Mathematica models

Each Variable in the **program has a type** ("program type")

e.g. int, double, ...

Each program type has a **mathmatical type**. that **model** it: any variable of that program type as having a value from its mathematical model's mathematical space/domain:

e.g. integer, real, whole number

Program type	Mathematical type
boolean	boolean
char	character
int	Integer (-2147483648 through 2147483647)
double	real (about ±10±308, 15 significant digits)
String	string of character
NaturalNumber	Integer (non-negative)
Queue<T>	string of T
Stack<T>	string of T
Sequence<T>	stirng of T
Set<T>	finite set of T
Map<K,V>	finite set of (K,V) (with function property)
BinaryTree	binary tree of T
SortingMachine	ordered triple (a.k.a. three-tuple): a boolean, a binary relation on T, and a finite multiset of T.

***Vacuously True: p => q where p= false q = false, then p => q is true.

string of T : <t1, t2, t3>

Abstract Classes

introduce of abstract classes

The Secondary abstract class is an incomplete class that covers some redundancy methods we don't have to write again in our subclass.

e.g. the add method between NaturalNumber and NaturalNumber, we don't have to know what's the **represent** inner the NaturalNumber's child method, because the add method is **using the implemented method from the child method to finish add operation**

Object

Every class in Java extends Object, which is a special built-in class that provides default implementations for the following instance methods (among a few others that are not so important):

- boolean equals(Object obj)
  - The default implementation in **Object checks reference equality**, though we expect object value equality! So, we (almost always) need to override this method.
- int hashCode()
  - The default implementation in Object **returns an int** that **depends** on the **reference value of this**, though we expect it to depend on the object value! So, we (almost always) need to override this method.
- String toString()
  - The default implementation in Object **returns a String** that shows the **reference value of this,** though we expect it to show the object value! So, we (almost always) need to override this method

Java permits you to write a kind of “partial” or “**incomplete**” class that **contains bodies** for **some but** (typically) not all of the methods of the interfaces it claims to implement

Because some methods still **might not have bodies**, Java **WILL NOT** let you **instantiate** an abstract class; that is, you **cannot** use an abstract class like a normal class and **create a new object** from it.

Every **class(include normal class and abstract class)** to Object class's relationshipt is **implicit**, which means : every class that does not extend some other class directly extends Object.

e.g. NaturalNumberSecondary

This abstract class has code that overrides the default implementations (inherited from Object) of equals, hashCode, and toString for NaturalNumbers, so they do “the right thing”, i.e., **so their behaviors are based on object values rather than reference values.**

This abstract class also **has code that implements all the methods introduced in the NaturalNumber interface(add, derement, increment, etc.)** but not those inherited by it from other interfaces – Details of this code later… see “Resources”

– Note that this still **leaves the methods** introduced in the **Standard** and **NaturalNumberKernel** interfaces without bodies; hence, it’s an abstract class

Secondary abstract class leaves kernal classes with few standard class method

```
- Standard (i.e., newInstance, clear, and transferFrom)
- — NaturalNumberKernel (i.e., multiplyBy10, divideBy10, and isZero)
```

Why we need Abstract classes?

Factoring Out Common Code -> reduce redundancy code

Some Method bodies that can be written once and work for any implementation of NaturalNumberKernel because **they are programmed to that interface**—have been **factored out into an abstract class**

> Yes, NaturalNumber interface is extends the NaturalNumberKernal interface, we programmed everything based on NaturalNumberKernal. If NaturalNumberKernal methods work, then our NaturalNumber methods should works too.

> Keep "factored out into an abstract class" think that to move out all common factor code to abstract class

Therefore, This leaves only constructors and a few kernel methods to be implemented in NaturalNumber1L, NaturalNumber2, and future kernel classes (if any)

The code in each kernel class (e.g., in the example **NaturalNumber1L** and NaturalNumber2) implements only 4 constructors and 6 methods each, not all 21 methods of NaturalNumber

But, implementing these **few methods** is **different** than implementing the other 15 layered methods where you can call the **kernel methods(The methods we impelment from our kernel class)** to do the work!

JUnit Test Fixture Pattern

Problem

(UUT)Unit Under Test is not enough to test every kernel class.

Every time we trying to run test fixtrue for different kernel class, we have to modify all the test cases.

Solution

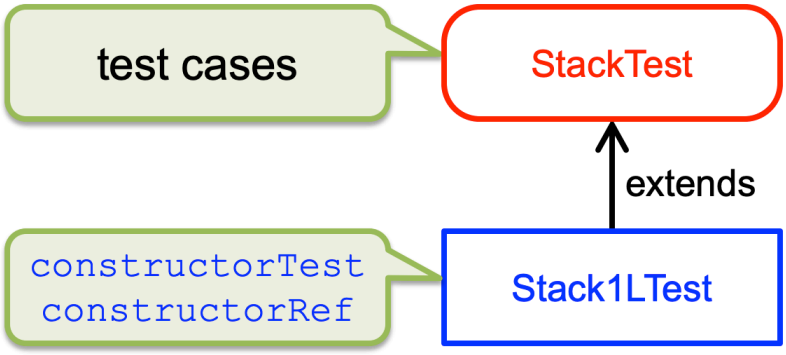
Single-Point control over change

“Re-route” all UUT constructor calls to **another method**, which **then calls the UUT constructor**, so only the body of that **one method needs to be changed** to accommodate a different UUT

Instead of calling the UUT’s constructor directly here, you call a method (perhaps named constructorTest), which then calls the UUT’s constructor.

There is constructorTest and constructorRef, if there is reference implementation of the same interface

Isolating This Change Point



makes the test cases class become abstract class, then we could implement our constructorTest and constructorRef method in the child Class. We could test different initial constructor by create different child class of test cases class.

```
public abstract class StackTest {

  protected abstract Stack<String> constructorTest();

  protected abstract Stack<String> constructorRef();

  ...

}
```

protected and Abstract method keyword

protecteed means that this method may be called or overridden in a subclass (which is our intent for Stack1LTest), but may not even be called from any other class declared outside this package. It could called the method in the same package

abstract in the method means this method **must** be **overridden** in some subclass.

Why we use protected and abstract here?

我们需要在当前testcases class里面调用这个method, 但是这个method还没有被重写, 所以我们需要保证constructorTest and constructorRef在subclass里面被重写. Protected保证了当前method能被当前包所有类内访问

How about we want to start with some args in our test kernel class?

In a typical test case, it takes many lines of code just to construct the values on which to run a test!
For instance, suppose you want to test a method with this Stack value as input: <"a", "stack", "with", "stuff">

So! we need a Convenience Method (or helper method) to help us using constructorTest and constructorRef return a instance and add some input to the instance

varargs

e.g. Object... String... int... allow programmer to cal this method with zero or more arguments of type String.

Kernel Implements I (kernel method?)

Interpretation of Representation

e.g. : For QueueKernel, one idea is to represent a Queue variable's value by using a java.util.List variable

By convention (of the OSU CSE components), a kernel class that directly represents the new type using a component from the Java libraries that is very similar, has a name ending in "L"

– In this case, it is called Queue1L

For Queue

What existing components (including builtin types of Java, and the Java libraries) could you build it on top of?

– In other words, what could you use as a **data representation** that could be **interpreted as a Queue value?**

Version Control Using Subversion

In team project, best practice is using version control system

this is not limit to code, a version control system can handle non-code file as well

Key idea : the Repository

A **central repository(Or remote repository, e.g. github is remote repository)** keeps all files (in our case, Java code) and a history of all modifications to them

- A new team member can **check out** their own private copy from the repository
- **Each member** can **update** their own copy to reflect the latest changes in the repository
    - e.g pull from remote repository
- **Each member** can **commit** changes from their own private copy to the repository

How work gets done?

Repository holds **master copy** of all files

– Never edited directly

– Stores complete history, too!

Each team member has a **local copy** (or **working copy**) in their own workspace

– All file creation, editing, deletion occurs here

Update and commit commands are used to **synchronize** local and master copies

The Optimistic Model

Any team member can modify their local copy of any file at any time

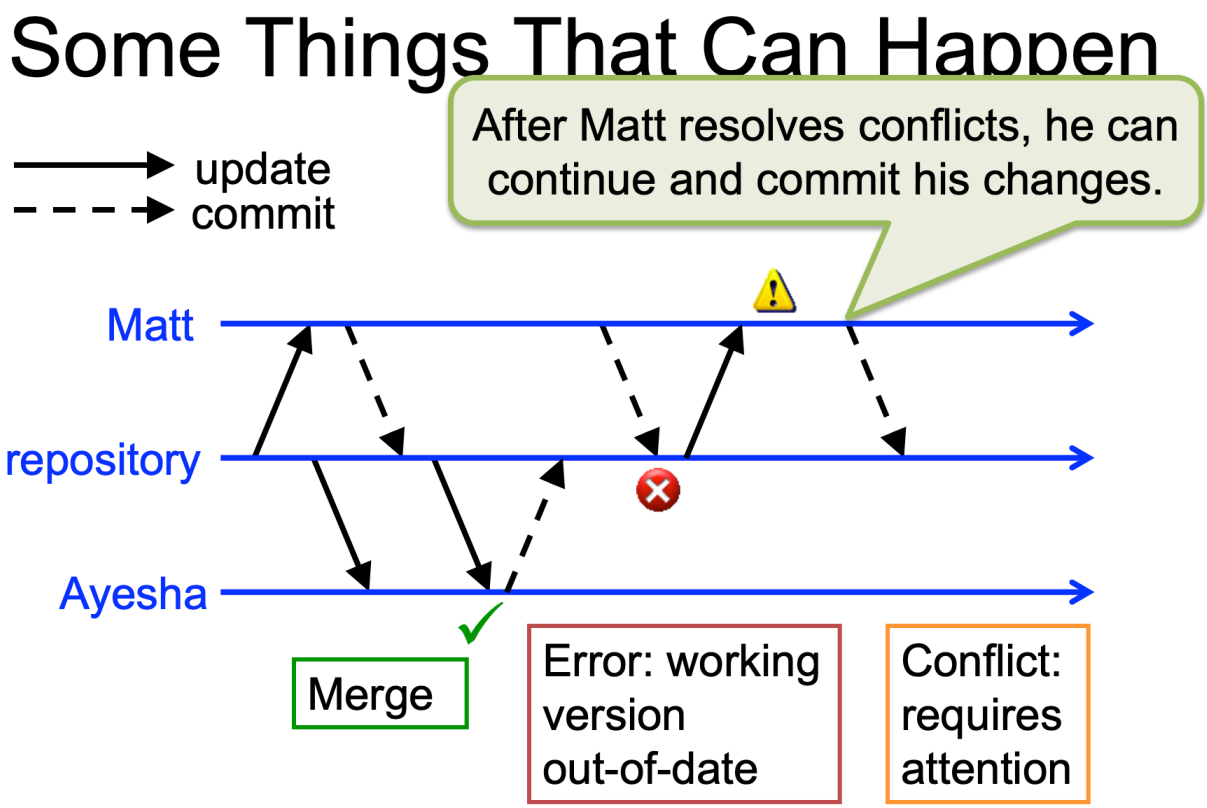
– No “locking” or other synchronization among team members takes place on local copies

note: 本地的修改不回影响到队员之间的同步, 及 其他人并不需要等待某一个人完成修改

On an **update**, the latest version from the repository often can be **merged** automatically into the local copy

– This is especially so when team members edit different files, so conflicts are rare

note: merge the file



check the progress of update and commit on the slide [version control]

Kernel Implementations II

Design Trade-offs

As the implementer of a kernel class, it is entirely **up to you how to represent every value allowed** by **the mathematical model** in the **kernel interface**!

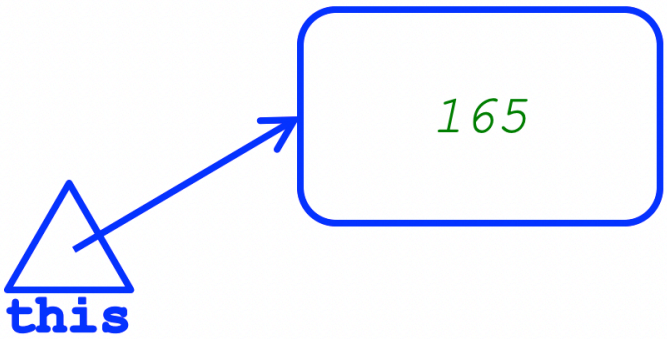
Some designs will make the code for the methods you need to write:

– Easier to understand and make correct

– More efficient

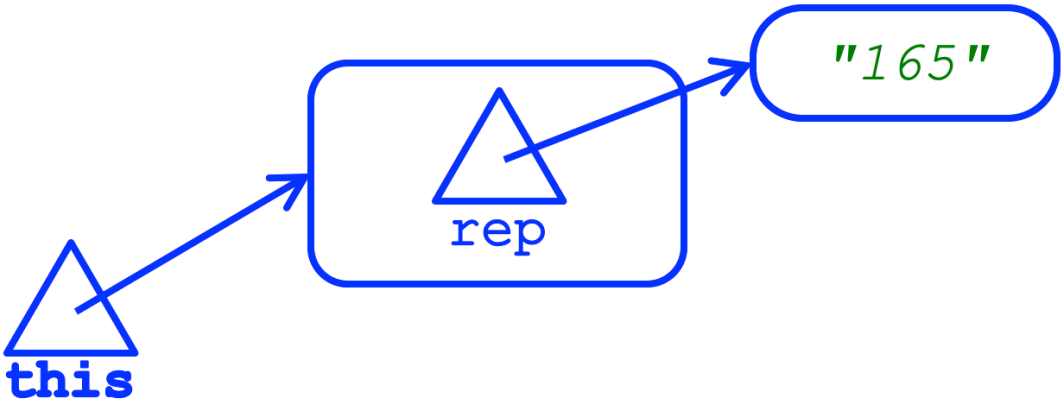
Client View

A situation as seen by a client of the kernel class `NaturalNumberAsString`, based on its interface:



Implementer View

The same situation as seen by the kernel class implementer, when the data representation is a single `String` called `rep`:



The implementer see different than the client.

Think about NaturalNumber

Represent It Using a String (above view)?

Represent It Using a Array?

Represent It Using a string? (mathematical model)

NaturalNumber as a string of integer whose entries are the digits of that number

e.g. <1,2,3> as NN 123

Note: that here we are thinking of a possible mathematical model of the representation, **not in terms of a specific Java component** family

The advantage to think this way -> there are serveral component with this mathematical model

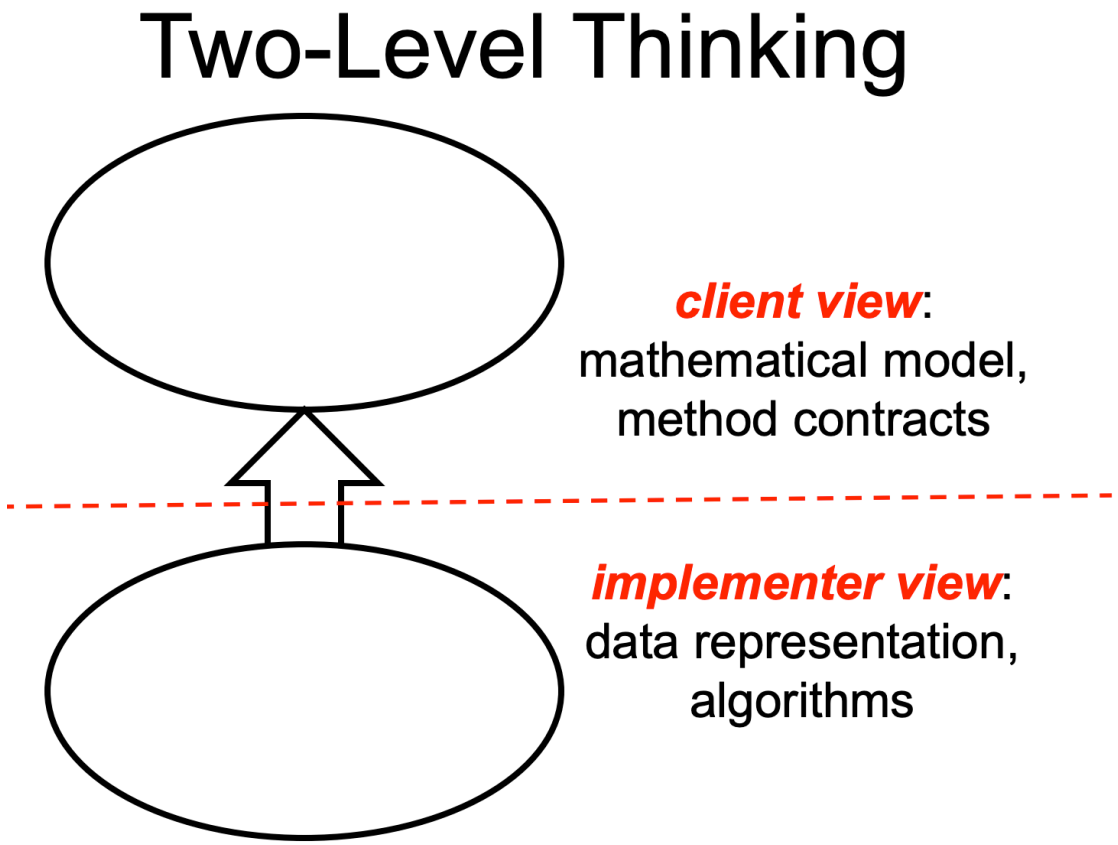
Sequence On Two Stacks ***

Have to look at HW and PPT

Kernel Implementations III

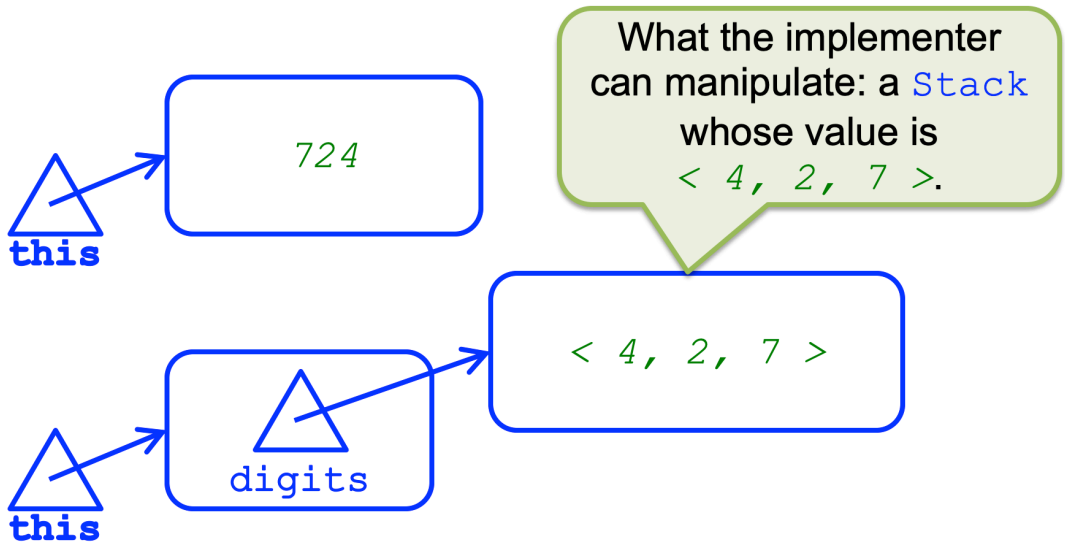
Two major question related to kernel class(mention: kernel class is the subclass of secondary that we only focus how to implements kernel interface method and standard interface method, other than that, anything from normal interface under the kernel interface is all base on the kernel interface API(methods))

1. What **data representation** (a.k.a. data structure) should be used to represent a value of the new type being implemented?
    1. Two level thinking
        1. One is the level for which you are implementing a new kernel class • See the kernel interface for the new type you are creating
            1. tower of abstractions for,
            2. what's the data structure we are using to represent current class?
        2. The other is the level directly below the level for the new kernel class you are creating • See the interfaces for the types of the variables you are using to represent a value of the new type\

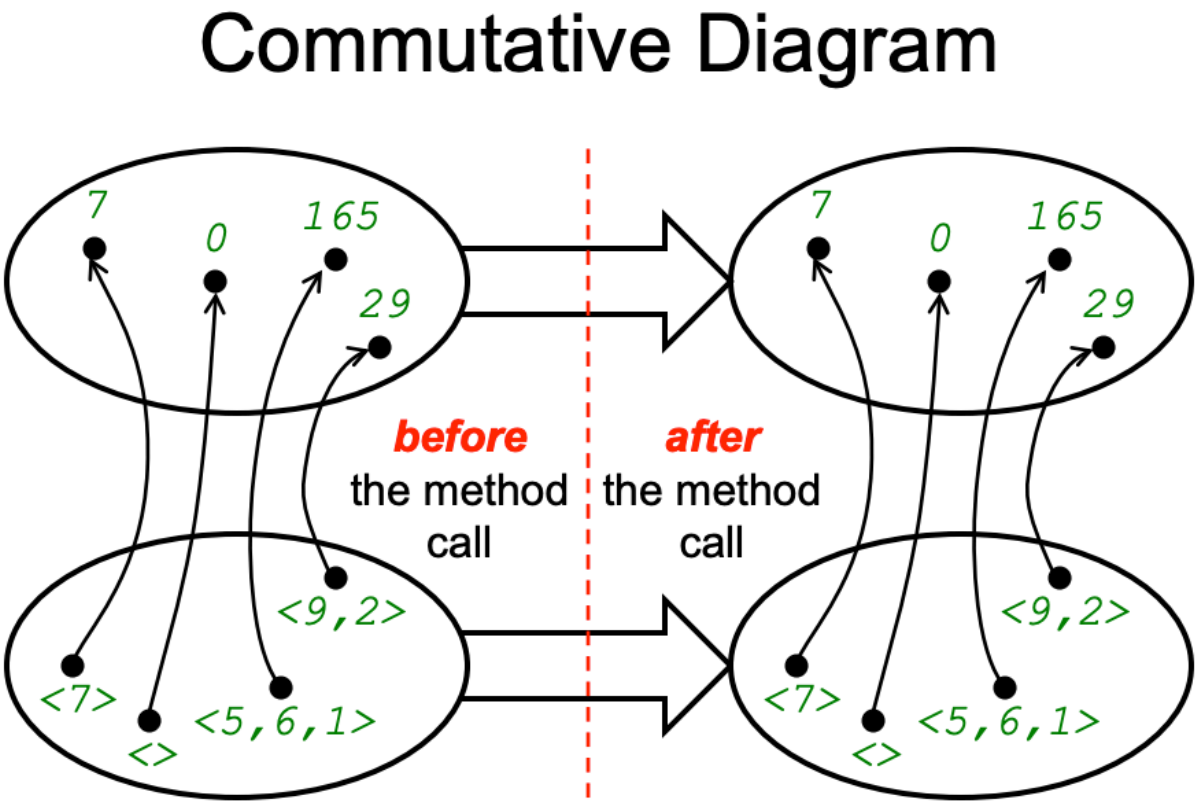


        1. 1. client view: See the kernel interface for a description of what the software behaves like, what the client sees.
                1. This is **abstract state space**: the set of all possible **math model** values as **seen by a client**.
            2. implementer view: See the kernel class for a description of how the software achieves its behavior.
                1. This is the **concrete state space**: the set of all possible math model values of the **data representation**.
        2.

Picture: `NaturalNumber2` on `Stack`

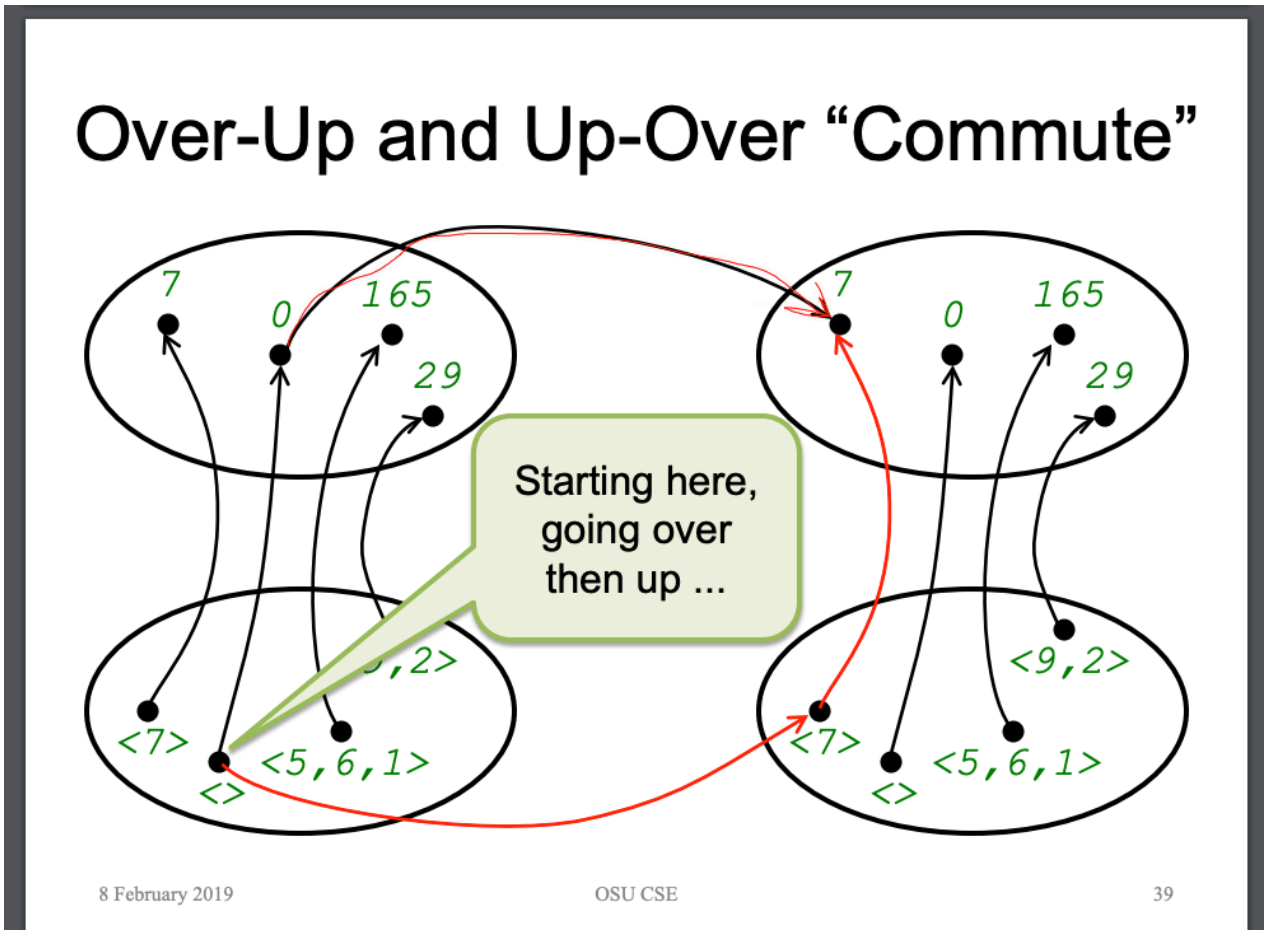


2. What **algorithms** should be used to manipulate that data representation to implement the contracts of the kernel methods?
    1. kernel 3 Restrict your attention to the **states** just before and just after a method call
    2. which leads to a device called a **commutative diagram**



3. 1. top arrow: **abstract transition**: for each state **before** the call, where it might **end up** according to the **method's contract**.
    2. bottom arrow: This is the **concrete transition**: for each state before the call, where it might end up according to the **method's body**.

4.



5. Correctness
    1. The **kernel class correctly implements the kernel interface** if and only if, for **every method** and for every legal input state for that method, the method body (over-thenup) **always results in a state that satisfies the method contract** (up-then-over)

Kernel Implementations IV

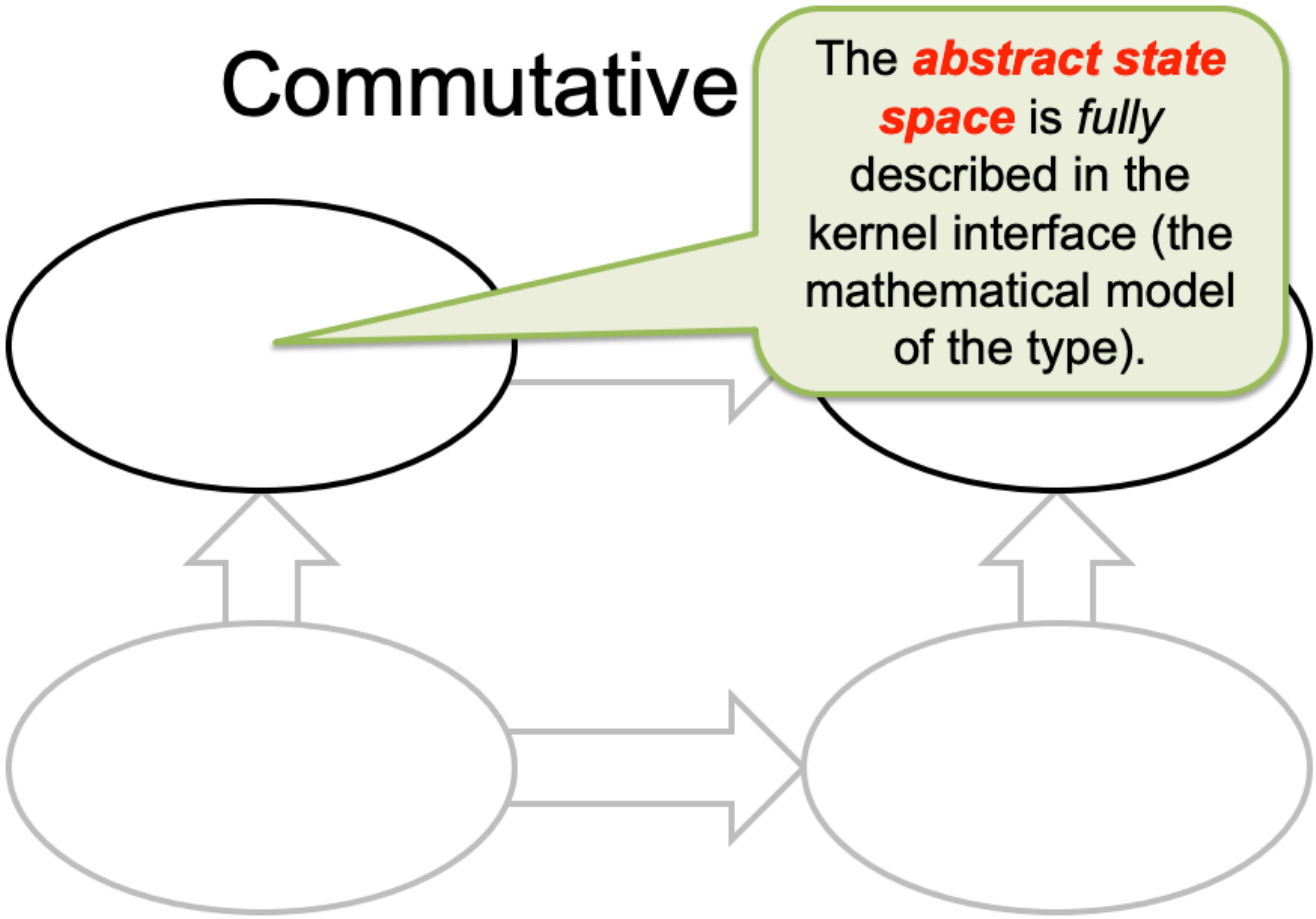
However, it is also **important to record** (document) the **key design decisions** illustrated in a commutative diagram, if they are not already recorded in the Java code itself

wo key design decisions that need to be recorded in (Javadoc) comments:

The **representation invariant**: Which “configurations” of values of the instance variables can ever arise?

The **abstraction function**: How are the values of the instance variables to be **interpreted** to get an **abstract value**?

Commutative Diagram



Example: Abstract

… a mathematical *integer* …

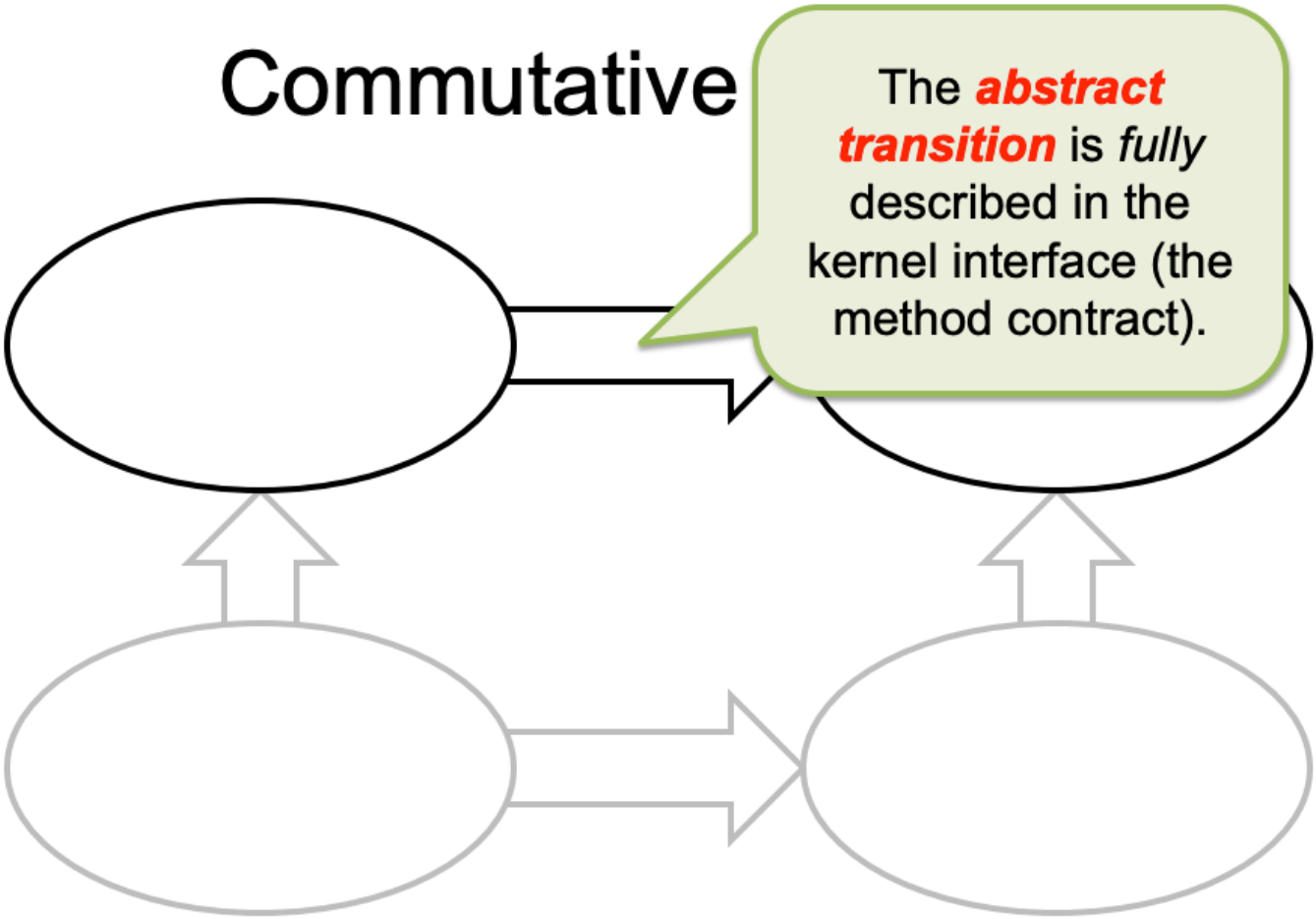
Consider NaturalNum
where we find this in

Mathematical Subtypes:

```
NATURAL is integer
   exemplar n
   constraint n >= 0
```

Mathematical Model (abstract value and abstract invariant of this):

```
type NaturalNumberKernel is modeled by
    NATURAL
```



Example: Abstract Transition

- Consider `multiplyBy10`, where we find this in the API:

  Updates:

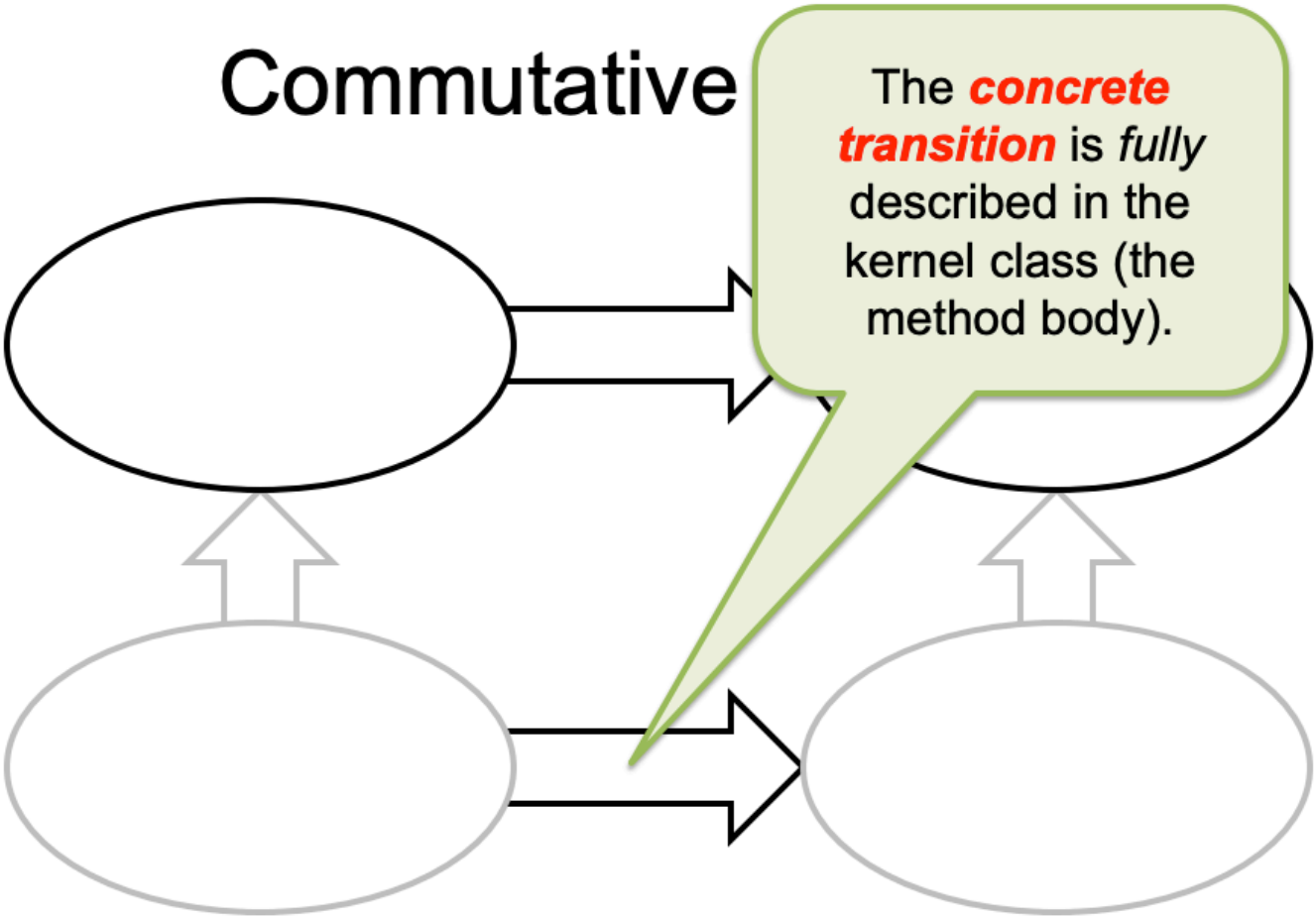
  this

  Requires:

  0 <= k < 10 Arrow Start

  Ensures:

  ***this* = 10 * #*this* + k** Arrow end



Like this one

```
private Stack<Integer> digits;
```

The **representation invariant characterizes the values that the data representation** (instance variables) **might have at the end of each kernel method body, including the constructor(s)**

The representation invariant is made to hold by the method bodies' code, and it is recorded in the convention clause in a (Javadoc) comment for the kernel class

Variable Life Cycle: Client

- Declared
- Initialized
- Called
- goes out of scope (destory)

representation invariant

@convention tag for representation Invariant

$this is special notation to name the data representation of this in such comment

Summarize

- The constructor(s) must make the representation invariant true
- The representation invariant may be **assumed to be true at the beginning** of each method body
- Each method body must make the representation invariant true (again) a**t the time it returns**

The Abstraction Function: The abstraction function describes how to interpret any concrete value (that satisfies the representation invariant) as an abstract value

帮忙解释data rep to abstract value 的function

The abstraction function is not computed by any code, but is merely recorded in the correspondence clause in a (Javadoc) comment for the kernel class

Kernel Purity Rule

Kernel Purity Rule — **No constructor or method body in the kernel class** should c**all any public method** from the **same component** family

Every public method in the component family relies (for its correctness) on the representation invariant being satisfied when it is called, and this might not be true when a call is made from inside a method of the kernel class

Implications of the kernel purity rule:

– No public kernel method should call any other public kernel method from the same class

– No public kernel method should call itself recursively

– No method (public or private) in the kernel class should call any layered/secondary method from the same component family

Hashing

Before we all using linear algorithm to search add or remove

Instead of searching through all the items, store the items in many smaller buckets and search through only one bucket that

How To Identify The Bucket

h(x) mod m this method giving a index of some bucket

The function that maps each value of type T to an integer is called the **hash function** we guaranteed to get the index of some bucket

hashCode Method

the type **Object** defines this instance method to **compute h**, i.e., as the programmatic version of **a hash function**: public int hashCode()

Note: As a best practice, nearly every type should override the default implementation of this method

hashCode always returns the same int hash value for the same Object Class (PhoneNumber class) value

An Empirical Matter

• How well hashing distributes the data among buckets depends, in part, on the data themselves

• Your worst enemy, knowing your hash function, could always provide data that would result in no performance gain over linear search

– Everything might fall into one bucket…

Binary Tree

(❌Haven't solve)What's the Binary Tree's Mathematical value?

binary Tree! Of course BinaryTree's Mathematical model is binary tree!

Traversal Orders

- Pre-order: root is visited before left and right
- In-order: root is visited between left and right (this order iterator in BinaryTree)
- Post-order: root is visited after left and right

Binary Search Tree (BST)

BST are general

BSTs may be used to search for items of any type T for which **one has defined a total preorder**, i.e., a **binary relation** on T that is **total**, **reflexive**, and **transitive**

- A **binary relation** on T may be viewed as a set of ordered pairs of T, or as a **boolean-valued** function **R** of two parameters of type T that is true iff that pair is in the set
- total : R(x, y) or R(y, x)
- Reflexive R(x, x)
- Transitive: if R(a, b) and R(b, c) then R(a, c)

e.g. kind of example

1. T = Integer
2. The ordering is <=
3. For simplicity we assume that no two nodes in a BST have the same labels

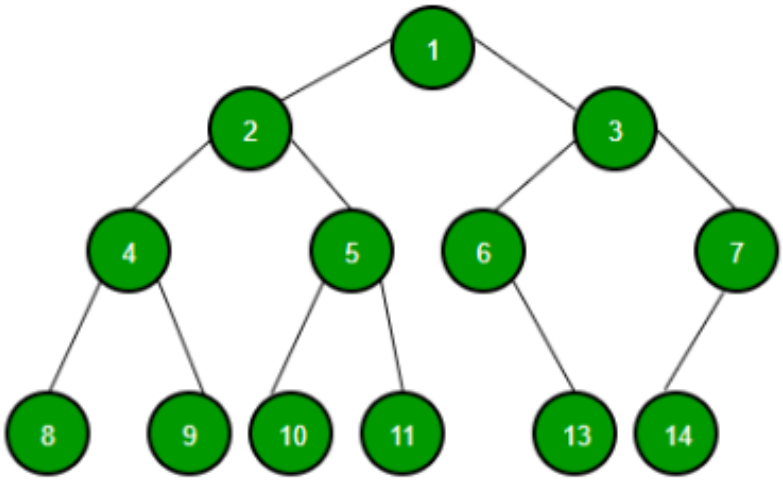
Note: Both these simplifications are inessential: BSTs are not limited to these situation

difference between normal binary tree and binary search tree

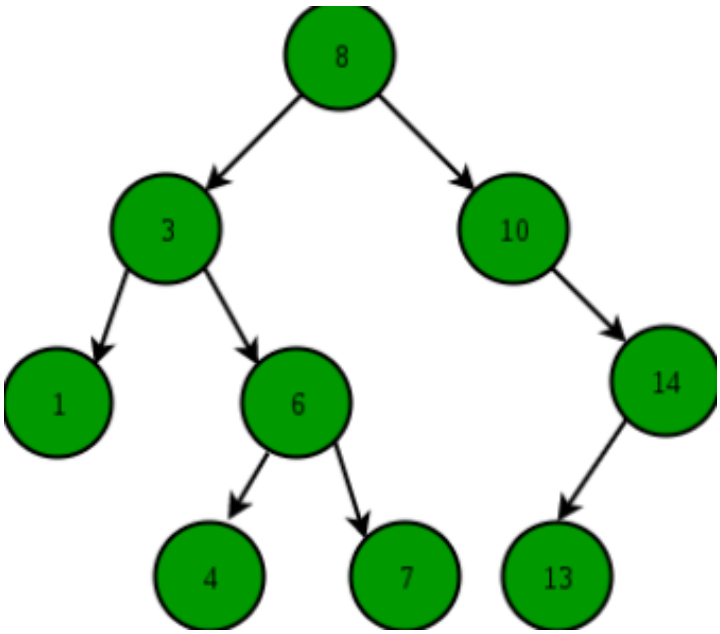
A binary tree is a BST whenever the arrangement of node labels satisfies these two properties:

1. For every node in the tree, if its label is x and if y is a label in that node's left subtree, then y < x
2. For every node in the tree, if its label is x and if y is a label in that node's right subtree, then y > x

Binary Tree:



Binary Search Tree:



SortingMachine (have to look)

The SortingMachine component family allows you to add elements of type T to a collection of such elements, and then to remove them one at a time in sorted order according to a client-supplied ordering

```
SORTING_MACHINE_MODEL is (
    insertion_mode: boolean,
    ordering: binary relation on T,
    contents: finite multiset of T
  )
  exemplar m
  constraint
    IS_TOTAL_PREORDER(m.ordering)
type SortingMachineKernel is modeled by
    SORTING_MACHINE_MODEL
```

Time based vs Values based

FIFO and LIFO are time-based orderings

osu.SortingMachine uses a value-based ordering.

The mathematical model is an ordered triple (a.k.a. three-tuple): a boolean, a binary relation on T, and a finite multiset of T.

***Recall: a binary relation on T may be viewed as a **set of ordered pairs of T**, or as a **boolean-valued function R of two parameters of type T** that is true iff that pair is in the set.

A **finite multiset** is essentially a **finite set** with **multiple copies** of elements allowed, so there are effectively (nonnegative) “counts” of all values of the element type T; details as necessary

reference link