Encapsulation, Interfaces, Subtyping & Subclassing: A Brief Glimpse

Nathaniel Osgood
MIT 15.879

May 9, 2012

Recall: A Key Motivator for Abstraction: Risk of Change

- Abstraction by specification helps lessen the work required when we need to modify the model
- 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: 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

Recall: 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)

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

Encapsulation: Key to Abstraction by *Specification*

- *Separation of contract ("interface", standards,terms) from implementation (allowing multiple implementations to satisfy the interface)* facilitates modularity
- Specifications indicate expected behavior of anything providing the interface (what is required, what is promised)
- This distinction between interface & implementation forms a key role in many areas of practical human activity

Encapsulation: 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
  - Client code should only be counting on what is promised in contract – not all of the implementation details
- *Substitutability*: Can replace one implementation by another
- *Reuse*: Can abstract over implementations that differ only in their details to only use one mechanism: e.g. Shared code using interface based polymorphism

Practical Examples of Separation of Interface from Implementation

- Dealing with taxi (via meter, credit card support)
  - Doesn't depend on which particular taxi you have!
- MBTA/MTA/BART/other metro train tickets (tickets bought anywhere work with any train station)
- Franchises:
  - Delivery companies (FedEx, Purulator, etc.)
    - Rules of delivery
    - Pricing
  - Car rental (certain insurance guaranteed, etc.)
- US Post Office (stamp requirements, pricing, ZipCodes &c)
- Computer-related examples
  - Protocols (e.g. WiFi, DHCP, TCP/IP)
    - If adhere to protocols, can deal with any router
  - Android Applications (app can work w/any android phone)
  - "PC": Windows/Linux Hardware Specifications

Recall: Defining the "Interface"

- We are seeking a form of *contract*
- We achieve this contact through the use of *specifications*

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

Linking Interface & Implementation

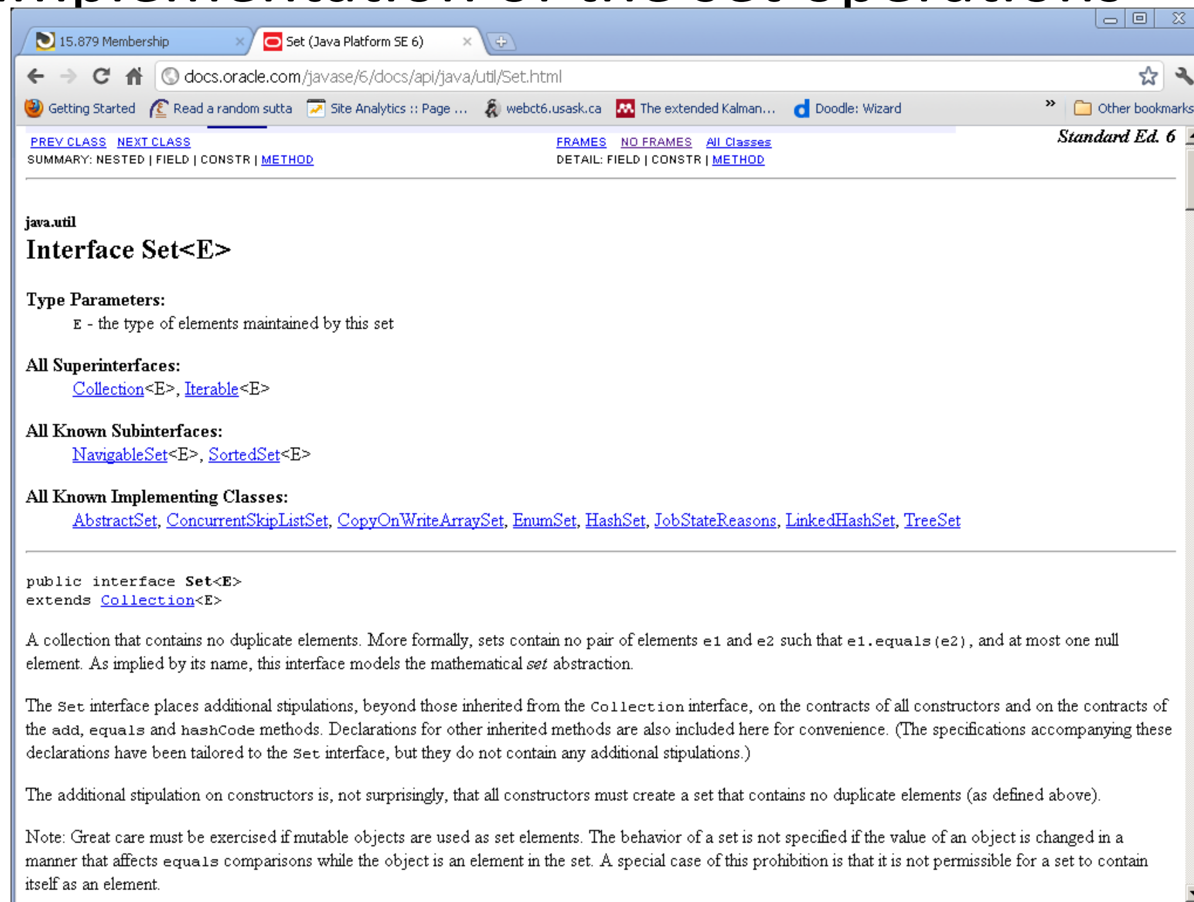
- In Java, if class C implements an interface, we use the keyword *implements*
- Implementing an interface allows one to deal with that class through that interface
  - At a technical level, the type of class C is a "subtype" of the type associated with the interface

Example of Implementing an Interface

```
interface Presentable { void draw(); void disappear(); }
interface SerialNumbered { int Id(); }
class X implements Presentable, SerialNumbered
  {
       void draw() { … }
       void disappear() { … }
        int Id() { …. }
  }
```

Example of Implementing Interfaces

- *Set* interface could be implemented by many particular classes
  - Each provides an implementation of the set operations
  - Some might be optimized for small sets, others for large sets



15.879 Membership | Set (Java Platform SE 6)

docs.oracle.com/javase/6/docs/api/java/util/Set.html

Getting Started | Read a random sutta | Site Analytics :: Page ... | webct6.usask.ca | The extended Kalman... | Doodle: Wizard | Other bookmarks

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes | *Standard Ed. 6*

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

java.util

Interface Set<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

NavigableSet<E>, SortedSet<E>

All Known Implementing Classes:

AbstractSet, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet

```
public interface Set<E>
extends Collection<E>
```

A collection that contains no duplicate elements. More formally, sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`, and at most one null element. As implied by its name, this interface models the mathematical *set* abstraction.

The `Set` interface places additional stipulations, beyond those inherited from the `Collection` interface, on the contracts of all constructors and on the contracts of the `add`, `equals` and `hashCode` methods. Declarations for other inherited methods are also included here for convenience. (The specifications accompanying these declarations have been tailored to the `Set` interface, but they do not contain any additional stipulations.)

The additional stipulation on constructors is, not surprisingly, that all constructors must create a set that contains no duplicate elements (as defined above).

Note: Great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects `equals` comparisons while the object is an element in the set. A special case of this prohibition is that it is not permissible for a set to contain itself as an element.

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

Subtype Hierarchies

- Below, we concentrated on subtyping between an interface and its implementation
  - This allowed the implementation to be used wherever the interface was required
- We can, in fact, construct extensive hierarchies via subtyping

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
  - Physiotherapists, Doctors, Nurses and Chiropractors 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)

Domain-Specific Subtyping

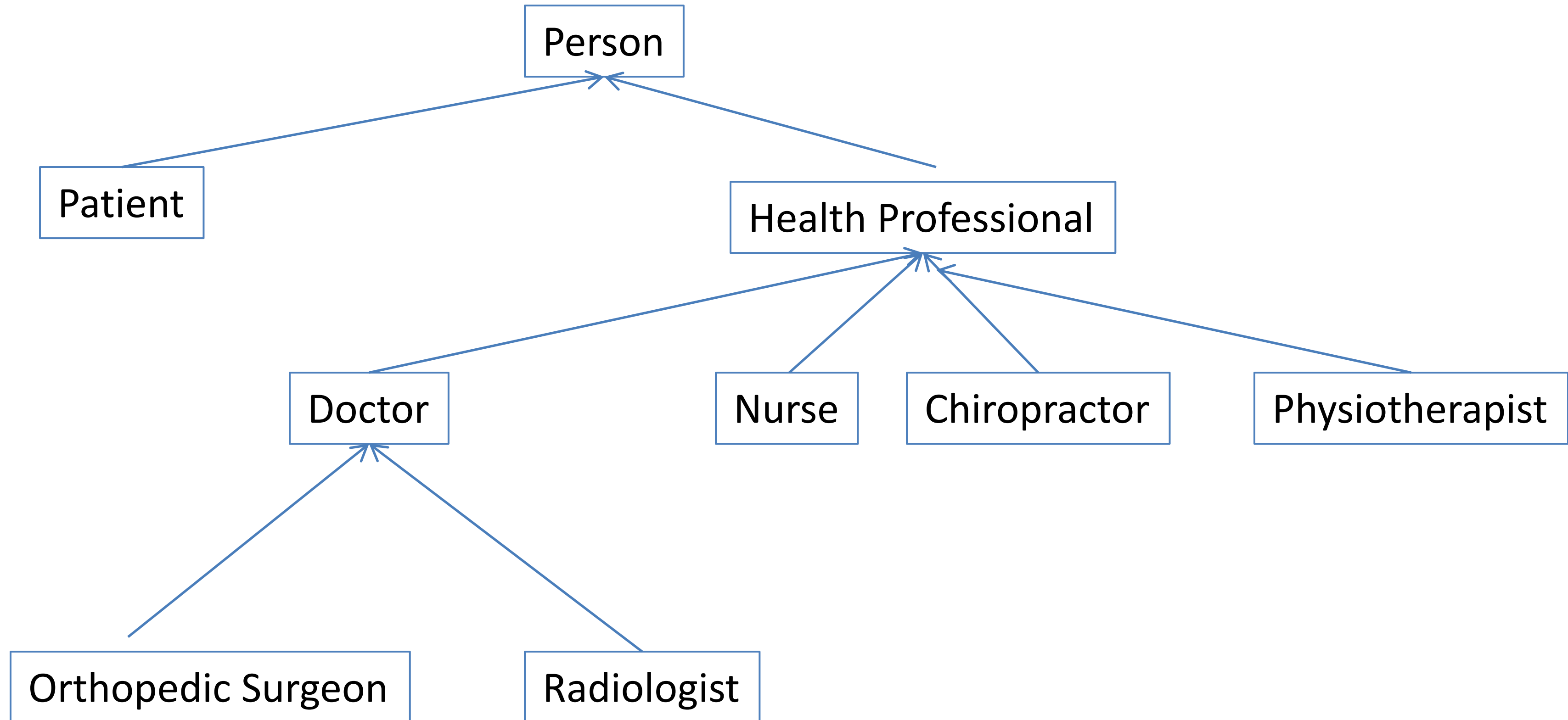
- Frequently we will have a taxonomy of types of objects (classes) that we wish to model
  - People
  - Chiropractors
  - Physiotherapists
  - Licensed Practical Nurses
  - Registered Nurses
  - Patients
  - Orthopedic surgeons
  - Radiologists

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

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 professional over some specified 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

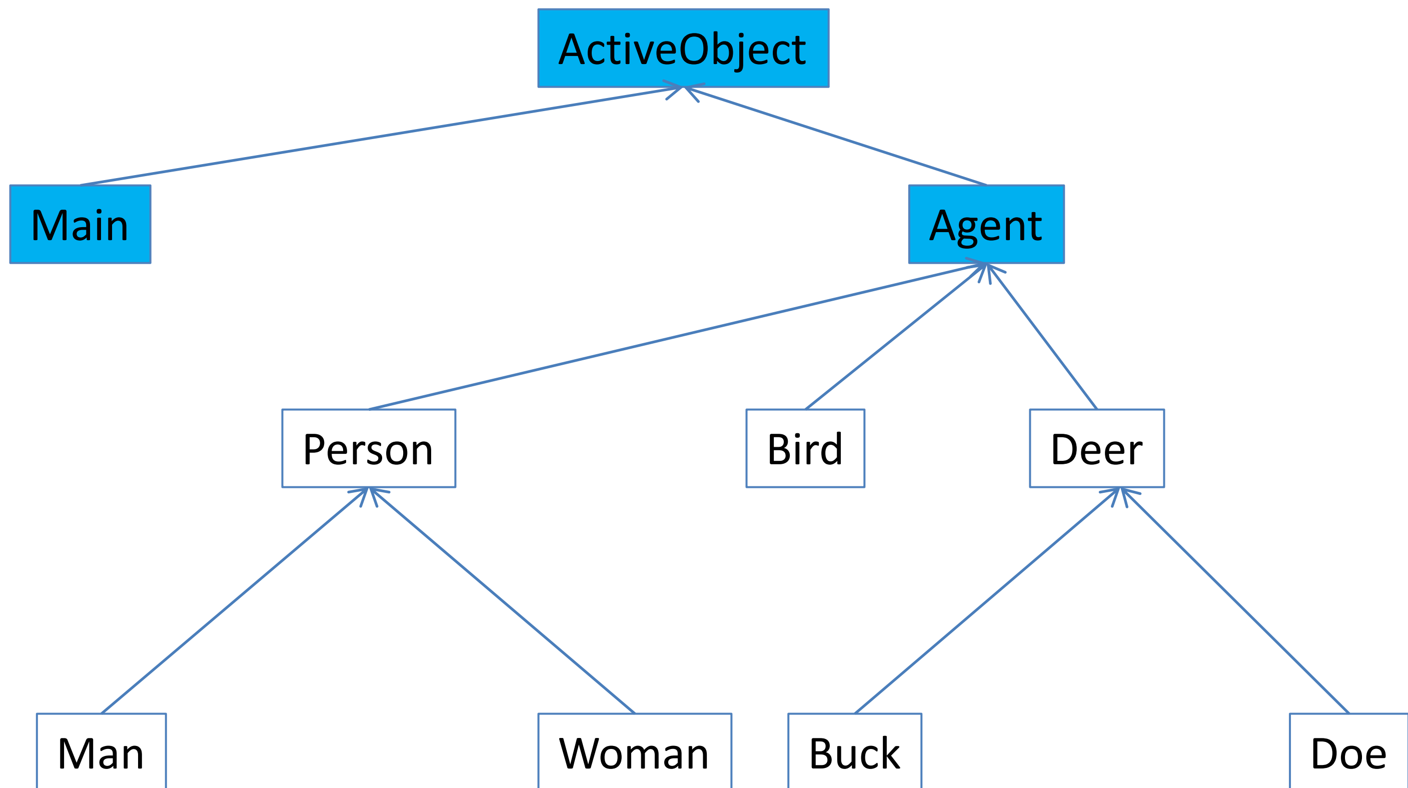
- We can pass around an object that provides the subtype as an object that provides the supertype.
- Polymorphism enables decoupling of
  - Apparent type
  - Actual type
- Programming against apparent type interface
- Dispatching is against actual type

- 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

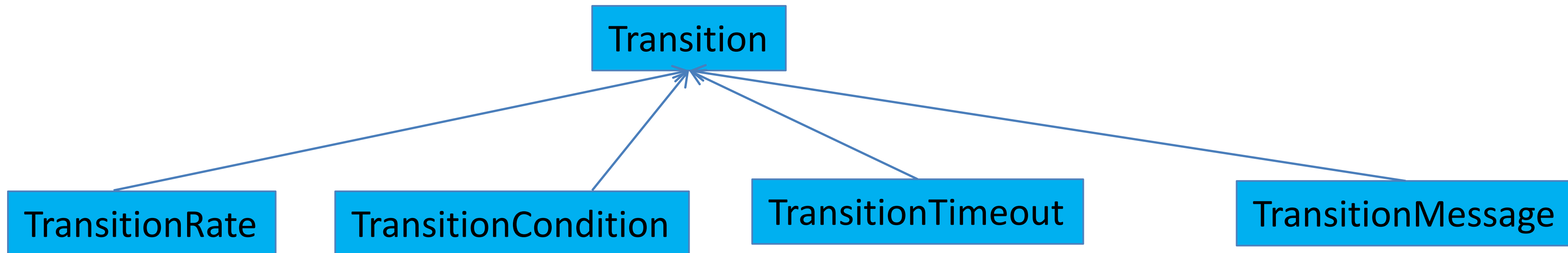
One AnyLogic Hierarchy



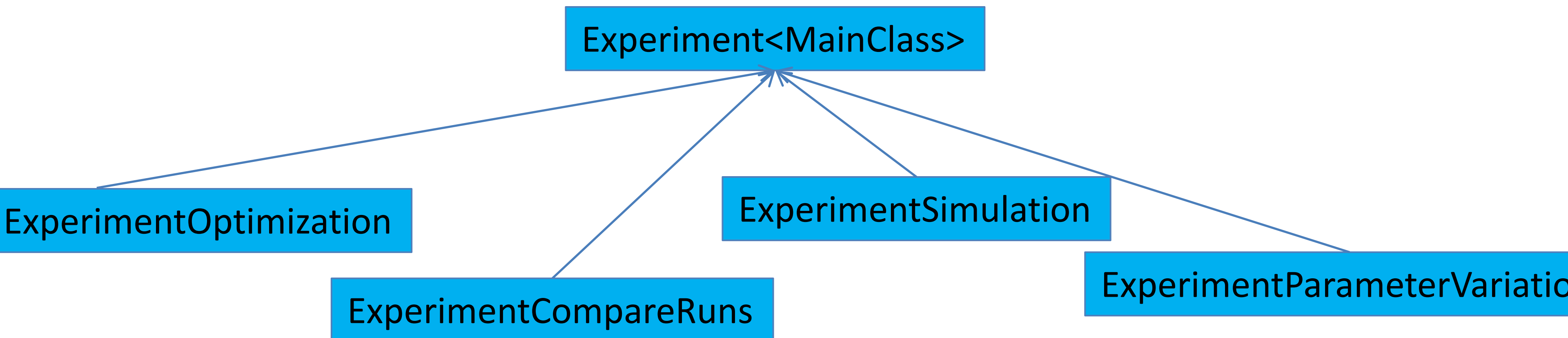
Nodes colored in blue are built in to AnyLogic. The other nodes could be generated automatically (e.g. “Person”, “Bird”, “Deer”) or built (“Man”/”Woman”, “Buck”/”Doe”) as part of a model

Other AnyLogic Hierarchies

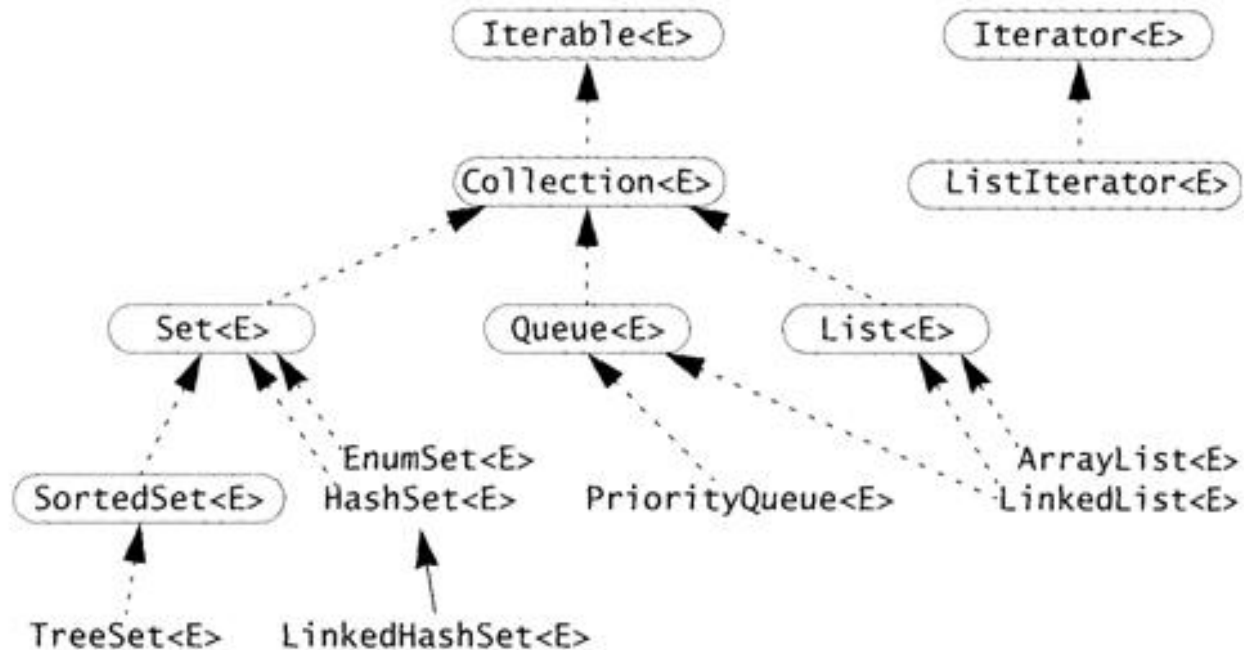
Transitions in Statecharts

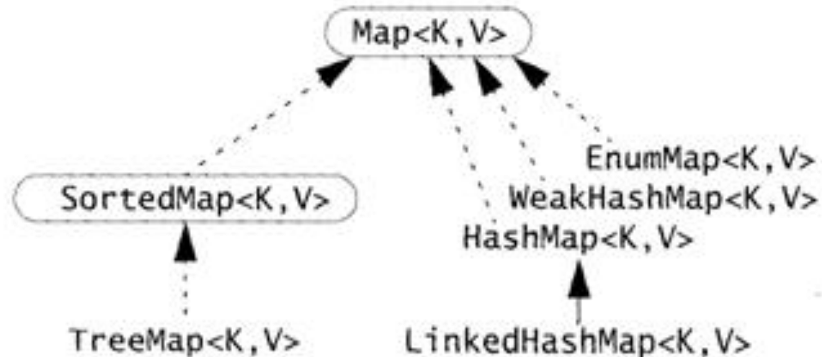


Model Experiments



Java.util Type Hierarchies





Java.io Type Hierarchies

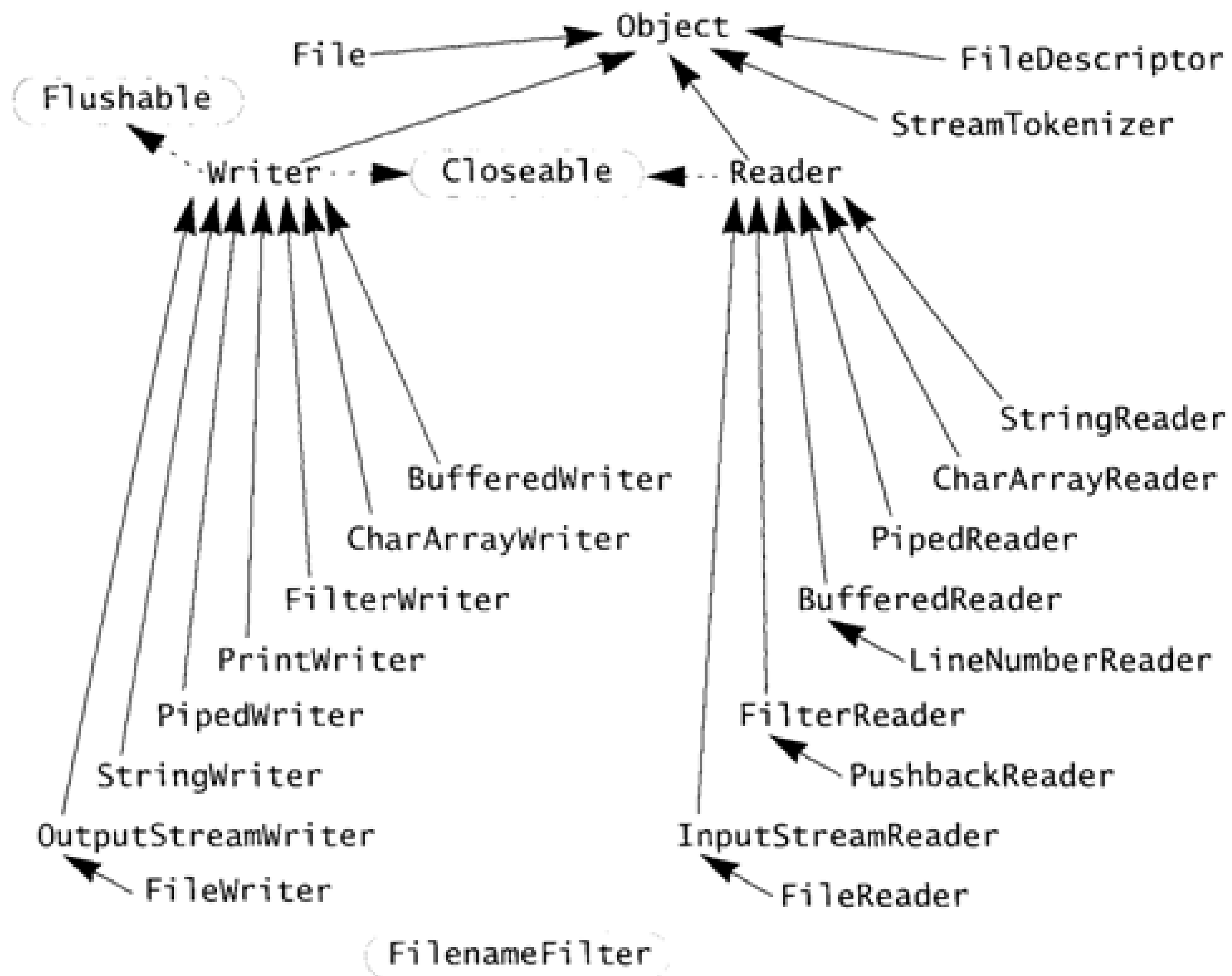


FIGURE 20–2: *Type Tree for Character Streams in* **java.io**

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 slides 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

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

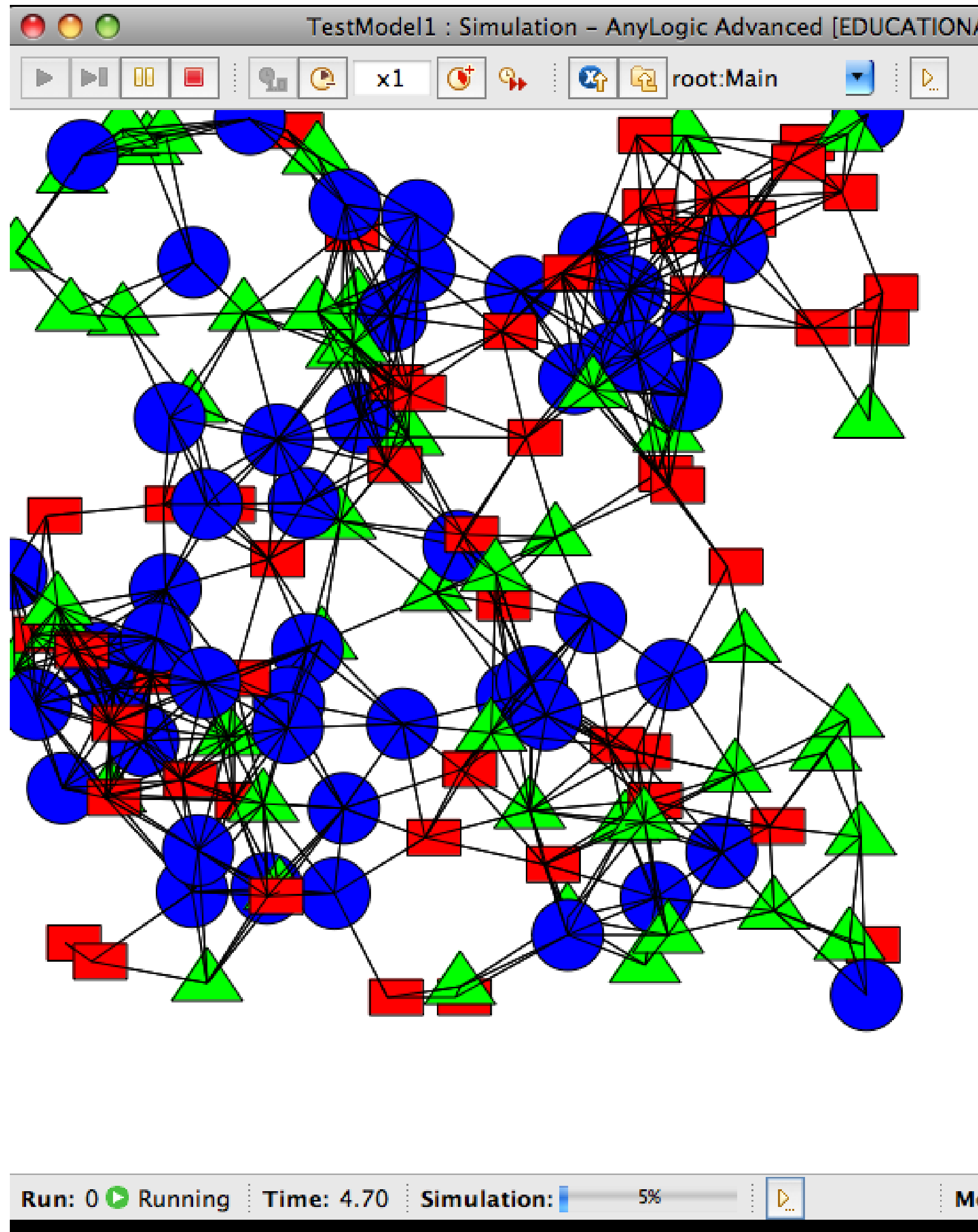
- Advantages
  - More flexible
    - Capture non-hierarchical relationships
    - Easily added to definition of an existing class
    - Enables “mixin” like style
  - Cleaner type & inheritance hierarchy
- Disadvantages
  - Cannot easily extend existing interfaces
  - No default implementations can be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

Hands on Model Use Ahead

Load Sample Model:

SIR Agent Based

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

Male Agents

Male is a subclass of **Person**

Male - Active Object Class

Extends (single ActiveObject or Agent subclass): Person

Female Agents

AnyLogic Advanced [EDUCATIONAL USE ONLY]

File Edit View Model Window Help

Project
- EclipseDebuggingExar
  - Main
  - Person
  - DebuggingSession
  - ProfilingSimulation
  - Simulation: Main
- HPVModel
  - Female
  - Main
  - Male
  - Person
  - AgentFactory
  - Simulation: Main
  - Simulation1: Main
  - Simulation2: Main

Description	Lo...
By convention, Ja...	N...

Female | Person

Female Smoking

Female Vaccination

Female Fertility

Progression Rates

Regression Rates

Cure Rates

Reocurrence Rates

Palette

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

More Libraries...

Properties | Console

Female - Active Object Class

General

Advanced

Agent

Parameters

Description

Imports section:

Extends (single ActiveObject or Agent subclass):

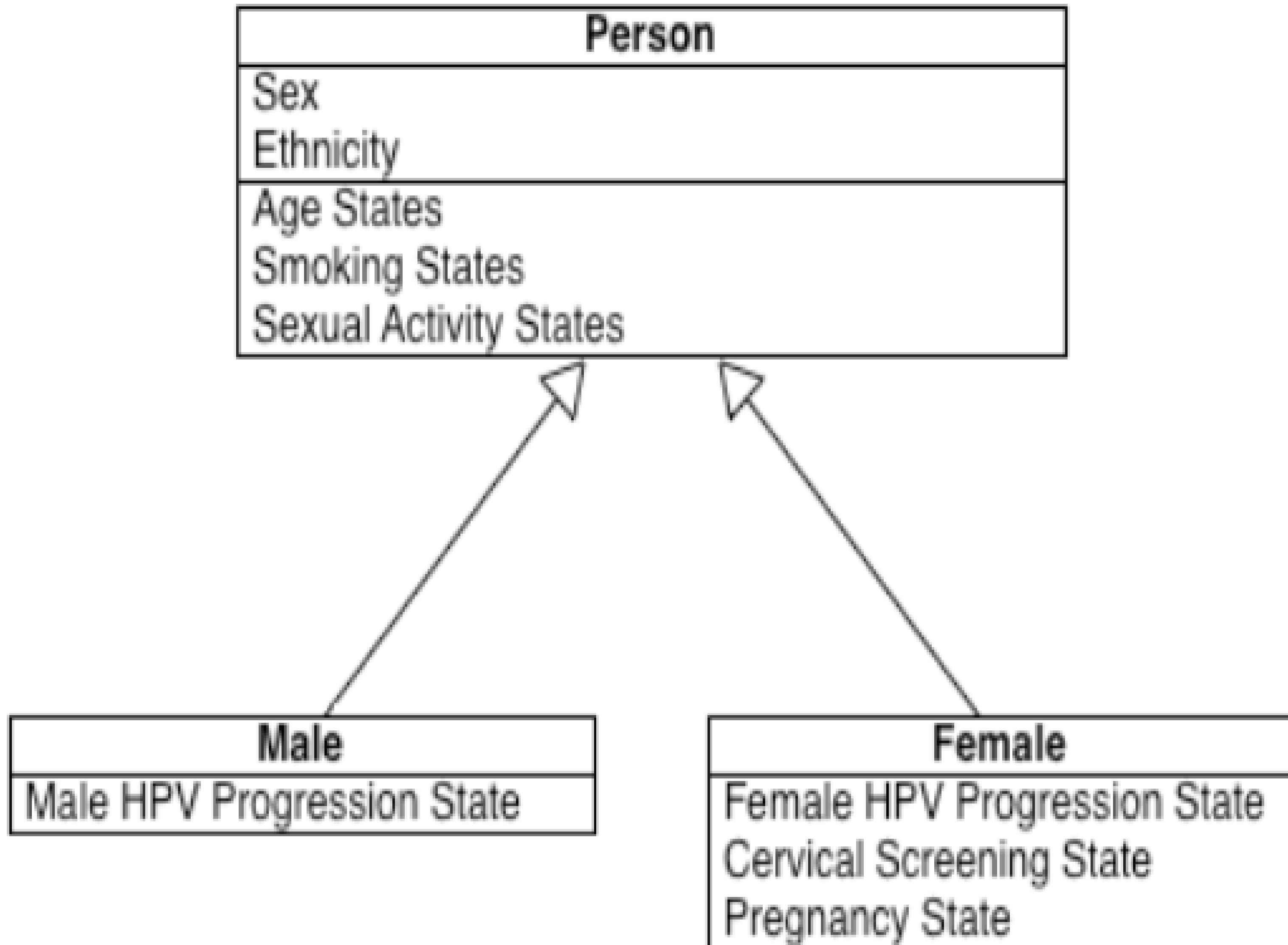
Person

Implements (comma-separated list of interfaces):

Male is (also) a subclass of **Person**

Selection

Example Subtypes (HPV Model)



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* subtypes
- 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 incompetible 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."

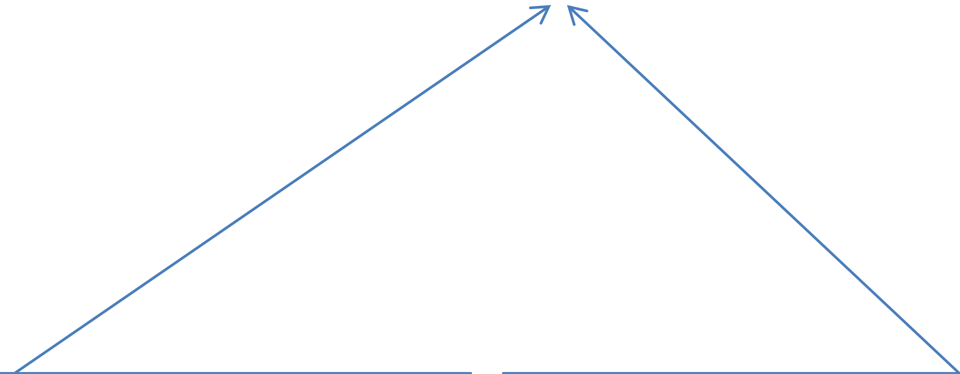
Persistent Metaphor: Service Contracts

- 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

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

Person | Main | AgentEntity.java | Person | Person

DaysPerTimeUnit
MeanDaysToNaturallyClearInfection
ReactivationRateForNormoGlycemicPeople
ReactivationRateForSmokingStatusAndCKDStage
ReactivationRateCoefficientForSmokingStatus
ReactivationRateHazardForNeverSmoker
ReactivationRateHazardForCurrentSmoker
RapidnessOfDecreaseInReactivationRateWithTimeSinceQuit
AgeCoefficientForSmokingInitiation
SmokingInitiationHazardLogisticSteepnessCoefficient

Tobacco Use

Properties | Console

AgeCoefficientForSmokingInitiation - Table Function

General
Description

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

Remove
Paste from Clipboard