersConnections is shown in the Properties panel below the diagram.

Function Definition: EstablishOffspringConnectionsBasedOnMothersConnections - Function

General

- Name: EstablishOffspringConnectionsBasedOnMothersConnections
- Show name:
- Ignore:
- Show at runtime:

Code

- Access: default
- Static:
- Return type: void boolean int double String Other: void
- Use Units: Unit:

Function arguments:

Name	Type
offspring	Person
mother	Person

Selection X=272, Y=333

Consider How one Method Calls Another

The screenshot displays the AnyLogic software interface. On the left is a project tree with a 'Person' agent selected. The main workspace shows a state transition diagram for the 'Person' agent. States include 'Susceptible', 'Infective', 'NonPregnant', and 'Pregnant', with transitions for infection and pregnancy. A red arrow points from the 'PerformBirth' function call in the diagram to the code editor below.

Prior to the method call

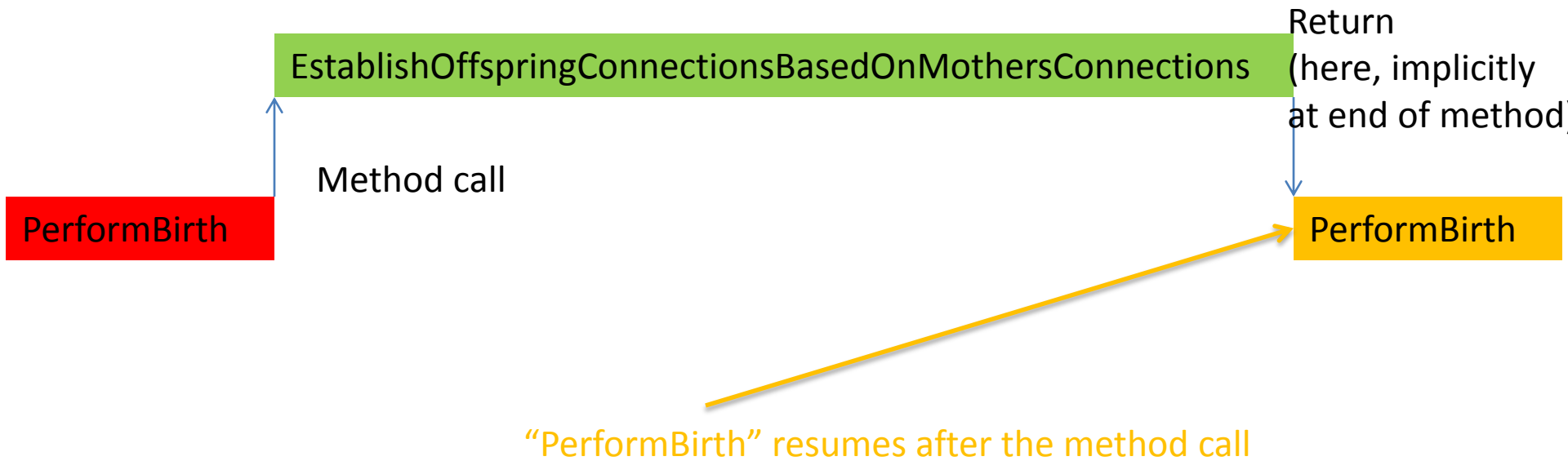
After the method call

PerformBirth - Function

Function body:

```
Person mother = this;
Person offspring = this.get_Main().add_Population((double) 0, ethnicity, RandomSex(), this.IsInfected(), mother);
println("A baby has been born! Baby's id is " + offspring + " while the mother is " + this);
// establish connections of infant
EstablishOffspringConnectionsBasedOnMothersConnections(offspring, mother);
// now position the baby to be close to the mother (otherwise leads to stretching of mother's connections across field of
EstablishOffspringLocationBasedOnMothersLocation(offspring, mother);
children.add(offspring);
```

The “Flow of Control”



The Call Stack

- Because one function can call another (and so on), and then return, we need a way of keeping track of “where we were” when we resume execution after the return
 - What the values of “local” variables are
 - Where in the program
- The call stack is the structure that performs this function
 - Method variables live on the “call stack”
 - When one method calls another, the new method’s variables are placed in an “activation record” or “stack frame” on the call stack
- You will sometimes encounter the call stack in
 - Error messages
 - The AnyLogic & Eclipse Debuggers

Recall: Code for “PerformBirth”

The screenshot displays the AnyLogic software interface. On the left, a project tree shows the hierarchy of models, with 'Person' selected. The main workspace shows a state transition diagram for the 'Person' class. States include 'Susceptible', 'Infective', 'NonPregnant', and 'Pregnant'. Transitions are labeled with actions like 'reportLastTimeInfectedForPersons', 'getAndIncrementNextIdForNewPerson', 'dictLastTimeExposedForPerson', 'dictAllTimesInfectedForPerson', 'getPersonName', 'strName', 'color', 'circlesize', 'CircleSize', 'InfectionStatechart', 'colorForRelation', 'isInitiallyInfected', 'causeError', 'PregnancyStatus', 'reportChildren', 'children', 'sex', 'ethnicity', 'mother', 'appearanceTime', 'InitialAge', 'CurrentAge', 'FinalizeDeath', 'FertilityRate', 'AgeSexEthnicity', 'PerformBirth', 'EstablishOffspringConnectionsBasedOnMothersConnections', and 'EstablishOffspringLocationBasedOnMothersLocation'. A red arrow points from the 'PerformBirth' state to the function code below.

Prior to the method call

After the method call

```
PerformBirth - Function
Function body:
Person mother = this;
Person offspring = this.get_Main().add_Population((double) 0, ethnicity, RandomSex(), this.IsInfected(), mother);
println("A baby has been born! Baby's id is " + offspring + " while the mother is " + this);
// establish connections of infant
EstablishOffspringConnectionsBasedOnMothersConnections(offspring, mother);
// now position the baby to be close to the mother (otherwise leads to stretching of mother's connections across field of
EstablishOffspringLocationBasedOnMothersLocation(offspring, mother);
children.add(offspring);
```

A call from

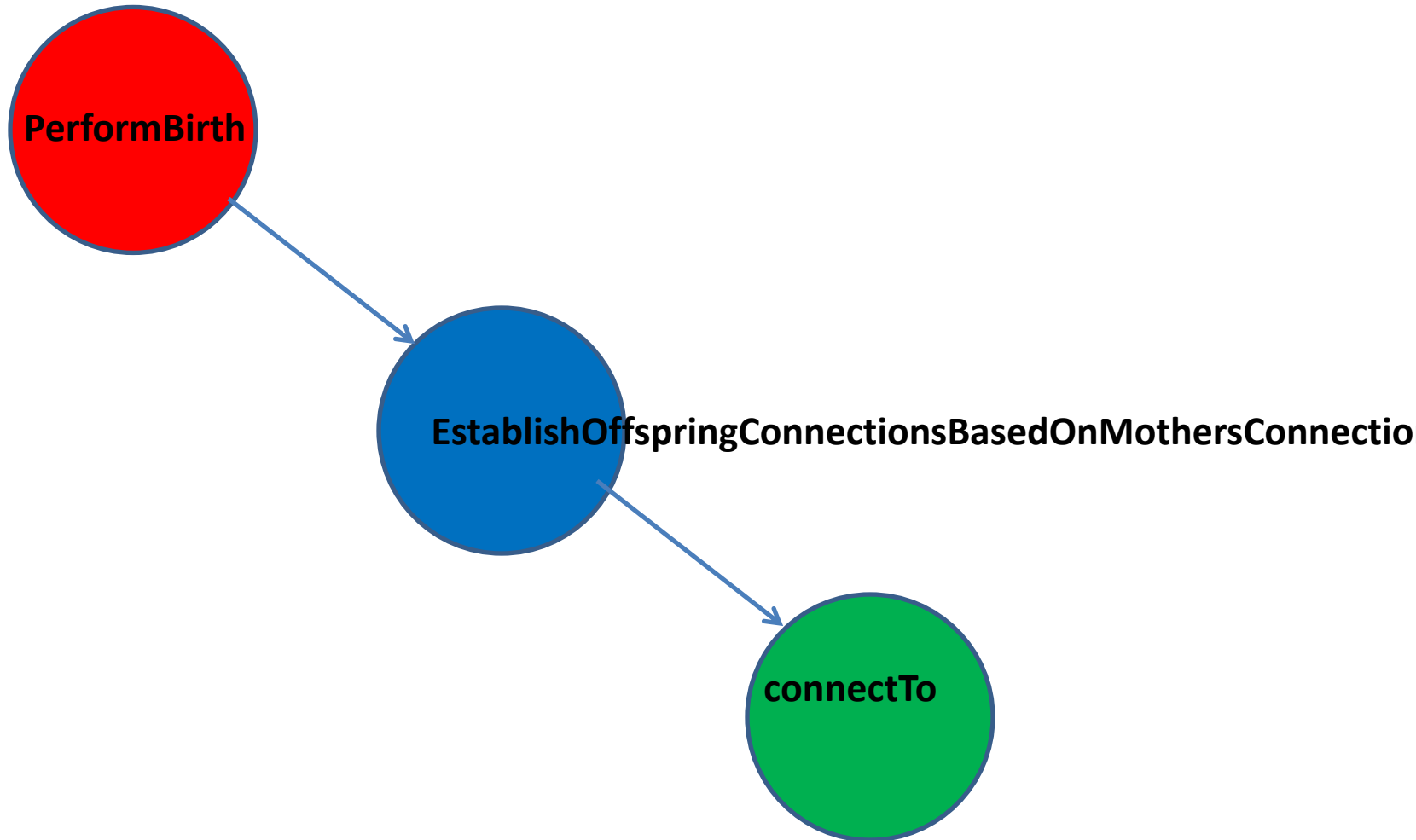
EstablishOffspringConnectionsBasedOnMothersConnections

The screenshot displays the AnyLogic University interface. On the left is a project tree with a hierarchy including 'Person' and 'Simulation: Main'. The main workspace shows a state transition diagram for a 'Person' agent. States include 'Susceptible', 'Infective', 'NonPregnant', and 'Pregnant'. Transitions are labeled with events like 'InfectionStatechart', 'PregnancyStatus', 'reportChildren', and 'children'. A function 'EstablishOffspringConnectionsBasedOnMothersConnections' is highlighted in the diagram. Below the diagram, the 'Properties' and 'Console' tabs are visible. The 'Function body' tab shows the following code:

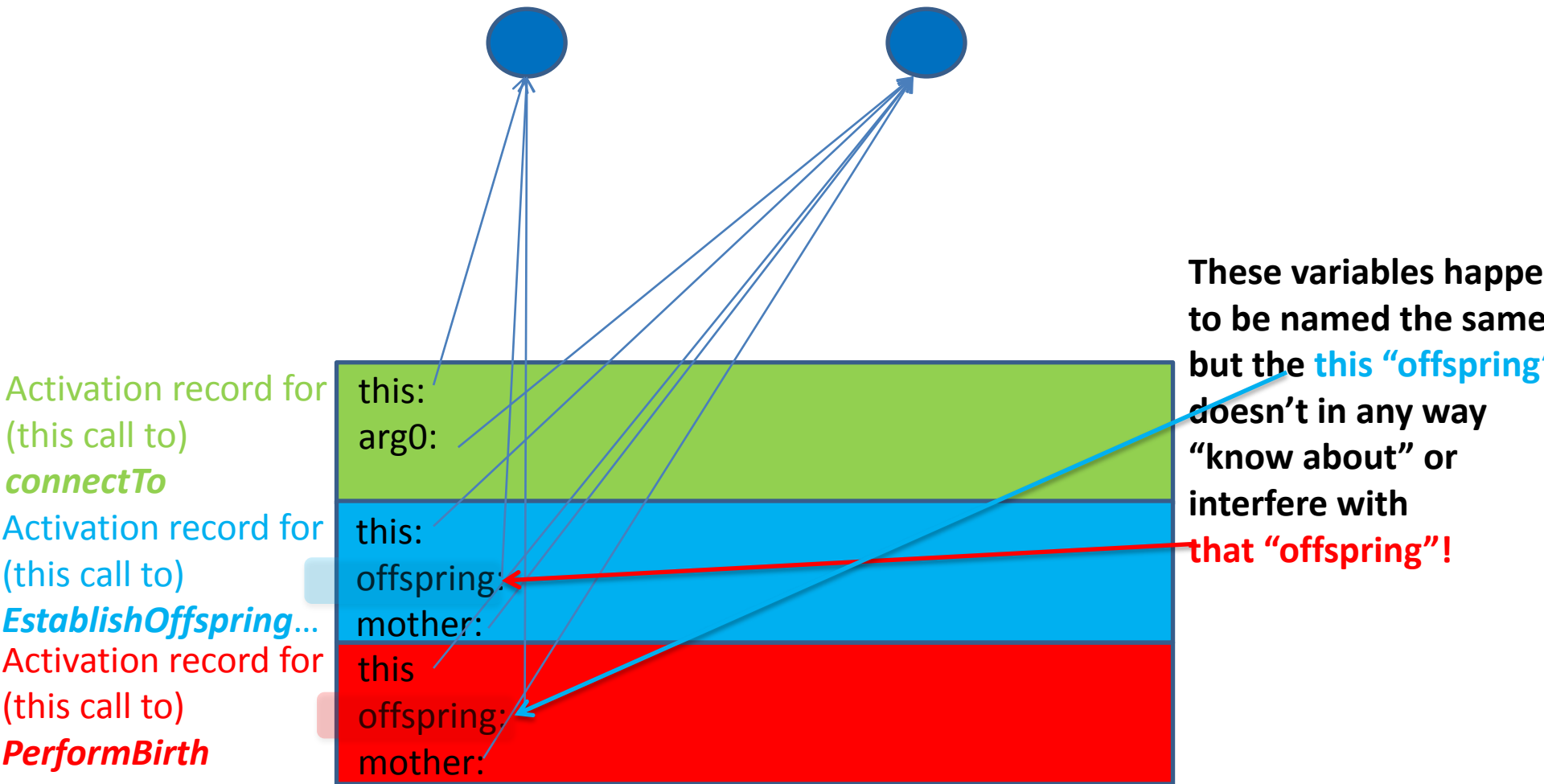
```
Function body:  
// now establish links between the baby and all of the mother's connections  
  
if (mother.getConnections() != null) // guard against a mother with no connections  
    for (Agent a : mother.getConnections())  
    {  
        Person p = (Person) a;  
        offspring.connectTo(p);  
    }  
  
// Finally, establish a link between the baby and the mother  
// (we do this last so we don't have to worry that one of  
// the mother's connections is to this offspring!  
  
offspring.connectTo(mother);  
// note that the "mother" property of the baby has already been set when it was created
```

A red arrow points from the text 'Call to "connectTo" to add the mother as a connection of the offspring' to the line `offspring.connectTo(mother);` in the code editor.

Call Graph (Static)



Call Stack (Dynamic) for Our Example



Reinforcing the Previous Points

- Note that because the names and existence of the parameters inside the method doesn't affect those outside, the preceding would still be true if we had the following

```
void MyMethod(double a) { a = 5.0; }
```

And we had this use of it

```
double a;
```

```
a = 2.0;
```

```
MyMethod(a);
```

```
println(a); // a would still be 2.0 here (it is unaffected  
by the fact that we happened to temporarily have  
something called by the same name within "MyMethod")
```

How is this Achieved?

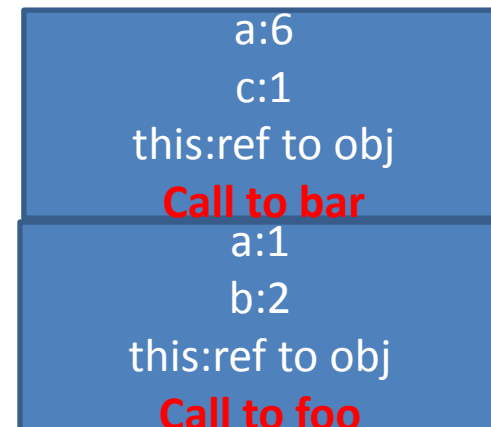
- There is a “call stack”
 - Everytime we make a call to a method, the new parameters get placed “on the stack”

- Suppose we had

`this.foo(1,2)`

and suppose we had

```
void foo(int a, int b) { this.bar((a+b)*2, b-a); }
```



Methods

- A method can do either or both of
 - Compute & return a value
 - Perform some action
- Methods come in 2 types
 - By far most common: Methods associated with objects (i.e. with instances of classes)
 - Less common: Static methods

Methods Associated with Objects

- These are by far the most common class of methods
- When we have a method of this sort, it *always takes an implicit parameter called “this”*
 - This method is always called *on an object*
 - “this” is a reference to the object on which it is called
 - This parameter is not stated explicitly, but is always passed to the method – even if the method appears to take no “arguments” (parameters)

Examples

`p.getConnectionsNumber()`

Within the call to the “`getConnectionsNumber()`” method, “`this`” will refer to the same object as does “`p`” outside

`p.getConnectedAgent(0).getName()`

Within the call to the “`getConnectedAgent`” method, “`this`” will refer to the same object as does “`p`” outside

Within the call to the “`getName`” method, “`this`” will refer to the same object as does `p.getConnectedAgent(0)` outside

Static Methods

- Static methods are associated with a *class*, and not a particular object
 - Syntactically, these look like `Person.nextId()`
- Because they are not associated with a specific object, static methods have no implicit “this” parameter
- These are much closer to our classic notion of a “function” (e.g. `sin(x)`, `sqrt(x)`, `square(x)`)
 - Like static methods, such classic functions have no object associated with them

Determining if a Method Needs to be Static

- Does the method need to depend of the value of some object, or of some information available through “this”
 - This may not be obvious – e.g. to call “get_Main()”, one needs to have a reference to the current Agent
- Given the same values of the arguments, would the value of the function be identical in any context?

Example “Static” (Non-Object-Specific) Method

The screenshot displays a software development environment with a class hierarchy on the left and a configuration panel for a table function on the right.

Class Hierarchy:

- DaysPerTimeUnit
- MeanDaysToNaturallyClearInfection
- ReactivationRateForNormoGlycemicPeople
- ReactivationRateForSmokingStatusAndCKDStage
- ReactivationRateCoefficientForSmokingStatus
- ReactivationRateHazardForNeverSmoker
- ReactivationRateHazardForCurrentSmoker
- RapidnessOfDecreaseInReactivationRateWithTimeSinceQuit
- AgeCoefficientForSmokingInitiation
- SmokingInitiationHazardLogisticSteepnessCoefficient

Table Function Configuration:

AgeCoefficientForSmokingInitiation - Table Function

General

Name: AgeCoefficientForSmokingIn Show Name Ignore Public Show At Runtime

Access: public Static

Interpolation: Linear Out of Range: Nearest Value: 0.0

Table Data:

Argument	Function
0	0
11	0
15	0.25

Buttons: Remove, Paste from Clipboard

A Hierarchy of Functional Abstractions

- We build up higher-level functional abstractions out of lower level ones, e.g.
 - Implementation of PerformBirth calls EstablishOffspringConnectionsBasedOnMothersConnections, which calls connectTo
 - The implementation of FractionOfContactsThatSmoke() might make use of CountSmokingContacts() and CountContacts()
 - We might define CountMen() and CountWomen() with implementation of both just calling CountPopulationOfSex()
- Particularly powerful functional abstractions are those which are parameterized by functions
 - In object-oriented programming, we generally do this by using *polymorphism* – passing objects that match some interface, but whose implementation of that interface can differ

Methods & Exceptions

- Recall: A functions header can be viewed as defining part of a contract that says “if you give me these parameters, I’ll do X for you, and return Y”
 - Normally, extra specifications (via comments or formal guarantees) are needed to make this contract precise
- Because things can go wrong within a method, exceptions may be thrown within it. To be consistent with its stated behavior, the method must either
 - Handle these exceptions itself
 - State explicitly that this sort of exception can be thrown by it (thus delegating handling of it to calling methods)

Recall: 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

Recall: 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 }

Recall: Output to a File

```
try
{
    FileOutputStream fos = new FileOutputStream("file.tab");
    PrintStream p = new PrintStream(fos);
    p.println(datasetName.toString()); //output tab delim vals
}
catch (Exception e)
{
    traceln("Could not write to file.");
}
```

This cleans up after itself in a generic way, but misses the opportunity to elicit much greater information using either the Type of the exception or from the exception object e

Declaration that a Method Can Throw Exceptions: “Throws” Clauses in Signature

The screenshot shows the AnyLogic IDE with the `PajekNetworkFileProcessor` class open. The code includes a `process` method that uses a `try-catch` block to handle `IOException`. A blue box highlights the `try` block, and a red box highlights the `catch` block. A red arrow points from the `throws IOException` clause in the `parseAndProcessPajekEdgeDeclarations` signature to the `IOException` in the `catch` block. A blue arrow points from the `try` block to a text box in the bottom left corner.

```
public static void process(Main associatedMain, String strFilePathAndName)
{
    try
    {
        java.io.BufferedReader br = new java.io.BufferedReader(new java.io.FileReader(strFilePathAndName));
        parseAndProcessPajekVertices(associatedMain, br);
        parseAndProcessPajekArcsAndEdgesDeclarations(associatedMain, br);
    }
    catch (Exception e)
    {
        System.err.println("Unable to impose specified networks due to error:" + e);
        e.printStackTrace(System.err);
    }
}

private static void parseAndProcessPajekEdgeDeclarations(Main associatedMain, BufferedReader br) throws IOException
{
    // because anylogic doesn't support unidirectional edges, we are going to just ignore arcs, and go to edges
    String strLine;

    while ((strLine = br.readLine()) != null)
    {
        if (strLine.matches("^\\s*\\d+\\s+\\d+.*$")) // ok, if this matches something like "5 6", with anything thereafter
            parseAndProcessPajekEdgeDeclaration(associatedMain, strLine);
    }
}

private static void parseAndProcessPajekEdgeDeclaration(Main associatedMain, String strLine)
{
    java.util.Scanner scanner = new java.util.Scanner(strLine).useDelimiter("\\s+"); // skips any whitespace
}
```

This method “cleans up after itself” if something goes wrong, and thus doesn’t need a “throws” clause for these exceptions

This method “delegates” handling of the exception to its callers, rather than handling them itself

throws IOException

Where to Handle Exceptions?

- Often the context of an exception is most clear closer to its source
 - The further one goes up the “call chain” (set of methods calling methods)
 - The less detail one has about where exactly things went wrong
 - The less detail one has for error messages
- Further up, one can more easily abort the overall computation

Shortcomings of AnyLogic Functions (vs. explicitly declared Methods)

- Both primary functions and supporting functions are all visible to others
- They cannot declare that they throw exceptions