Kernel Implementations III
Implementing a Kernel Class

• From the examples so far, you should see there are two major questions to address:
  – **Q1**: What *data representation* (a.k.a. *data structure*) should be used to represent a value of the new type being implemented?
  – **Q2**: What *algorithms* should be used to manipulate that data representation to implement the contracts of the kernel methods?
How To Think About Q1

• **Two-level thinking**: Restrict your attention to exactly two adjacent levels in the tower
  – One is the level for which you are implementing a new kernel class
    • See the kernel interface for the new type you are creating
  – 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
Example revisited: Queue represented using a Sequence.
The class **Queue3** uses the interface **Sequence**.
The two levels of the tower of abstractions for Queue3 are this level with Queue ...
Queue3 implements QueueKernel extends QueueSecondary extends Object extends Standard extends Sequence uses ... and this level with Sequence.
Example: `Queue3` on `Sequence`

```java
public class Queue3<T> extends QueueSecondary<T> {

    private Sequence<T> entries;

    ...
}
```
Example: `Queue3` on `Sequence`

```java
public class Queue3<T> extends QueueSecondary<T> {

    private Sequence<T> entries;

    ...
}
```

There is one *instance variable* (a.k.a. *data member*) in this representation of a `Queue`: a `Sequence` called `entries`. 
Picture: Queue3 on Sequence

What the client sees: a Queue whose value is < 4, 7, 3 >.
Picture: **Queue3 on Sequence**

What the implementer can manipulate: a **Sequence** whose value is `< 4, 7, 3 >`. 

```
< 4, 7, 3 >
```

```
< 4, 7, 3 >
```

```
< 4, 7, 3 >
```

```
entries
```

```
Example revisited: `NaturalNumber` represented using a `Stack`.
The class `NaturalNumber2` uses the interface `Stack`.
The two levels of the tower of abstractions for NN2 are this level with NN ...
... and this level with Stack.
Example: `NaturalNumber2` on Stack

```java
public class NaturalNumber2
    extends NaturalNumberSecondary {

    private Stack<Integer> digits;

    ...
}
```
There is one *instance variable* (a.k.a. *data member*) in this representation of a *NaturalNumber*: a *Stack*<Integer*> called *digits*. 
What the client sees: a `NaturalNumber` whose value is 724.
What the implementer can manipulate: a Stack whose value is $\langle 4, 2, 7 \rangle$. 

**Picture:** NaturalNumber2 on Stack
Two-Level Thinking

**client view:**
- mathematical model,
- method contracts

**implementer view:**
- data representation,
- algorithms
Two-Level Thinking

See the *kernel interface* for a description of what the software behaves like, what the client sees.

**client view:**
- mathematical model,
- method contracts

**implementer view:**
- data representation,
- algorithms
Two-Level Thinking

client view: mathematical model, method contracts

implementer view: data representation, algorithms

See the *kernel class* for a description of *how* the software achieves its behavior.
Two-Level Thinking

The ovals denote sets of *mathematical model* values, this one for the client’s view of the new type ...
Two-Level Thinking

client view: mathematical model, method contracts

implementer view: data structure, algorithms

... and this one for the kernel-class implementer’s view of the chosen data representation.
This is the *abstract state space*: the set of all possible math model values as seen by a client.
This is the **concrete state space**: the set of all possible math model values of the data representation.
Two-Level Thinking

This is the interpretation of each concrete value (below) as an abstract value (above).
Example: For interface `NaturalNumber` this is the set of all `integer` values that are non-negative.
Example: For class `NaturalNumber2` this is the set of all string of integer values that satisfy the properties we want in the data representation.
Example: For class `NaturalNumber2` the interpretation maps a string of integer to the integer whose decimal digits are those in the string, in reverse order.
Implementing a Kernel Class

• Recall, there are two major questions to address:
  
  – **Q1**: What *data representation* (a.k.a. *data structure*) should be used to represent a value of the new type being implemented?
  
  – **Q2**: What *algorithms* should be used to manipulate that data representation to implement the contracts of the kernel methods?
How To Think About Q2

• **One-step thinking**: Restrict your attention to the states *just before and just after* a method call

• Combined with two-level thinking about the data representation, this leads to a device called a *commutative diagram*
before the method call  after the method call
This is the **abstract transition**: for each state before the call, where it might end up according to the method’s **contract**.
This is the **concrete transition**: for each state before the call, where it might end up according to the method’s **body**.
Example: \( \text{n.multiplyBy10}(7) \)

The contract says that, if \( n = 0 \), then the call \( \text{n.multiplyBy10}(7) \); will result in \( n = 7 \).
Example: `n.multiplyBy10(7)`

So, given any data representation of $n = 0$, the method body must change it so that after the call it has some data representation of $n = 7$. 
Over-Up and Up-Over “Commute”
Over-Up and Up-Over “Commute”

Starting here, going over then up ...
Over-Up and Up-Over “Commute”

... is the same as going up then over.
Correctness

• 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-then-up) always results in a state that satisfies the method contract (up-then-over)

• The commutative diagram lets you *think clearly* about how to make your code correct
Some Questions To Think About

• Given a kernel class, can there ever be more than one data representation for a given abstract value? In other words, can there be more than one concrete value that can be interpreted as the same abstract value?
  – If so, what would it look like in the diagram?
Some Questions To Think About

• Given a kernel interface, can there ever be more than one possible correct (abstract) result of a call, according to a method’s contract?
  – If so, what would it look like in the diagram?
Some Questions To Think About

• Given a kernel class, can there ever be more than one possible correct (concrete) result of a call, according to a method’s body?
  – If so, what would it look like in the diagram?