

Subtyping & Subclassing:
A Brief Glimpse
Additional Java Tips

Nathaniel Osgood

CMPT 858

4-5-2011

Recall: 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

Recall: Defining the “Interface”

- Knowing the signature of something we are using is necessary but grossly insufficient
 - If could count only on the signature of something remaining the same, would be in tremendous trouble: could do something totally different
 - We want some sort of way of knowing what this thing does
 - We don't want to have to look at the code
- We are seeking a form of *contract*
- We achieve this contact through the use of *specifications*

Recall: 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 previously talked about this
- 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

Encapsulation: Key to Abstraction by *Specification*

- *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

Two Common Mechanisms for Defining Interfaces

- Interface alone: explicit java “interface” constructs
 - Interface defines specification of contract
 - Interface provides no implementation
- Interface & implementation: Classes (using java “class” construct)
 - A class packages together data & functionality
 - Superclasses provide interface & implementations
 - *Abstract classes* as mechanism to specify contract & define some implementation, but leave much of the implementation unspecified
- **We will focus on this**

What is a Class?

- A class is like a mould in which we can cast particular objects
 - From a single mould, we can create many “objects”
 - These objects may have some variation, but all share certain characteristics – such as their behaviour
 - This is similar to how objects cast by a mold can differ in many regards, but share the shape imposed by the mould
- In object oriented programming, we define a class at “development time”, and then often create multiple objects from it at “runtime”
 - These objects will differ in lots of (parameterized) details, but will share their fundamental behaviors
 - Only the class exists at development time
- Classes define an interface, but also provide an *implementation* of that interface (code and data fields that allow them to realized the required behaviour)

Recall: A Familiar Analogy

- The distinction between model design time & model execution time is like the distinction between
 - Time of Recipe Design: Here, we're
 - Deciding what exact set of steps we'll be following
 - Picking our ingredients
 - Deciding our preparation techniques
 - Choosing/making our cooking utensils (e.g. a cookie cutter)
 - Time of Cooking: When we actually are following the recipe
 - A given element of the recipe may be enacted many times
 - One step may be repeated many times
 - One cookie cutter may make many particular cookies

Cooking Analogy to an Agent Class: A Cookie Cutter

- We only need one cookie cutter to bake many cookies
- By carefully designing the cookie cutter, we can shape the character of many particular cookies
- By describing an Agent class at model design time, we are defining the cookie cutter we want to use

Familiar Classes in AnyLogic

- Main class
- Person class
- Simulation class

Work Frequently Done with Objects

- Reading “fields” (variables within the object)
- Setting fields
- Calling methods
 - To compute something (a “query”)
 - To perform some task (a “command”)
- Creating the objects

Distinction between Class and Object

- Sometimes we want information or actions that only relates to the class, rather than to the objects in the class
 - Conceptually, these things relate to the mould, rather than to the objects produced by the mould
 - For example, this information may specify general information that is true regardless of the state of an individual object (e.g. agent)
 - We will generally declare such information or actions to be “static”

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

Subtyping Relationship (Informal)

- We say that type A is a subtype of type B if we can safely substitute an A where a B was expected (e.g. substitute in a *Person* argument where an *Agent* was expected by the parameter)
- A subtype must be in some sense “compatible” with its supertype
 - This compatibility is not merely a matter of signatures, but also involves *behaviour*
 - It is not possible for a compiler to verify the behavioural compatibility of a subtype & supertype
- If we are expecting a B, we should not be “surprised” by the behaviour of an A

Domain-Specific Subtyping

- Frequently we will have a taxonomy of types of objects (classes) that we wish to model
 - People
 - Chiropractors
 - Physiotherapists
 - Licensed Practical Nurses
 - Registered Nurses
 - Patients
 - Orthopedic surgeons
 - Radiologists

We may group objects into classes, but there are commonalities among the classes as well!

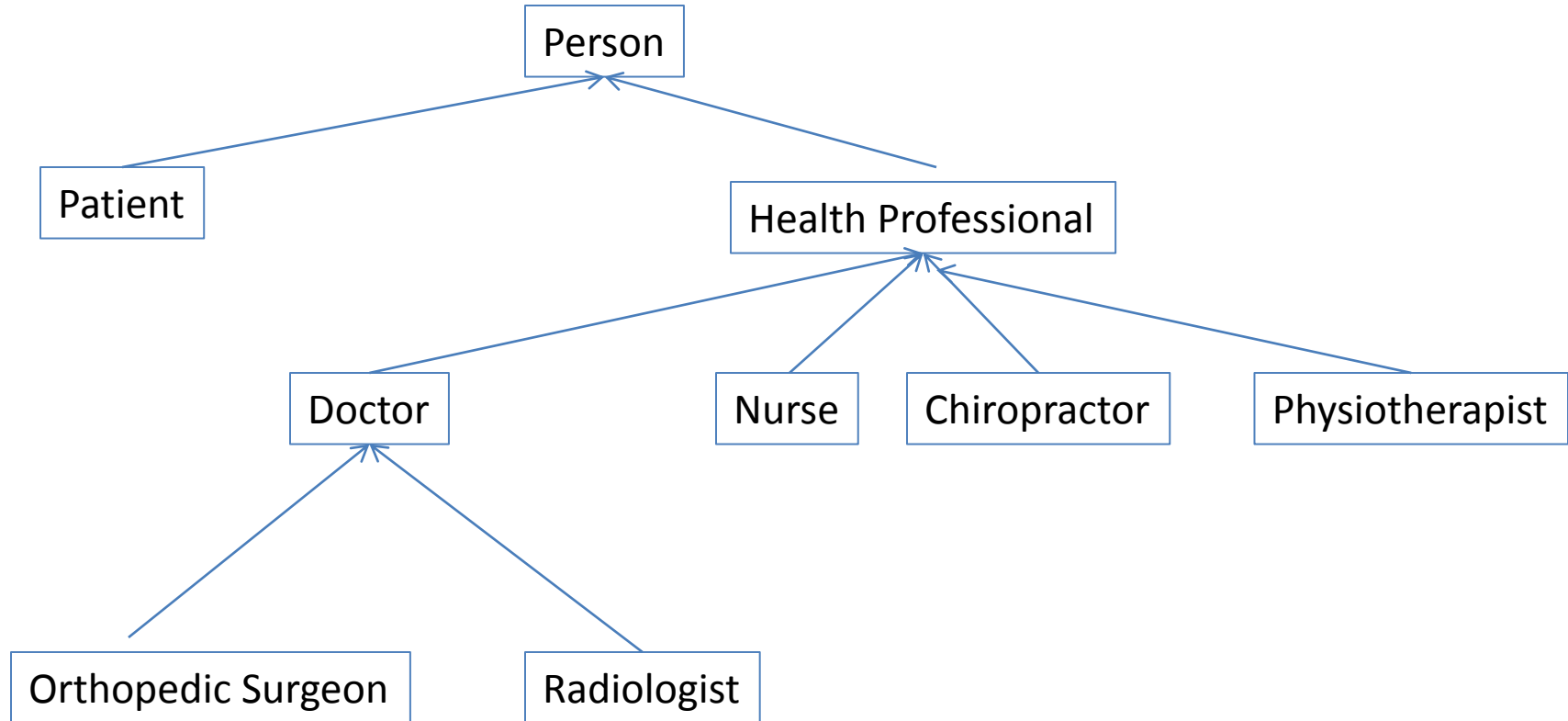
Commonality Among Groups

- Frequently one set of objects (C) is just a special type of another (D)
 - All of the C's share the general properties of the D's, and can be treated as such – but C's have other, more specialized characteristics as well
- For example,
 - Radiologists & Orthopedic surgeons are both types of doctors
 - Licensed Practical Nurses and Registered Nurses are types of nurses
 - Chiropractors, Physiotherapists, Doctors and Nurses are types of health professionals
 - All health professionals and patients are types of people, and share the characteristics of people (e.g. susceptibility to aging, illness and death)

Example

- “Person” interface might provide methods including (but not limited to)
 - IsInfected
 - Infect
 - Age
 - Sex
- In addition to the above, a “HealthProfessional” interface might provide a method “RecentPatients” yielding patients seen by the prof. over a period of time (e.g. the most recent year)
- The “Doctor” interface might further provide a method ResidencyInsitution()

Health Professional Hierarchy



Some Benefits of Type Hierarchies

- Polymorphism – we can pass around an object that provides the subtype as an object that provides the supertype. (e.g. any method expecting a person argument can take a Doctor radiologist)
- Understanding
 - Capturing specialization hierarchies
- Reuse
 - Code can be written for supertypes, but reused for subtypes
- Extensibility
 - Open/closed principle (ideally no need to modify code of superclass when add a subtype)

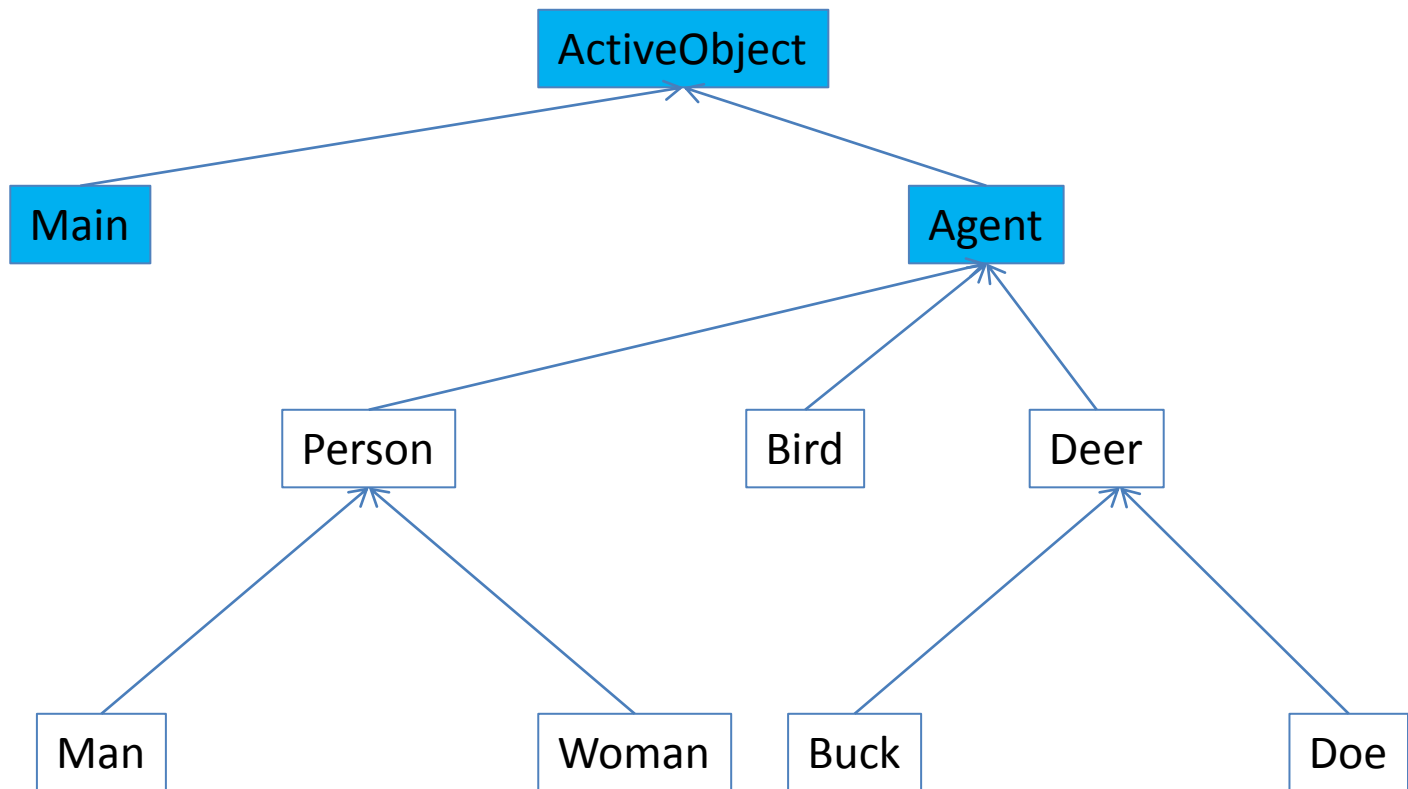
Polymorphism

- We can pass around an object that provides the subtype as an object that provides the supertype.
- Polymorphism enables decoupling of
 - Apparent type
 - Actual type
- Programming against apparent type interface
- Dispatching is against actual type
- E.g. Reference to Dictionary, but actual object is a hash table

AnyLogic Subtyping Relationships

- AnyLogic models are built around a set of classes with subtype relationships to each other
- The presence of these subtype relationships allows us to pass instances (objects) of a subtype around as if it's an instance of the supertype

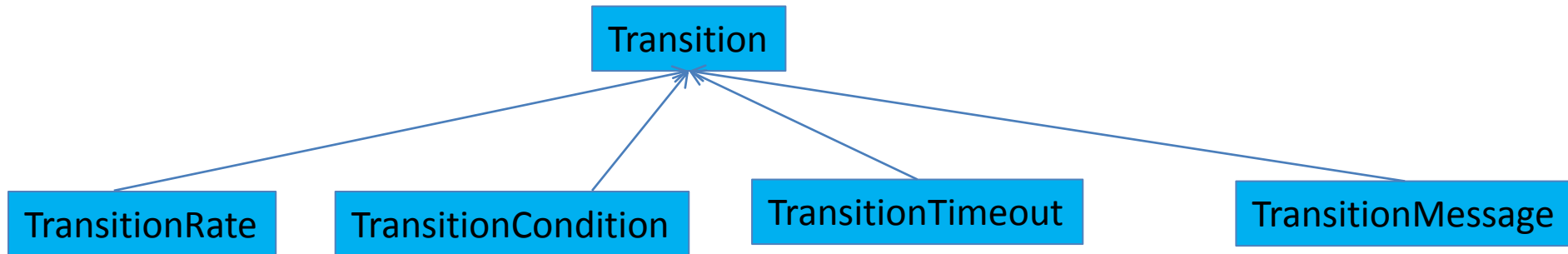
One AnyLogic Hierarchy



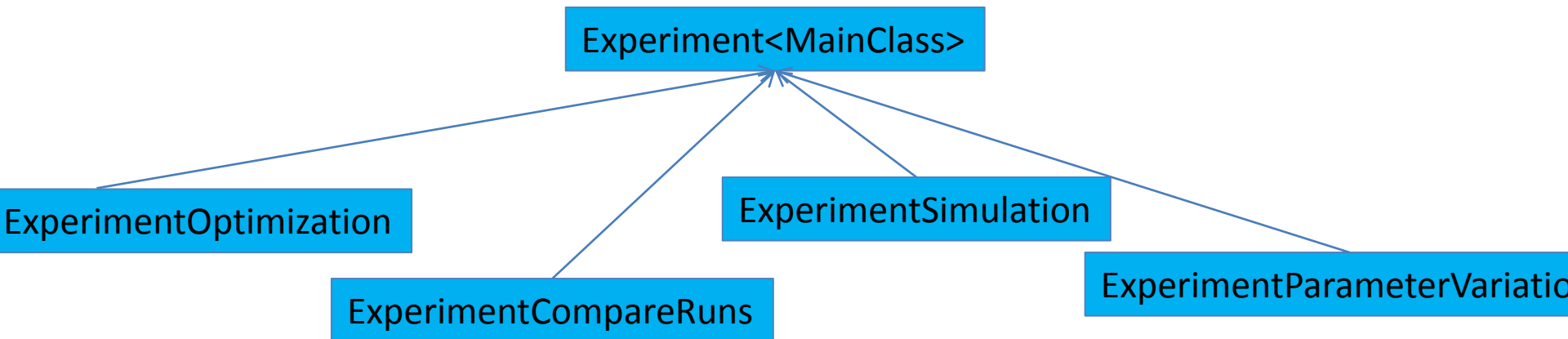
Nodes colored in blue are built in to AnyLogic. The other nodes could be generated automatically (e.g. “Person”, “Bird”, “Deer”) or built (“Man”/”Woman”, “Buck”/”Doe”) as part of a model

Other AnyLogic Hierarchies

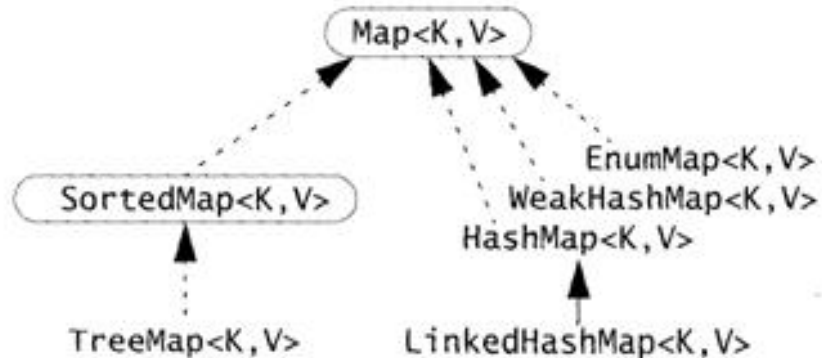
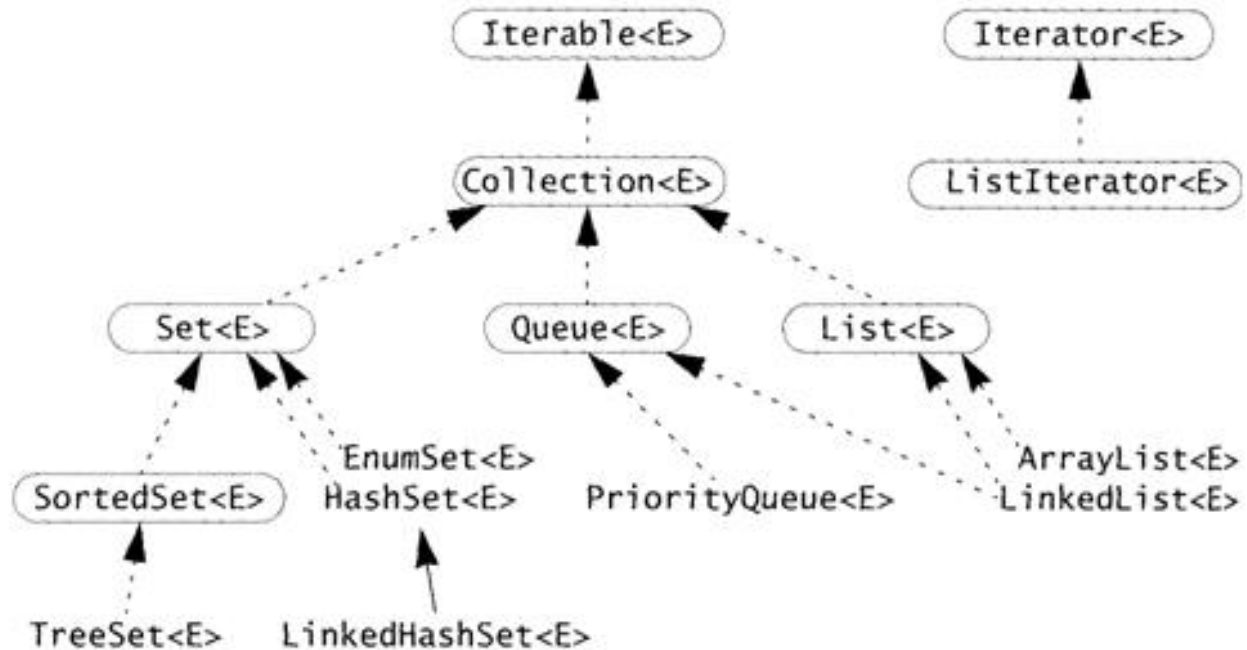
Transitions in Statecharts



Model Experiments



Java.util Type Hierarchies



Java.io Type Hierarchies

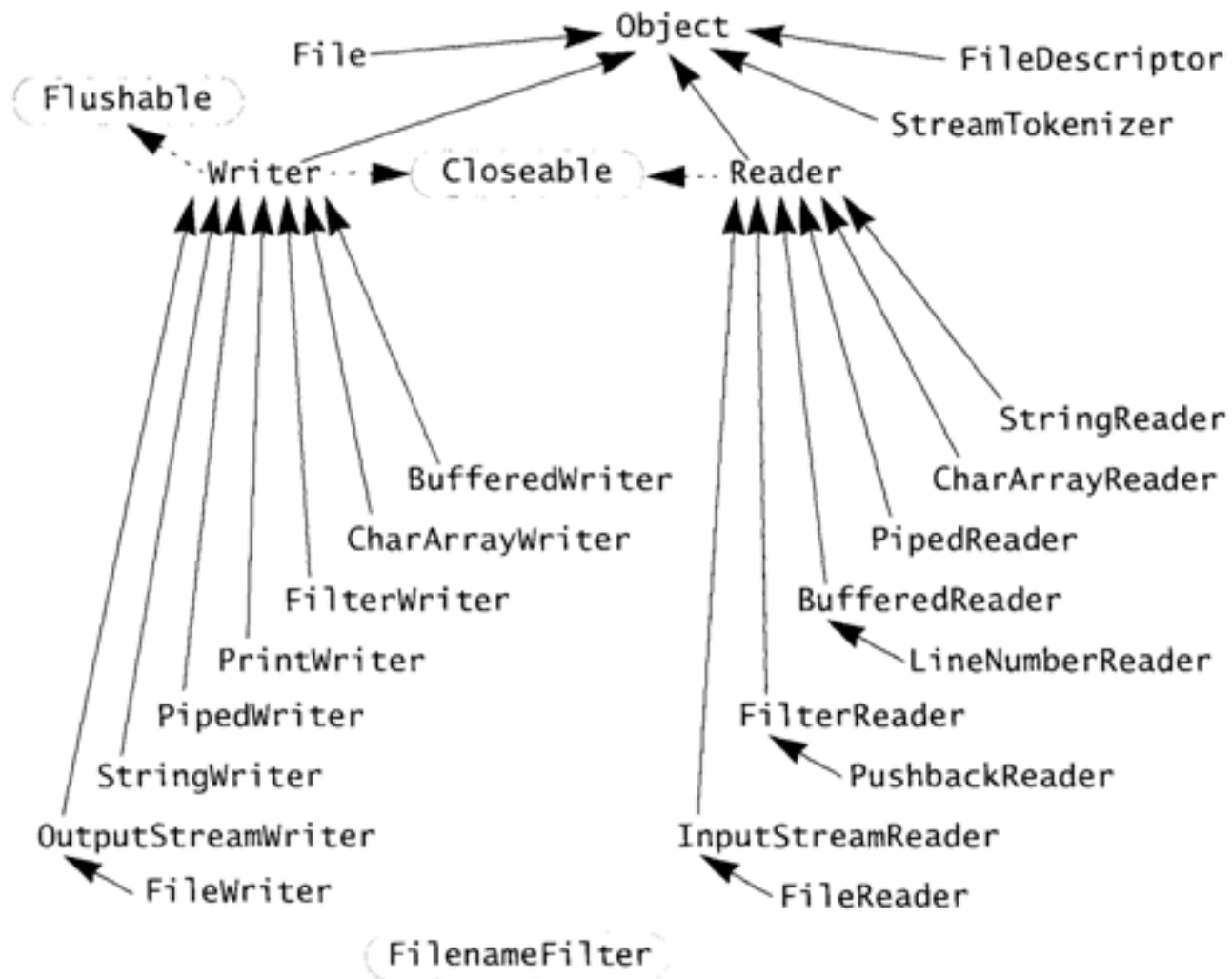


FIGURE 20-2: *Type Tree for Character Streams in java.io*

Subtyping AnyLogic Objects

- One of the most powerful ways of customizing AnyLogic's behavior is by subtyping classes in AnyLogic that are either built-in or auto-generated
- Examples
 - ResourceUnit
 - Entity
- Here, instances of your class can circulate as if it's an instance of the original class

Capturing Hierarchies via Subtyping

- We can capture a hierarchy such as that in the previous slide by
 - Defining interfaces
 - Each interface would specify the methods that are to be supported by any object that provides (supports) that interface
 - Setting up “subclass” relationships of these interfaces through the use of the “extends” keyword

Scoping

- When information is placed in a certain context (e.g. within an object, or “static” things in a class) we have to retrieve it from those places

Subclassing

- “Subclassing” is a special type of subtyping that also allows the subtype to reuse (“inherit”) the *implementation* of the supertype
- This means that, to achieve a small modification for the supertype behavior, the subtype doesn’t have to go through and re-implement everything that is supported by the supertype
- Subclassing brings two things
 - Subtyping
 - Provides e.g. polymorphism
 - Code reuse
 - via inheritance of methods, fields

Contrasting Tradeoffs

Interfaces

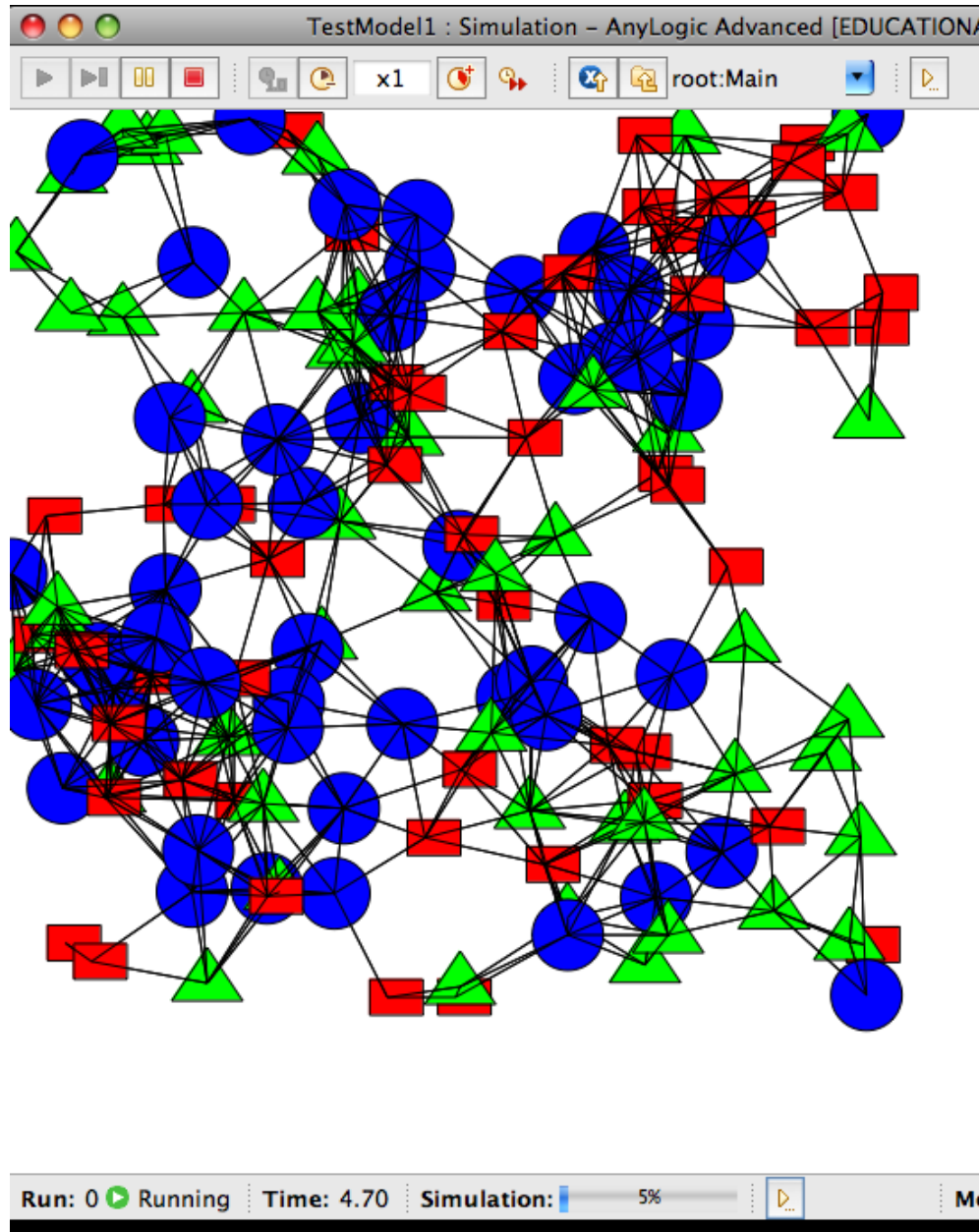
- Advantages
 - More flexible
 - Capture non-hierarchical relationships
 - Easily added to definition of an existing class
 - Enables “mixin” like style
 - Cleaner type & inheritance hierarchy
- Disadvantages
 - Cannot easily extend existing interfaces
 - No default implementations can be provided

Class-Based Inheritance

- Advantages
 - Easier extension with new functionality
 - Permits implementation reuse
- Disadvantages
 - Subtype constraint (LSP) violation
 - Desire to reuse code can lead to deliberate ignoring
 - Inheritance can lead to accidental violation & violation of open-closed principal
 - Distort inheritance hierarchy
 - Abstract classes pushed up
 - Combinatorial explosion for dual interfaces
 - Single inheritance limits to tree
 - Multiple inheritance is dangerous
 - Semantically tricky
 - Confusing

(Some Items Adapted from Bloch, *Effective Java*, 2001, Pearson Education)

Network with Multiple Agent Classes



Realizing Multiple Agent Classes Sharing Same Network

- Create an agent superclass
- Create multiple subclasses of that superclass
 - In “Properties”
 - indicate that “Extends” superclass
 - Provide constructor to associate with agent population & Main class
- For the Agent population, use a replication of 0
- Create Startup code for “Main” that adds the various types of agents to the model
 - This uses code adopted from Java code output by build

Common Problems

- References to concrete classes leads to multiple changes for a simple conceptual change
 - Can be fixed by consistent programming against *interfaces*
- *Claimed subtypes are not behavioural subtypes of supertypes*
 - *Subclassing for code reuse or mistaken notion of specialization(“is-a”) causes flawed design, defects*

We'll comment on these

“Fraudulent Subtypes”

- When building a subtype/class hierarchy, we specify (“tell the compiler”) which units are subtypes of which
 - In Java, this is specified using “implements” & “extends”
- The compiler generally accepts user information on type structure at face value
 - Full checking is not possible
 - Limited checking (e.g. on signatures) errs on the side of being conservative (may report error even in cases where legitimate) (e.g. incompatible signatures)
- It is very easy to create a subtype that is not a safe behavioural subtype of its alleged supertype

Subclasses: A Particularly Common Type of Fraudulent Subtype

- Misplaced use of subclassing can very easily create classes that are *not* sub**types**
- When such “fraudulent subclasses” are used with polymorphism, the code can break easily
- Two prime ways in which code can break
 - Implementers deliberately chooses subtype behaviour that makes it behaviorally incompatible with the superclass type (supertype)
 - Implementers try to make this a behavioral subtype, but don't have the necessary guarantees on superclass implementation(*later*)

Why Are Fraudulent Subclasses So Common?

- Subclassing is abused as way to reuse code via inheritance
 - This is a matter of convenience
 - Want to avoid redefining a broad set of methods just to override a few
- Classes are used to group a set of objects where an “is-a” relationship applies but which are not behavioural subtypes
 - E.g. Square “is a” type of rectangle

Liskov Substitution Principle

- Principle is key to recognizing a legitimate subtype relationship
- *The principle reflects the need to reason safely about types in the presence of polymorphism*
- Statement of Principle (Liskov & Wing)
 - “Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be true for objects y of type S where S is a subtype of T .”*

Persistent Metaphor: Service Contracts

- Desire for encapsulation \Rightarrow Clear understanding of what is guaranteed
- Example: Franchise of Delivery service
 - Question: Given parent company guarantees, what must a franchise offer to be legitimate?
 - Precondition: Condition for guarantee to hold
 - Parent company: Customer must drop off package by noon
 - Ok Franchise: Customer can drop off up to 3pm
 - Illegal Franchise: Customer must drop off package by 9am
 - Postcondition: Service guarantee if precondition met
 - Parent company: Delivery is by 5pm the next day
 - Ok Franchise: Delivery is by noon the next day
 - Illegal Franchise: Delivery is by next year

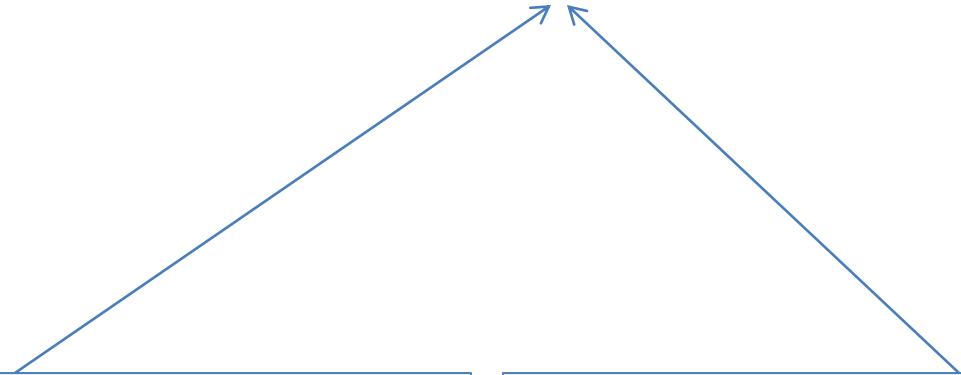
Contract Hierarchy

Fedex

Deliver()

// Precondition: Package available by 12 noon

// Postcondition: Package delivered by 5pm next day



Fedex Franchise 1

Deliver()

// Precondition: Package available by 3pm

// Postcondition: Package delivered
by noon next day

Fedex Franchise 1

Deliver()

// Precondition: Package available by 3pm

// Postcondition: Package delivered
by noon next day

Liskov Substitution Principle: Intuition

- Consider a situation in which a programmer is creating code with a variable v whose
 - Apparent type is $T1$
 - Actual type is a subtype $T2$ of $T1$ (due to polymorphism)
- To avoid risk this code will have to be changed with every new subtype of $T1$, it is critical that anything the programmer can rely upon for a variable of type $T1$ is also true for v (despite being of type $T2$)
 - Any type $T2$ that which departs from the contract of $T1$ can break this code
 - Bear in mind that other code may treat v as a $T2$

Parameterized Types

- Via “Generic” classes and interfaces, Java supports *parameterized types*
 - Here, the definition of one class can be defined with respect to an arbitrary number of classes that are provided via “Type parameters”
- Examples: `ArrayList<ClassName>`, `Set<ClassName>`

This is an array list and set that can hold any type of classes (as specified by “ClassName”)
- A given use of such a “Generic” class will specify a specific class name for the type parameter
 - e.g. `Set<Person>`, `ArrayList<Double>`, `List<Deer>`
- The definition of the generic can restrict the types that can be used for the type parameter via constraints

Examples of Type Parameterization in AnyLogic

- Experiment<MainClass> (and other experiment classes)
- ResourcePool<ResourceUnit>
- NetworkResourcePool<ResourceUnit>
- ActiveObjectArrayList<ActiveObject>
Typically used (among other things) for the population in a main class
- ActiveObjectList<ActiveObject>

Defensive Programming

- Naming conventions
- Formatting
- Separate
 - Commands (side effects)
 - Queries (pure)
- Don't do side effects in e.g. macros
- Mark temporary code (e.g. scaffolding) using a convention
- Avoid manifest constants
- Consolidate condition checks in methods or objects (“specification” pattern)
- Minimize variable lifetime & span between references
- Use “dog tags” to recognize overwrites, double deallocation
- Check return values, value legality
- Display results of successive language processing
- Naming conventions
- Always handle all cases (even illegal)
- Overriding default methods as a rule
- Always put in { } after if
- Beware empty catch blocks
- Use *finally* blocks
- Don't reuse temporary variables
- Initialize vars, member data as they are declared or in constructor
- Use pseudocode programming process

Other suggestions

- Strive for transparent code
 - Use variable name conventions
 - Consistent formatting
- Strive for higher abstraction level
 - Spot commonality
- Use explicit in and out parameters
- Use restrictive modifiers
 - Const
 - Private/protected
- Encapsulation
 - Information hiding
 - Program to interfaces
 - Design by contract
- Use type abstractions (generics)
- Delegate
- Use enumerations
- Encapsulate repetitive actions
- Move whole & partial conditionals to methods

Bad Smells (Many from McConnell, Code Complete 2.0)

- Duplicate code
- Long routine
- Deep/long if/loops
- Inconsistent interface abstraction
- Lots of special cases
- Poor cohesion
- Too many parameters
- Single update yields changes to many places
- Keep on creating ad-hoc data structures/classes
- Global variables
- Primitive types
- Need to update multiple inheritance hierarchies
- Subclasses not really subtypes
- Related items spread among multiple classes
- Method deals more with other classes than its own
- Need to know implementation of other class
- Unclear name
- Setup & takedown code around call

Style & Convention

- Naming Conventions
- Commenting
- Metadata (e.g. Javadocs)
- Indentation
- Module Naming
- Construct placement
- Compiler Pragma & Mechanisms

Naming Conventions

- Naming conventions are a powerful tool
- Benefits
 - Reduce risk of errors
 - Easier understanding of others' code
 - Easier understanding of code in future
 - Lower risk of name clashes
 - Easier search for desired item (e.g. method/variable/class)

Java Naming Conventions

- Distinguish Typographic & Grammatical
- Packages
 - Short lowercase alphabetic (digits rare)
 - Start with organization internet domain name (e.g. ca.usask)
- Classes/interfaces
 - First word of each capitalized (TagHasher)
 - Avoid all but most common abbreviations
 - Generally nouns/noun phrase
 - Interfaces sometimes adjective

Java Naming Conventions 2

- Method & Fields
 - Same as classes but first letter lowercase
 - Const static fields all uppercase, “_” as separ.
 - “Action” methods named with verb
 - “is” for booleans
 - Query: noun/noun phrase or verb w/“get” prefix
 - Converters: “toX”, primitiveValue
- Local variables
 - Same as members but can be short, context-dependent

Scope Naming Conventions

Table 11-3 Sample Naming Conventions for C++ and Java

Entity	Description
<i>ClassName</i>	Class names are in mixed uppercase and lowercase with an initial capital letter.
<i>TypeName</i>	Type definitions, including enumerated types and typedefs, use mixed uppercase and lowercase with an initial capital letter.
<i>EnumeratedTypes</i>	In addition to the rule above, enumerated types are always stated in the plural form.
<i>localVariable</i>	Local variables are in mixed uppercase and lowercase with an initial lowercase letter. The name should be independent of the underlying data type and should refer to whatever the variable represents.
<i>routineParameter</i>	Routine parameters are formatted the same as local variables.
<i>RoutineName()</i>	Routines are in mixed uppercase and lowercase. (Good routine names are discussed in Section 7.3.)
<i>m_ClassVariable</i>	Member variables that are available to multiple routines within a class, but only within a class, are prefixed with an <i>m_</i> .
<i>g_GlobalVariable</i>	Global variables are prefixed with a <i>g_</i> .
CONSTANT	Named constants are in <i>ALL_CAPS</i> .
MACRO	Macros are in <i>ALL_CAPS</i> .
<i>Base_EnumeratedType</i>	Enumerated types are prefixed with a mnemonic for their base type stated in the singular—for example, <i>Color_Red</i> , <i>Color_Blue</i> .

Booleans

- Base name should give clear sense of condition in question
- Use common convention to indicate boolean
 - “f” prefix (e.g. fOpen)
 - is prefix (e.g. isOpen)
 - “?” suffix (e.g. open? – legal scheme)
- Avoid negation in names (e.g. isNotOpen)

Loop Etiquette

- Make clear what iterating over
- Label index variables with the type of thing being iterated over
- Avoid overly deep loops
 - Confusion
 - Control flow: break/continue
 - Placement of items in loop
 - Consider making internals of loop a separate method/function

Enumerations

- Enumerations help avoid manifest constants, group common names
- Good for bitwise operations : Consider values that will allow this rather than combinatorial names
- If language does not support enumerations, use carefully named global constants
- Leverage compiler checking
- If no class prefix, consider naming enumeration values with prefix giving type enumeration
- Make default enumeration value illegal
- Always explicitly handle all values

Example of Enums in AnyLogic

The screenshot displays the AnyLogic Advanced interface. The main workspace shows a state transition diagram for a **Person** class. The states are **Susceptible**, **Infective**, **NonPregnant**, and **Pregnant**. Transitions include **isInitiallyInfected**, **sex**, **ethnicity**, **appearanceTime**, **InitialAge**, **CurrentAge**, **FinalizeDeath**, **FertilityRateAgeSexEthnicity**, **PerformBirth**, **EstablishOffspringConnectionsBasedOnMothersConnections**, and **EstablishOffspringLocationBasedOnMothersLocation**. The **Properties** window shows the **Person - Active Object Class** with the following code:

```
public enum Sex { Male, Female };  
public enum Ethnicity { FirstNations, Metis, EastAsian, SouthAsian, Ca  
public static java.util.Random random = new java.util.Random();
```

The **Palette** on the right lists various modeling elements: 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, and Enterprise Library.

A Closer Look

Additional class code:

```
public enum Sex { Male, Female };  
public enum Ethnicity { FirstNations, Metis, EastAsian, SouthAsian, Caucasian };
```

Use of Enums to Delineate Possible Parameter Values

The image displays a software development environment with two main windows: a class diagram and a parameter configuration window.

Class Diagram (Person):

- Classes:** Susceptible, Infective, NonPregnant, Pregnant, and Death.
- Associations:**
 - A diamond-shaped association connects Susceptible and Infective.
 - Arrows point from Susceptible to Infective and from Infective to Susceptible.
 - Arrows point from both Susceptible and Infective to Death.
 - An arrow points from NonPregnant to Pregnant.
 - An arrow points from Pregnant to NonPregnant.
 - An arrow points from NonPregnant to a circle labeled PregnancyStatus.
- Attributes:** color, circlesize, CircleSize, isInitiallyInfected, sex, ethnicity, appearanceTime, InitialAge, CurrentAge, FinalizeDeath, FertilityRateAgeSexEthnicity, PerformBirth, EstablishOffspringConnectionsBasedOnMothersConnections, EstablishOffspringLocationBasedOnMothersLocation.

Parameter Configuration Window (sex - Parameter):

- Name:** sex
- Show Name:**
- Ignore:**
- Public:**
- Show At Runtime:**
- Type:** Other: `Person.Sex`
- Default Value:** `Sex.Female`
- Dynamic:**
- Save in snapshot:**
- On Change:** (Empty text box)

Use of Enums to Delineate Possible Parameter Values

The image displays a software development environment with two main windows. The top window, titled "Person", shows a class diagram with the following elements:

- Classes:** Susceptible, Infective, NonPregnant, Pregnant, and Death.
- Relationships:** Susceptible and Infective are connected by a bidirectional arrow. NonPregnant and Pregnant are connected by bidirectional arrows. Arrows point from Susceptible and Infective to Death. A diamond symbol is connected to Susceptible and Infective.
- Parameters:** color, circlesize, CircleSize, isInitiallyInfected, sex, ethnicity, appearanceTime, InitialAge, CurrentAge, FinalizeDeath, FertilityRateAgeSexEthnicity, PerformBirth, EstablishOffspringConnectionsBasedOnMothersConnections, and EstablishOffspringLocationBasedOnMothersLocation.
- Enums:** PregnancyStatus (with NonPregnant and Pregnant states) and Ethnicity (with Metis state).

The bottom window, titled "Properties", shows the configuration for the "ethnicity" parameter:

- Name:** ethnicity
- Show Name:**
- Ignore:**
- Public:**
- Show At Runtime:**
- Type:** Other: `Person.Ethnicity`
- Default Value:** `Ethnicity.Metis`
- Dynamic:**
- Save in snapshot:**
- On Change:** (empty text box)

Generating Random Possible Values

The image displays a software development environment with two main windows. The top window, titled 'Person', shows a state machine diagram. The states are represented by yellow rounded rectangles: 'NonPregnant', 'Pregnant', and 'Infective'. There are also two unlabeled yellow rounded rectangles. Transitions between states are indicated by arrows. A diamond-shaped node is connected to the top-left yellow rectangle. A circle labeled 'Death' is also connected to the diagram. The bottom window, titled 'Properties', shows the configuration for the 'RandomEthnicity' function. It includes fields for Name, Access, Return Type, and Function arguments.

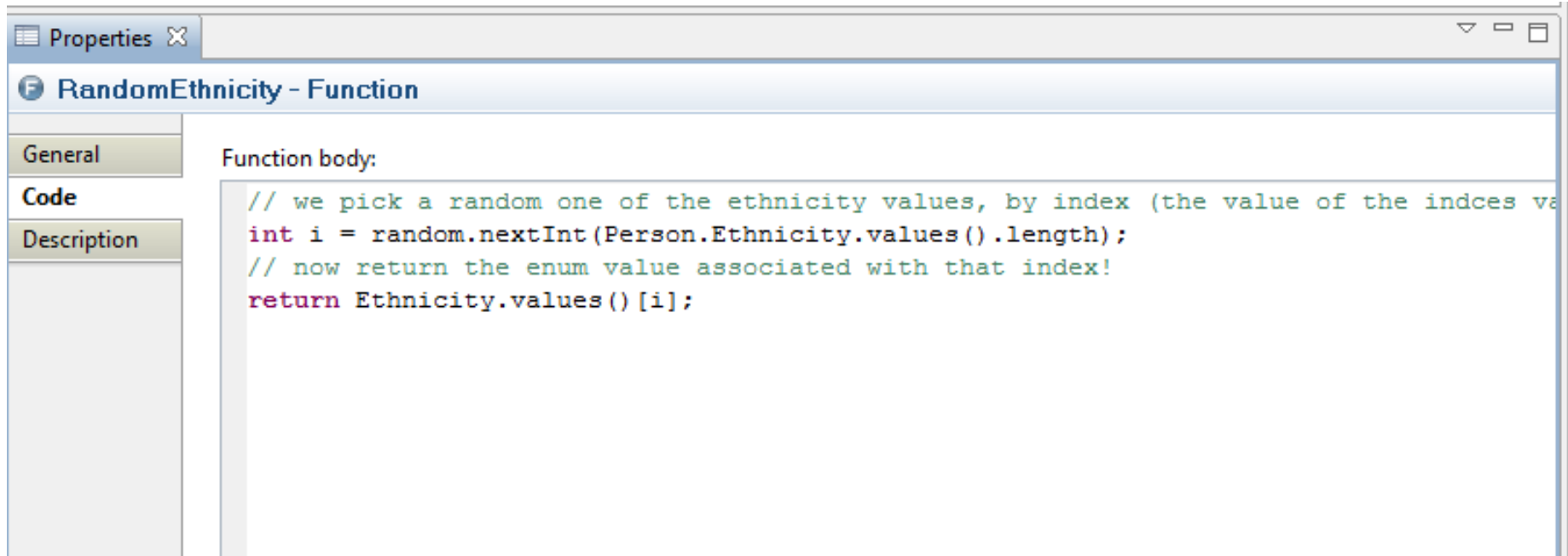
Person State Machine Diagram:

- States: NonPregnant, Pregnant, Infective, and two unlabeled states.
- Transitions: NonPregnant to Pregnant, Pregnant to NonPregnant, NonPregnant to Infective, Pregnant to Infective, Infective to unlabeled state, unlabeled state to Infective, Infective to Death.
- Other elements: CircleSize, isInitiallyInfected, PregnancyStatus, sex, ethnicity, appearanceTime, InitialAge, CurrentAge, FinalizeDeath, FertilityRateAgeSexEthnicity, PerformBirth, EstablishOffspringConnectionsBasedOnMothersConnections, EstablishOffspringLocationBasedOnMothersLocation, RandomSex, RandomEthnicity, RandomAge.

RandomEthnicity - Function Properties:

- Name: RandomEthnicity
- Show Name:
- Ignore:
- Public:
- Show At Runtime:
- Access: default
- Static:
- Return Type: Other: Person.Ethnicity
- Function arguments: (Empty table)

The Associated Code...



The screenshot shows a window titled "Properties" with a sub-tab "RandomEthnicity - Function". The "Code" tab is selected, displaying the following C# code:

```
Function body:  
  
// we pick a random one of the ethnicity values, by index (the value of the indices va  
int i = random.nextInt(Person.Ethnicity.values().length);  
// now return the enum value associated with that index!  
return Ethnicity.values()[i];
```

Use Modifiers

- Use “const” or “final” (including for parameters in Java) to prevent side-effects
 - Examples
 - Prevent modification to *this* in method
 - Prevent assignment to parameter
 - Poor man’s option: use “const” in name
- Static can prevent needless memory use

Process Complexity: A Barrier to Quality

System Dynamics Modeling

- Medium+ scale SD projects generate a large # & diversity & versions of related artefacts
- Careful coordination of these artefacts is important for ensuring quality insights
- Efficient coordination is important for productivity
- Existing tools offer limited support for such coordination
- Difficulties limit what can be accomplished

Recall: Process Suggestions

- 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

Java Modifiers

- “Final”
 - Indicates that the value of a field cannot be changed
 - Indicates that method cannot be “overridden”
- “Static”: associated with a class (only one variable associated with the class – no how many objects of the class are circulating)
- Annotations
- Access modifiers

Access Modifiers

- **Public:** Visible to (fields: modifiable by) other classes, in “package” or not
- **Private:** Not visible to (fields: modifiable by) by any other classes
- **Package (default):** Only visible to (fields: modifiable by) other classes
- **Protected:** Only visible (fields: modifiable by) in this class & subclasses

Annotations

- Allows custom indications concerning program elements e.g.
 - Field declarations
 - Class declarations
- Uses: Compiler processing/advising, deployment-time, custom runtime information availability
- Syntax: Indicated by a word (“identifier”) with “@” sign
 - Optional additional information

Custom Annotations Example

- Example:

```
@interface DataProvenance
{
    String originalReference();
    String
intermediateDerivationLocation()
default "";
    String sourcePersonName();
}
```

```
@interface Uncertainty
{ double stdDev() default -1; }
```

```
@DataProvenance(originalReferen
ce = "TB Control 2009 Report",
intermediateDerivationLocation = "
Historical TB Data v12.xls",
sourcePersonName = "Nate
Osgood")
@Uncertainty(stdDev=3.5) double
meanYearsBetweenRelapses = 15;
```

Annotation Retention

- Annotation information can be used at different time
 - Different levels of retention of annotation information are possible, via the Retention meta-annotation & the RetentionPolicy enum
- Options
 - SOURCE: Only preserved during compilation
 - CLASS: preserved in class information, but not necessarily available at runtime
 - RUNTIME: Annotations are preserved in class representation & are available at runtime for access via reflection (except for local variables, which are not preserved)

Annotations with Compiler Support

- `@Override` (compiler issues error if not found to be overriding method)
- `@Deprecated` (compiler warns when used)
- `@SuppressWarnings` (can instruct compiler to suppress one or both of 2 common types of warnings)

Valuable Uses of Annotations

- Documentation
 - Authorship information
 - Revision information
- Data
 - Provenance
 - Pedigree
- Capturing intentions
- Consistency with
- Verifying that goal is being met (e.g. that are, in fact, overriding)