NaturalNumber
NaturalNumber

• The **NaturalNumber** component family allows you to manipulate natural numbers (i.e., non-negative integers)
  – Unlike an **int** variable, a **NaturalNumber** variable has no upper bound on its value
  – On the other hand, you need to call methods to do arithmetic; there are no nice built-in operators (e.g., +, −, *, ==, <, …) or literals (e.g., 0, 1, 13, …) as with **int** variables
Interfaces and Classes

Standard

extends

NaturalNumber-Kernel

extends

NaturalNumber

implements

NaturalNumber1L

implements

NaturalNumber2
Interfaces and Classes

Standard

extends

Kernel

extends

NaturalNumber

implements

NaturalNumber

has contracts for three methods:

clear
clear

newInstance

newInstance

transferFrom

transferFrom

NaturalNumber1L

implements

NaturalNumber2

implements
Interfaces and Classes

NaturalNumberKernel

has contracts for three methods:
multiplyBy10
divideBy10
isZero

NaturalNumber

extends

Standard

implements

NaturalNumber1L

extends

NaturalNumberKernel

has contracts for three methods:
multiplyBy10
divideBy10
isZero

NaturalNumber2
Interfaces and Classes

NaturalNumber has contracts for 14 other methods, e.g., add, subtract etc.

NaturalNumber

Kernel

Standard

extends

NaturalNumber

implies

NaturalNumber1L

implies

NaturalNumber2
The **Standard** Interface

- The interface **Standard** has methods that are part of most (nearly all) OSU CSE component families
  - Separating the *standard methods* into their own interface means that these highly reused methods are *described in exactly one place*
The **Standard Interface**

- The interface *Standard* methods are part of most (nearly all) OSU CSE component families
  - Separating the *standard* methods into their own interface means that these highly reused methods are described in exactly one place

This design goal in software engineering is usually called *single point of control over change*. 
The Kernel Interface

• The interface \texttt{NaturalNumberKernel} has a minimal set of methods that are \textit{primitive} in the \texttt{NaturalNumber} component family
  – Separating these \textit{kernel (primary) methods} into their own interface identifies them as special in this regard
The Kernel Interface

• The interface `NaturalNumberKernel` has a minimal set of methods that are primitive in the `NaturalNumber` component family.
  – Separating these *kernel (primary) methods* into their own interface identifies them as special in this regard.

The choice of kernel methods is a key decision by the designer of a component family.
The Enhanced Interface

• The interface \texttt{NaturalNumber} has all other methods that are \textit{convenient to have} in the \texttt{NaturalNumber} component family
  
  – These \textit{secondary methods} are often more “powerful” than the kernel methods and are introduced to make the component family readily usable in typical client code
Mathematical Model

• The value of a `NaturalNumber` variable is modeled as a non-negative `integer`

• Formally:

```plaintext
NATURAL is integer
  exemplar  n
  constraint n >= 0
  type NaturalNumber is modeled by
    NATURAL
```
Mathematical Model

• The value of a NaturalNumber is modeled as a non-negative integer.

• Formally:

```plaintext
NATURAL is integer
exemplar n
constraint n >= 0
type NaturalNumber is modeled by
NATURAL
```

First, we define the mathematical model we intend to use, including any constraints that limit the values it might have.
Mathematical Model

• The value of a NaturalNumber variable is modeled as a non-negative integer.

• Formally:

\[
\text{NATURAL is integer}\quad \text{exemplar } \quad n \\
\text{constraint } \quad n \geq 0 \\
\text{type NaturalNumber is modeled by} \\
\text{NATURAL}
\]

Second, we state that a NaturalNumber variable has that mathematical model.
Constructors

• There are four *constructors* for each implementation class

• As always:
  – The name of the constructor is the name of the implementation class
  – Constructors differ only in their parameters
  – Each has its own contract (which is in the kernel interface `NaturalNumberKernel`)

9 January 2015 OSU CSE
No-argument Constructor

• A constructor with no parameters is called a *no-argument constructor*

• Ensures:

\[ \textit{this} = 0 \]
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NaturalNumber n = new NaturalNumber2();</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>NaturalNumber n = new NaturalNumber2();</code></td>
<td><code>n = 0</code></td>
</tr>
</tbody>
</table>
Copy Constructor

• There is a constructor with one parameter of the same type (NaturalNumber n), and it returns a copy of the parameter value so it is called a copy constructor.

• Ensures:
  
  \[ \textit{this} = n \]
**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>k = 12345678909</code></td>
<td></td>
</tr>
<tr>
<td><code>NaturalNumber m = new NaturalNumber2(k);</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 12345678909$</td>
<td>$k = 12345678909$</td>
</tr>
<tr>
<td><code>NaturalNumber m = new NaturalNumber2(k);</code></td>
<td>$m = 12345678909$</td>
</tr>
</tbody>
</table>
Constructor from \texttt{int}

- There is a constructor with one parameter \texttt{int i}
- Requires: \( i \geq 0 \)
- Ensures: \( \texttt{this} = i \)
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaturalNumber n = <code>new</code></td>
<td></td>
</tr>
<tr>
<td>NaturalNumber2(j);</td>
<td><code>j = 13</code></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$j = 13$</td>
</tr>
<tr>
<td>NaturalNumber n = new</td>
<td></td>
</tr>
<tr>
<td>NaturalNumber2(j);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$j = 13$</td>
</tr>
<tr>
<td></td>
<td>$n = 13$</td>
</tr>
</tbody>
</table>
Constructor from \texttt{String}

- There is a constructor with one parameter
  \texttt{String s}

- Requires:
  \texttt{there exists n: NATURAL}
  \[(s = \text{TO\_STRING}(n))\]

- Ensures:
  \[s = \text{TO\_STRING}(\text{this})\]
Constructor from **String**

- There is a constructor with one parameter
  
  ```
  String s
  ```

- Requires:
  
  ```
  there exists n: NATURAL
  (s = TO_STRING(n))
  ```

- Ensures:
  
  ```
  s = TO_STRING(this)
  ```

In other words, *s* must *look like* the result of converting some *NaturalNumber* value to a *String* ...
Constructor from `String`

- There is a constructor with one parameter:
  ```
  String s
  ```

- Requires:
  ```
  there exists n: NATURAL
  (s = TO_STRING(n))
  ```

- Ensures:
  ```
  s = TO_STRING(this)
  ```

... and the `NaturalNumber` value resulting from the constructor is what would have given you that `String`. 
**Example**

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = &quot;265&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>NaturalNumber n = new NaturalNumber2(s);</code></td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = &quot;265&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>NaturalNumber n = new NaturalNumber2(s);</code></td>
<td></td>
</tr>
<tr>
<td><code>s = &quot;265&quot;</code></td>
<td><code>n = 265</code></td>
</tr>
</tbody>
</table>
Methods for \texttt{NaturalNumber}

- All the methods for \texttt{NaturalNumber} are \textit{instance methods}, i.e., you call them as follows:
  \[ n . \texttt{methodName}(\texttt{arguments}) \]
  where \( n \) is an initialized variable of type \texttt{NaturalNumber}
Methods for **NaturalNumber**

- All the methods for **NaturalNumber** are *instance methods*, i.e., you call them as follows:

  ```
  n.methodName(arguments)
  ```

  where `n` is an initialized variable of type **NaturalNumber**.

Recall: `n` is called the *receiver*; for all instance methods, the corresponding *distinguished formal parameter* implicitly has the name *this*. 
Order of Presentation

• The methods are introduced here starting with those you might expect to see as a client, and then proceeding to ones that might seem more surprising

• Methods not discussed here:
  - `setFromInt`, `canConvertToInt`, `toInt`
  - `setFromString`, `canSetFromString`
  - `increment`, `decrement`
add

```c
void add(NaturalNumber n)

• Adds \( n \) to \textit{this}.
• Updates: \textit{this}
• Ensures:
  \[
  \textit{this} = \#\textit{this} + n
  \]
```
add

```c
void add(NaturalNumber n)
• Adds \( n \) to \( \text{this} \).
• Updates: \( \text{this} \)
• Ensures:
  \[
  \text{this} = \#\text{this}
  \]
```

The *parameter mode* called *updates* in a contract means the variable’s value *might be changed* by a call to the method.
void add(NaturalNumber n)

• Adds $n$ to $this$.
• Updates: $this$
• Ensures:
  $this = \#this$

If $this$ is an updates-mode parameter in any method, then the type in question is mutable.
void add (NaturalNumber n)

• Adds \textit{n} to \textit{this}.

• Updates: \texttt{this}

• Ensures: \texttt{this} = \#this + n

In an ensures clause, a \# in front of a variable whose value might be changed is pronounced “old”; \#this denotes the old, or incoming, value of \texttt{this}.
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = 143</code></td>
<td><code>m = 143</code></td>
</tr>
<tr>
<td><code>k = 70</code></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td><code>m.add(k);</code></td>
<td></td>
</tr>
</tbody>
</table>

```java
m.add(k);
```
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = 143</code></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td><code>m.add(k);</code></td>
<td></td>
</tr>
<tr>
<td><code>m = 213</code></td>
<td><code>k = 70</code></td>
</tr>
</tbody>
</table>
void subtract (NaturalNumber n)

• Subtracts $n$ from this.
• Updates: this
• Requires: $this \geq n$
• Ensures: $this = \#this - n$
void subtract (NaturalNumber n)

• Subtracts \( n \) from this.
• Updates: \( \text{this} \)
• Requires: \( \text{this} \geq n \)
• Ensures: \( \text{this} = \#\text{this} - n \)

Important! It could have been written as:
\[
\#\text{this} = \text{this} + n
\]
subtract

\texttt{void subtract(NaturalNumber n)}

- Subtracts \texttt{n} from \texttt{this}.
- Updates: \texttt{this}
- Requires: \texttt{this} \texttt{\geq} \texttt{n}
- Ensures: \texttt{this} = \#\texttt{this} \texttt{- n}

Or even as:
\texttt{this + n = \#this}
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td>k = 70</td>
</tr>
<tr>
<td>m.subtract(k);</td>
<td></td>
</tr>
</tbody>
</table>

In the code, `m = 143` and `k = 70` are defined, and then the operation `m.subtract(k);` is performed.
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.subtract(k);</code></td>
<td><code>m = 143</code></td>
</tr>
<tr>
<td></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td></td>
<td><code>m = 73</code></td>
</tr>
<tr>
<td></td>
<td><code>k = 70</code></td>
</tr>
</tbody>
</table>
multiply

\textbf{void} multiply(NaturalNumber n)

• Multiplies \textbf{this} by \textit{n}.
• Updates: \textbf{this}
• Ensures:
  \[ \textbf{this} = \#\textbf{this} \times n \]
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td></td>
</tr>
<tr>
<td>k = 70</td>
<td></td>
</tr>
<tr>
<td>m.multiply(k);</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m.multiply(k);}</td>
<td>$m = 143$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td></td>
<td>$m = 10010$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
</tbody>
</table>
divide

NaturalNumber divide(NaturalNumber n)

• Divides \textit{this} by \textit{n}, returning the remainder.
• Updates: \textit{this}
• Requires: 
  \[ n > 0 \]
• Ensures:
  \[
  \#this = n \times this + divide \quad \text{and} \quad 0 \leq divide < n
  \]
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td>m = 143</td>
</tr>
<tr>
<td>k = 70</td>
<td>k = 70</td>
</tr>
<tr>
<td>NaturalNumber r = m.divide(k);</td>
<td></td>
</tr>
</tbody>
</table>

```java
NaturalNumber r = m.divide(k);
```
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = 143</code></td>
<td><code>m = 2</code></td>
</tr>
<tr>
<td><code>k = 70</code></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td><code>r = m.divide(k);</code></td>
<td><code>r = 3</code></td>
</tr>
<tr>
<td><strong>NaturalNumber</strong></td>
<td></td>
</tr>
</tbody>
</table>
void power(int p)

• Raises this to the power p.
• Updates: this
• Requires:
  \[ p \geq 0 \]
• Ensures:
  \[ this = #this ^ (p) \]
void power(int p)
• Raises this to the power p.
• Updates: this
• Requires: p >= 0
• Ensures: this = #this ^ (p)

Note: 0 ^ (0) = 1 by definition of the ^ operator.
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| ```
m = 143
k = 4
m.power(k);
``` |       |
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( m = 143 ) &lt;br&gt; ( k = 4 )</td>
</tr>
<tr>
<td>( m \cdot \text{power}(k); )</td>
<td>( m = 418161601 ) &lt;br&gt; ( k = 4 )</td>
</tr>
</tbody>
</table>
void root(int r)

• Updates this to the $r$-th root of its incoming value.

• Updates: this

• Requires:

\[ r \geq 2 \]

• Ensures:

\[ this^r \leq this < (this + 1)^r \]
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td></td>
</tr>
<tr>
<td>k = 2</td>
<td></td>
</tr>
<tr>
<td>m.root(k);</td>
<td></td>
</tr>
</tbody>
</table>
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
|          | $m = 143$
|          | $k = 2$
| `m.root(k);` |        |
|          | $m = 11$
|          | $k = 2$ |
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 144</td>
<td>m = 144</td>
</tr>
<tr>
<td>k = 2</td>
<td>k = 2</td>
</tr>
<tr>
<td>m.root(k);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>m = 12</td>
</tr>
<tr>
<td></td>
<td>k = 2</td>
</tr>
</tbody>
</table>
void copyFrom(NaturalNumber n)

• Copies \( n \) to \texttt{this}.

• Replaces: \texttt{this}

• Ensures:

\begin{center}
\texttt{this} = n
\end{center}
copyFrom

```java
void copyFrom(NaturalNumber n)
• Copies n to this.
• Replaces: this
• Ensures:
  this = n
```

The parameter mode called replaces in a contract means the variable’s value might be changed by a call to the method, but the new value is independent of the old value.
copyFrom

```java
void copyFrom(NaturalNumber n)
```

- Copies `n` to `this`.
- Replaces: `this`
- Ensures:
  ```java
  this = n
  ```

If `this` is a replaces-mode parameter in any method, then the type in question is `mutable`. 
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.copyFrom(k);</code></td>
<td><code>m = 143</code></td>
</tr>
<tr>
<td></td>
<td><code>k = 70</code></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m = 143$</td>
<td>$m = 70$</td>
</tr>
<tr>
<td>$k = 70$</td>
<td>$k = 70$</td>
</tr>
<tr>
<td>$m.copyFrom(k);$</td>
<td></td>
</tr>
</tbody>
</table>
```java
int compareTo(NaturalNumber n)

• Compares n to this, returning a negative number if this < n, 0 if this = n, and a positive number if this > n

• Ensures: compareTo = [a negative number, zero, or a positive integer as this is less than, equal to, or greater than n]
```
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$m = 143$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td>\textbf{int} $\text{comp} =$</td>
<td></td>
</tr>
<tr>
<td>\texttt{m.compareTo(k);}</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
</table>
| \[\text{int } \text{comp} = \text{m.compareTo(k);}\] | \[m = 143 \]
| \[k = 70\] | \[k = 70\]
| \[\text{comp} = 1\] | \[\text{comp} = 1\] |
### Example

```java
int comp = m.compareTo(k);
```

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
</tr>
<tr>
<td>k = 70</td>
</tr>
<tr>
<td>comp = 1</td>
</tr>
</tbody>
</table>

Though here the result of the method is 1, it could be any positive int, so don’t assume it is 1.
**multiplyBy10**

```c
void multiplyBy10(int k)

• Multiplies this by 10 and adds k.
• Updates: this
• Requires: 
  \[ 0 \leq k < 10 \]
• Ensures: 
  \[ this = 10 \times \#this + k \]
```
multiplyBy10

```c
void multiplyBy10(int k)
```

- Multiplies `this` by 10 and adds
- Updates: `this`
- Requires: 
  \[ 0 \leq k < 10 \]
- Ensures: 
  \[ this = 10 \times \#this + k \]

This is a kernel method.
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m = 143$</td>
<td>$m = 143$</td>
</tr>
<tr>
<td>$d = 7$</td>
<td>$d = 7$</td>
</tr>
<tr>
<td><code>m.multiplyBy10(d);</code></td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = 143</code></td>
<td><code>m = 1437</code></td>
</tr>
<tr>
<td><code>d = 7</code></td>
<td><code>d = 7</code></td>
</tr>
<tr>
<td><code>m.multiplyBy10(d);</code></td>
<td></td>
</tr>
</tbody>
</table>
divideBy10

```cpp
int divideBy10() {
    // Divides this by 10 and returns the remainder.
    // Updates: this
    // Ensures:
    // #this = 10 * this + divideBy10 and
    // 0 <= divideBy10 < 10

    // Implementation
}
```
**divideBy10**

```c
int divideBy10() {
    // Divides this by 10 and returns the remainder.
    // Updates: this
    // Ensures:
    #this = 10 * this + divideBy10 and
    0 <= divideBy10 < 10
}
```

This is a kernel method.
### Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>m = 1437</td>
</tr>
<tr>
<td>int ( r = )</td>
<td></td>
</tr>
<tr>
<td>( m \text{.} \text{divideBy10()} );</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>int r =</td>
<td>m = 1437</td>
</tr>
<tr>
<td>m.divideBy10();</td>
<td></td>
</tr>
<tr>
<td>m = 143</td>
<td>r = 7</td>
</tr>
<tr>
<td>r = 7</td>
<td></td>
</tr>
</tbody>
</table>
isZero

boolean isZero()

• Reports whether this is zero.
• Ensures:

\[
\text{isZero} = (\text{this} = 0)
\]
isZero

boolean isZero()

• Reports whether this is zero.

• Ensures:

  \[ isZero = (this = 0) \]

This is a kernel method.
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean z = m.isZero();</td>
<td>m = 143</td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$m = 143$</td>
</tr>
</tbody>
</table>
| ```java
  boolean z =
  m.isZero();
``` | $m = 143$     |
|                                  | $z = false$  |
clear

void clear()

• Resets this to an initial value.
• Clears: this
• Ensures:
  
  \[ this = 0 \]


**clear**

```c
void clear()
```

- Resets `this` to an initial value.
- Clears: `this`
- Ensures:
  ```c
  this = 0
  ```

This is a standard method.
clear

```java
void clear()

• Resets this to an initial value.
• Clears: this
• Ensures:
  \[ \text{this} = 0 \]
```

The parameter mode called clears in a contract means the variable’s value is reset to an initial value by a call to the method.
clear

```java
void clear()

• Resets this to an initial value.
• Clears: this
• Ensures:
  this = 0
```

If this is a clears-mode parameter in any method, then the type in question is mutable.
void clear()

- Resets this to an initial value.
- Clears: this
- Ensures: 
  \[ \text{this} = 0 \]

The ensures clause is redundant in this case because this is a clears-mode parameter.
<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m = 143}</td>
<td></td>
</tr>
<tr>
<td>\texttt{m.clear();}</td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m = 143</code></td>
<td></td>
</tr>
<tr>
<td><code>m.clear();</code></td>
<td></td>
</tr>
<tr>
<td><code>m = 0</code></td>
<td></td>
</tr>
</tbody>
</table>
newInstance

NaturalNumber newInstance()

• Returns a new object with the same implementation as this, having an initial value.

• Ensures:

  newInstance = 0
newInstance

NaturalNumber newInstance() 

• Returns a new object with the same implementation as this, having an initial value.
• Ensures: 

  newInstance = 0

This is a standard method.
newInstance

NaturalNumber newInstance()  
• Returns a new object with the same implementation as this, having an initial value.  
• Ensures:  
  newInstance = 0

This is similar to a constructor; the difference is that you don’t need to know the name of any implementation class to call this method.
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m = 143$</td>
<td></td>
</tr>
<tr>
<td><code>NaturalNumber k = m.newInstance();</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td>m = 143</td>
</tr>
<tr>
<td>NaturalNumber k = m.newInstance();</td>
<td>k = 0</td>
</tr>
</tbody>
</table>
void transferFrom(NaturalNumber n)

- Sets this to the incoming value of n, and resets n to an initial value; n must be of the same implementation as this.
- Replaces: this
- Clears: n
- Ensures:
  
  \[ this = \#n \]
**transferFrom**

```java
void transferFrom(NaturalNumber n)
```

- Sets `this` to the incoming value of `n`, and resets `n` to an initial value; `n` must be of the same implementation as `this`.
- Replaces: `this`
- Clears: `n`
- Ensures:
  
    `this = #n`

This is a standard method.
transferFrom

```java
void transferFrom(NaturalNumber n)
```

- Sets `this` to the incoming value of `n`, and resets `n` to an initial value; `n` must be of the same implementation as `this`.
- Replaces: `this`
- Clears: `n`
- Ensures:
  ```
  this = #n
  ```

This is similar to `copyFrom` but is always more efficient, so it should be used if you don’t really need a duplicate.
# Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$m = 143$</td>
</tr>
<tr>
<td></td>
<td>$k = 70$</td>
</tr>
<tr>
<td><code>m.transferFrom(k);</code></td>
<td></td>
</tr>
</tbody>
</table>
## Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m.transferFrom(k);</code></td>
<td><code>m = 143</code></td>
</tr>
<tr>
<td></td>
<td><code>k = 70</code></td>
</tr>
<tr>
<td></td>
<td><code>m = 70</code></td>
</tr>
<tr>
<td></td>
<td><code>k = 0</code></td>
</tr>
</tbody>
</table>
Whoa! It Clears $n$?

• Did you notice that `transferFrom` changes the value of its argument? How can it do this? Didn’t we say that this can’t happen?
  – It can’t for arguments of Java’s primitive types

• There is a crucial difference between Java’s primitive types and all other types, that allows this behavior for other types
  – Details coming soon...
toString

String toString()

• Returns the string representation of this.
• Ensures:

\[\text{toString} = [\text{the string representation of this}]\]
Example

<table>
<thead>
<tr>
<th>Code</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>m = 143</td>
<td></td>
</tr>
<tr>
<td>String s = m.toString();</td>
<td></td>
</tr>
</tbody>
</table>
### Example

<table>
<thead>
<tr>
<th><strong>Code</strong></th>
<th><strong>State</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{m = 143}</td>
<td>\texttt{m = 143}</td>
</tr>
<tr>
<td>\texttt{String s = m.toString();}</td>
<td>\texttt{m = 143} \texttt{s = &quot;143&quot;}</td>
</tr>
</tbody>
</table>
Resources

• OSU CSE Components API:
  NaturalNumber
  – http://cse.osu.edu/software/common/doc/