Tutorial:
Classes, Objects & References

Nathaniel Osgood

CMPT 858

2-15-2011

Recall: 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

Recall: The Challenges of Complexity

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

Recall: 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

Recall: 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]

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

Recall: 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 anther)
  - Return some representation (e.g. a string) of or information about a person in the model

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)

Fecall: A Critical Distinction: Design (Specification) vs. Execution (Run) times

- The computational elements of Anylogic support both design & execution time presence & behaviour
  - Design time: Specifying the model
  - Execution time ("Runtime"): Simulating the model
- It is important to be clear on what behavior & information is associated with which times
- Generally speaking, design-time elements (e.g. in the palettes) are created to support certain runtime behaviors

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

“Methods” to Call on (or from within, using “this”) an Agent

- a.getConnectionsNumber() returns number of connections between this agent and others
- a.get_Main() gets reference to Main object
- a.toString() gets string rendition of agent
- a.getConnections() gets a collection (linked) list of agents to which this agent is connected (& over which we can iterate)
- a.connectTo(Agent b) connects a to b
- a.disconnectFrom(Agent b) disconnects b from a
- a.disconnectFromAll() disconnects all agents from a
- a.getConnectedAgent(int i) gets the ith agent connected to a
- a.isConnectedTo(Agent b) indicates if a is connected to b

Composition of Methods

- Suppose we have an agent called a
- a.getConnectedAgent(2).toString()
  - This will print out the “name” of the 3rd agent to which a is connected
- a.getConnectedAgent(0).getConnectionsNumber()
  - This will print out the number of connections possessed by the 1st agent to which a is connected

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”

Java Variables include…

- “Parameters” (“Arguments”) to functions
- Local variables within a function
- Fields within a class

Values & References

- In Java, variables hold values
  - It is the contents of these variables that is of interest – variables themselves just store values
- There are many types of variables could be
  - Parameters to a function
  - “Local” (temporary) variables within a function
  - Variables within a class (to be found in every object that is “instantiated” from that class
  - “Static” variables associated with a class (only one variable associated with the class – no how many objects of the class are circulating)

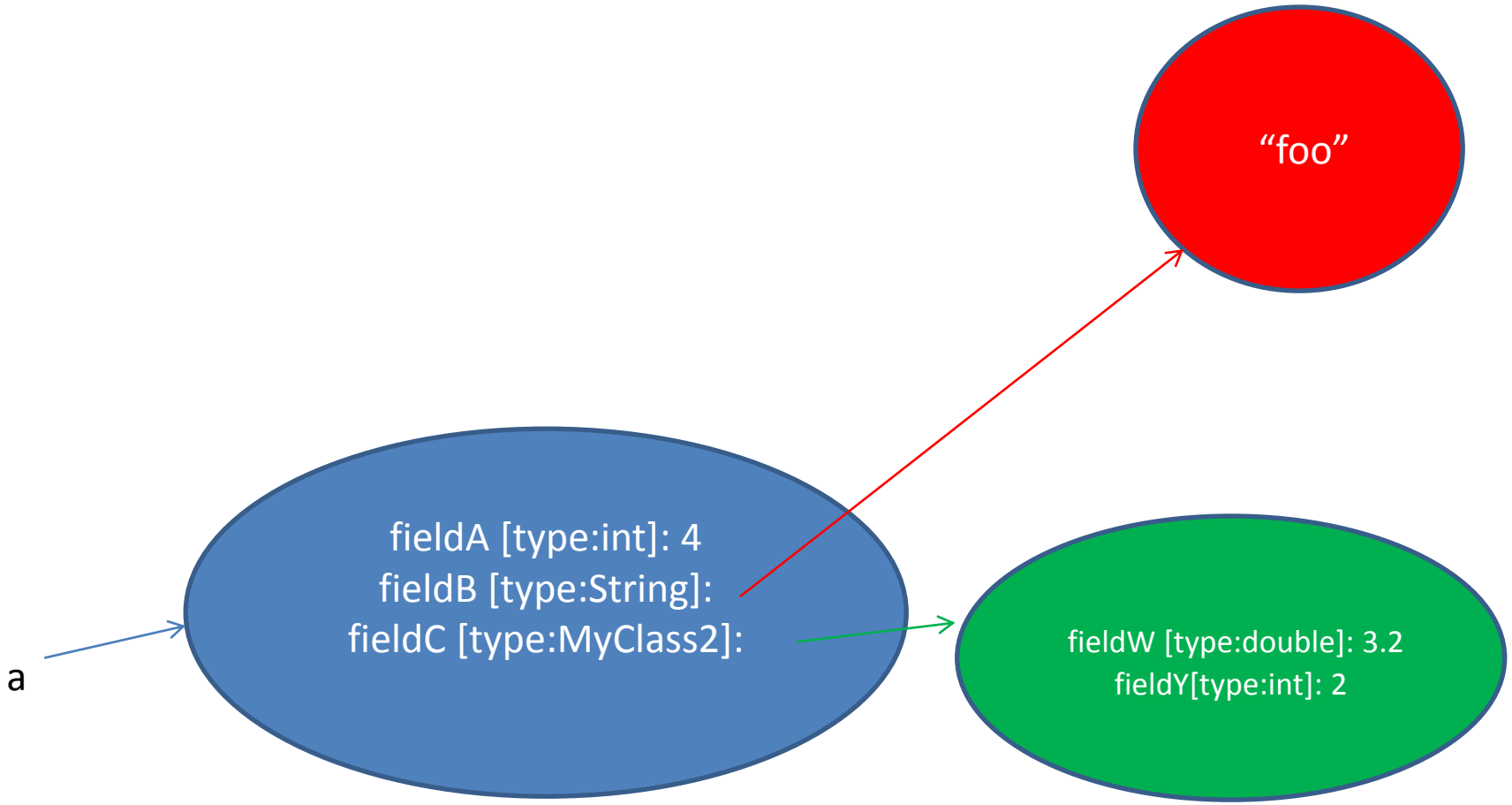
Broad Types of Java Values

- Primitive values
  - Here, the value is directly stored in the variable
    - int, double, float, etc.
- References
  - Here, the value within the variable actually points to either
    - An object (could have many other references to it as well!)
    - A distinguished value “null” (means “doesn’t refer to any object”)

Objects in Java

- Contain
  - Data: “Fields”, “Property”
    - These store information
  - Behavior: “Methods”/”Functions”
    - These allow the object to undertake certain tasks

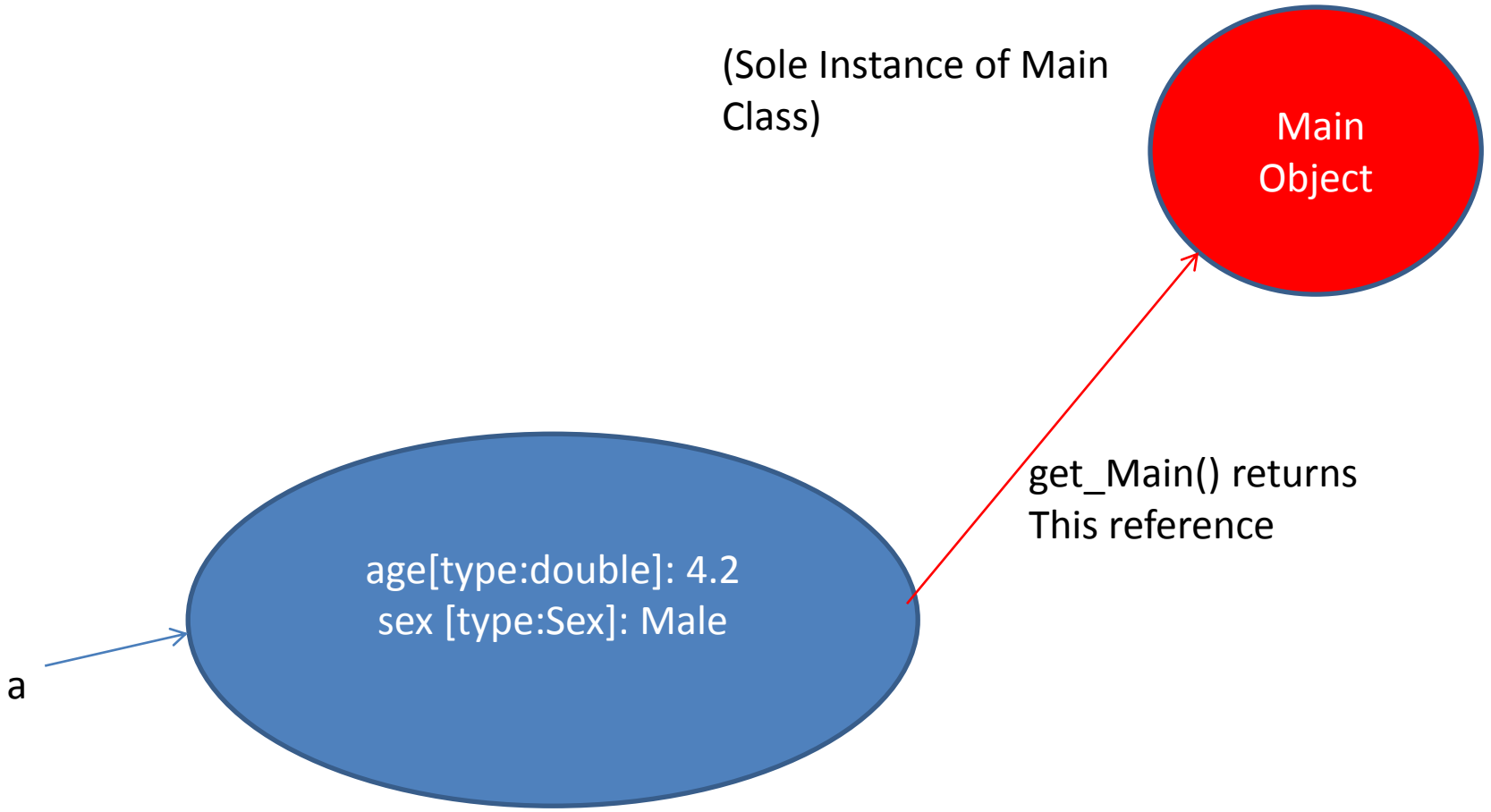
Object can contain References to Other Objects



Finding the Enclosing “Main” class from an Embedded Agent

- From within an embedded Agent, one can find the enclosing “Main” class by calling get_Main()
  - This will give a reference to the single instance (object) of the Main class in which the agent is embedded
  - An alternative approach is to call ((Main) getOwner)

Reference from Agent Class to Main Object

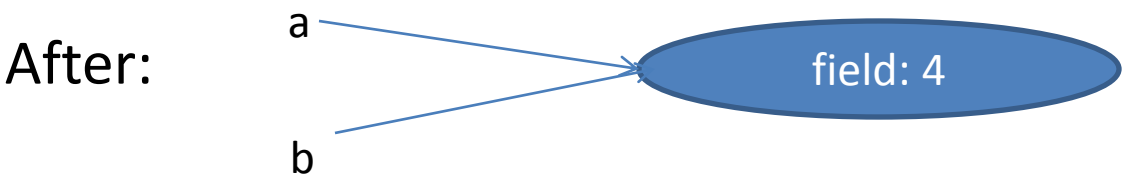


Assignment

- Consider two variables *a* and *b* that hold values
- Consider further the statement *a*=*b*
- How this is interpreted depends on the "type" of *b*
  - If *b* is a "primitive" (e.g. int, double): Here, the assignment will make a copy of that value

    Before: a: 2, b: 4

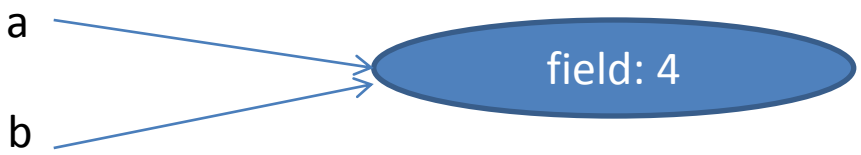
    After: a:4, b:4
  - If *b* holds a reference to an object, a will now hold a reference to that same object



Assignment

– If the programmer later modifies that object through a that same change will be visible through b as well

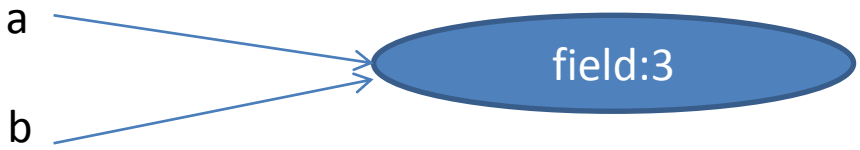
- Before



- Assignment to a “field” (property”) of the object through variable a

  a.field=3

- After



References Vs. Values

- The “type” of a variable indicates the sort of data to which it can refer
- Looking at a variable’s type will tell you much about how it can be used
  - Whether primitive or reference
  - Sort of operations that are possible on the data it holds

Arrays

- Java supports collections called “Arrays”
  - These store collections of values in an “indexed” fashion
    - By giving an “index”, we can get back an element
- These arrays can be of 1 or more “dimensions”
  - An array of dimension 2 is just a (1D) array of references to (1D) arrays

Example: Landscape Information