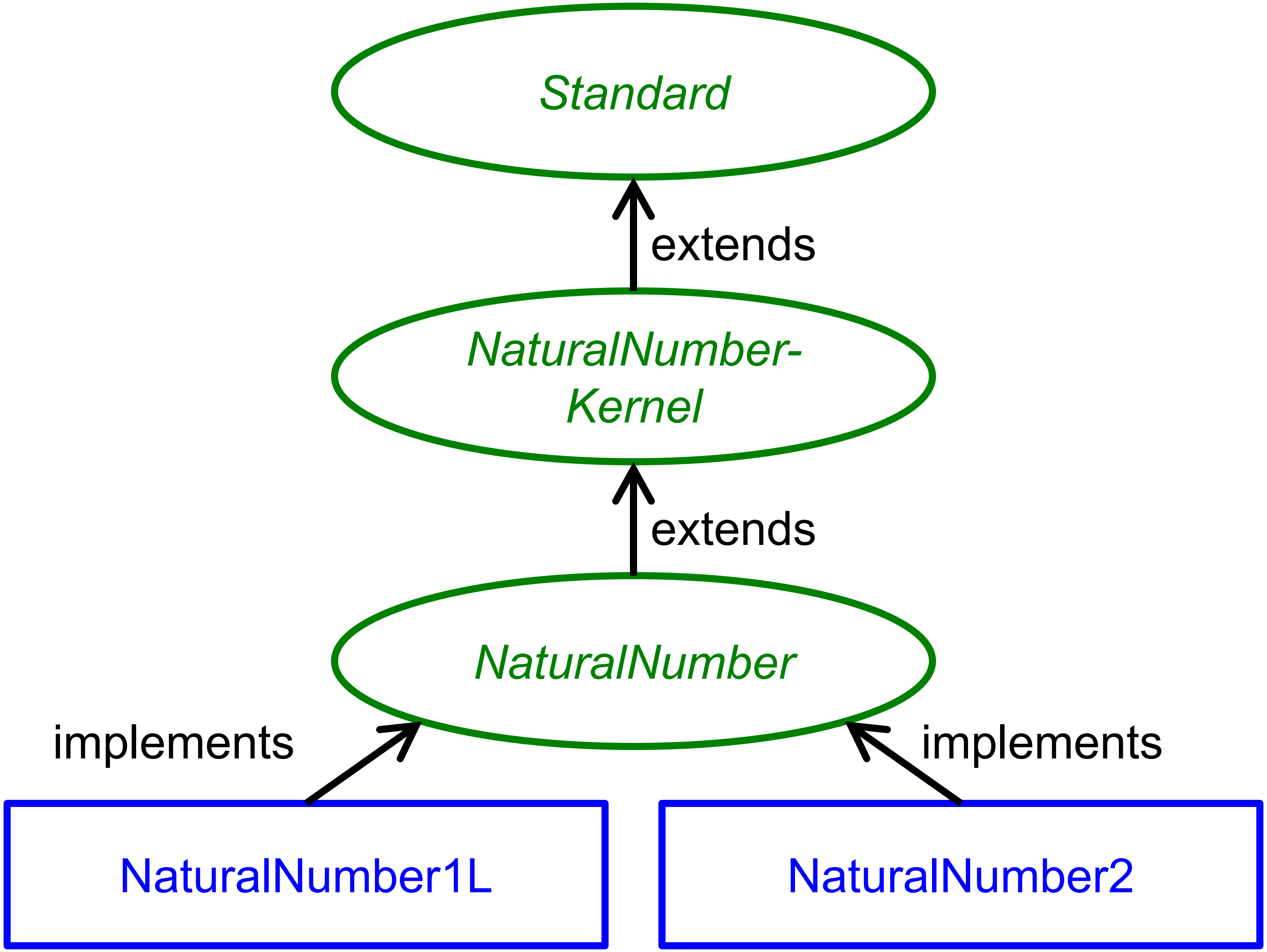
NaturalNumber

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

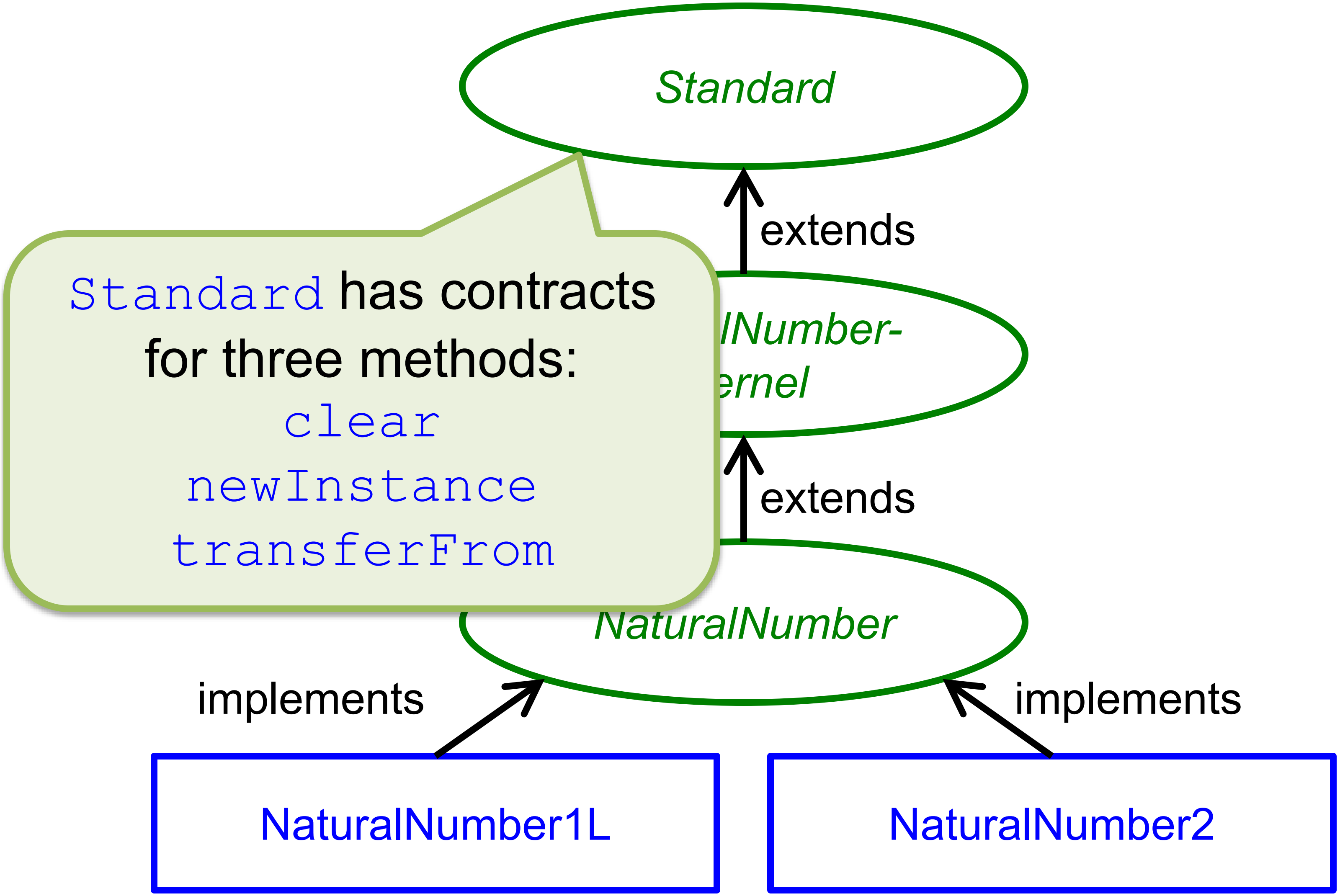
implements

implements

NaturalNumber1L

NaturalNumber2

Interfaces and Classes



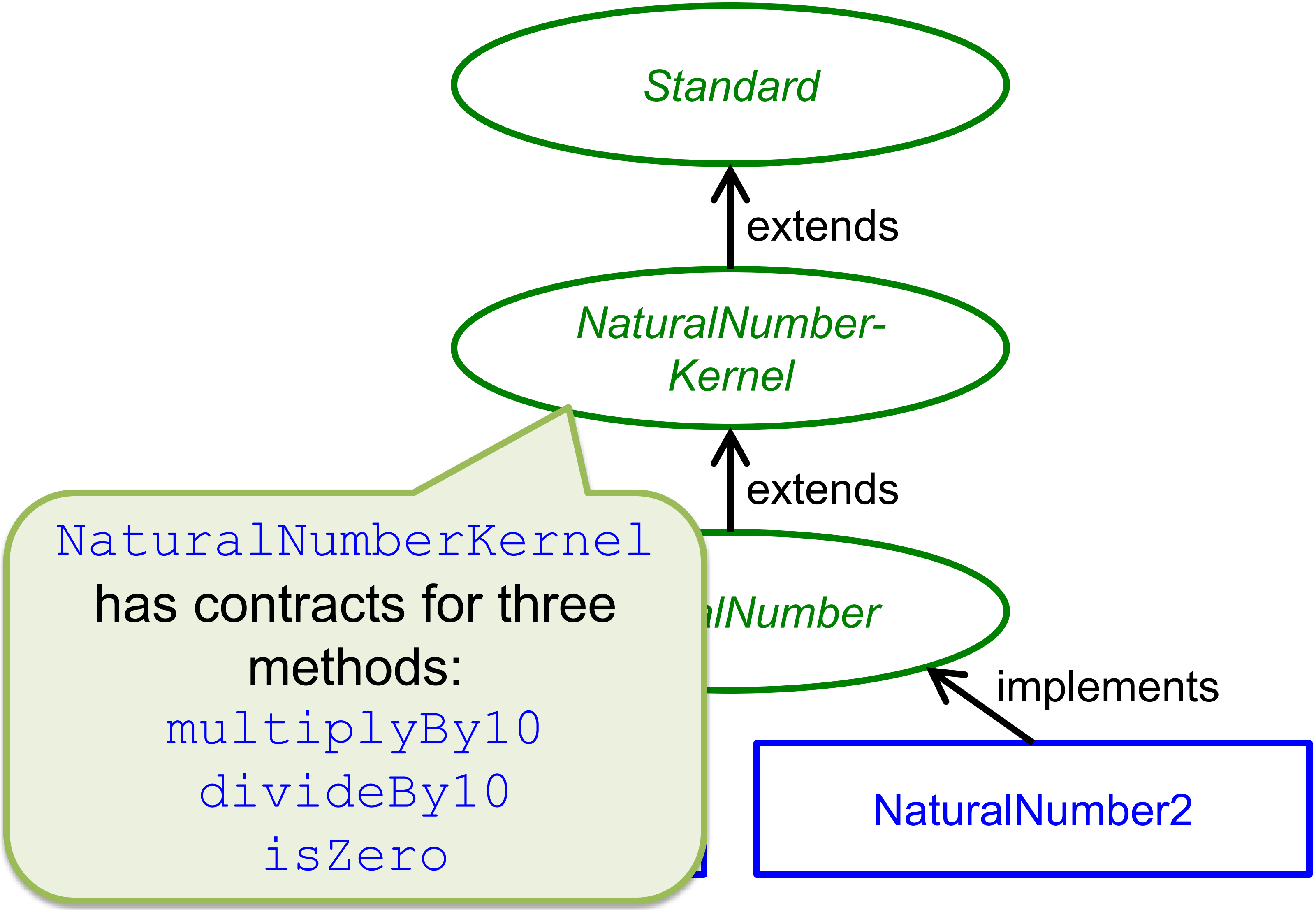
Standard has contracts for three methods:

clear

newInstance

transferFrom

Interfaces and Classes



Standard

extends

NaturalNumber-Kernel

extends

alNumber

implements

NaturalNumber2

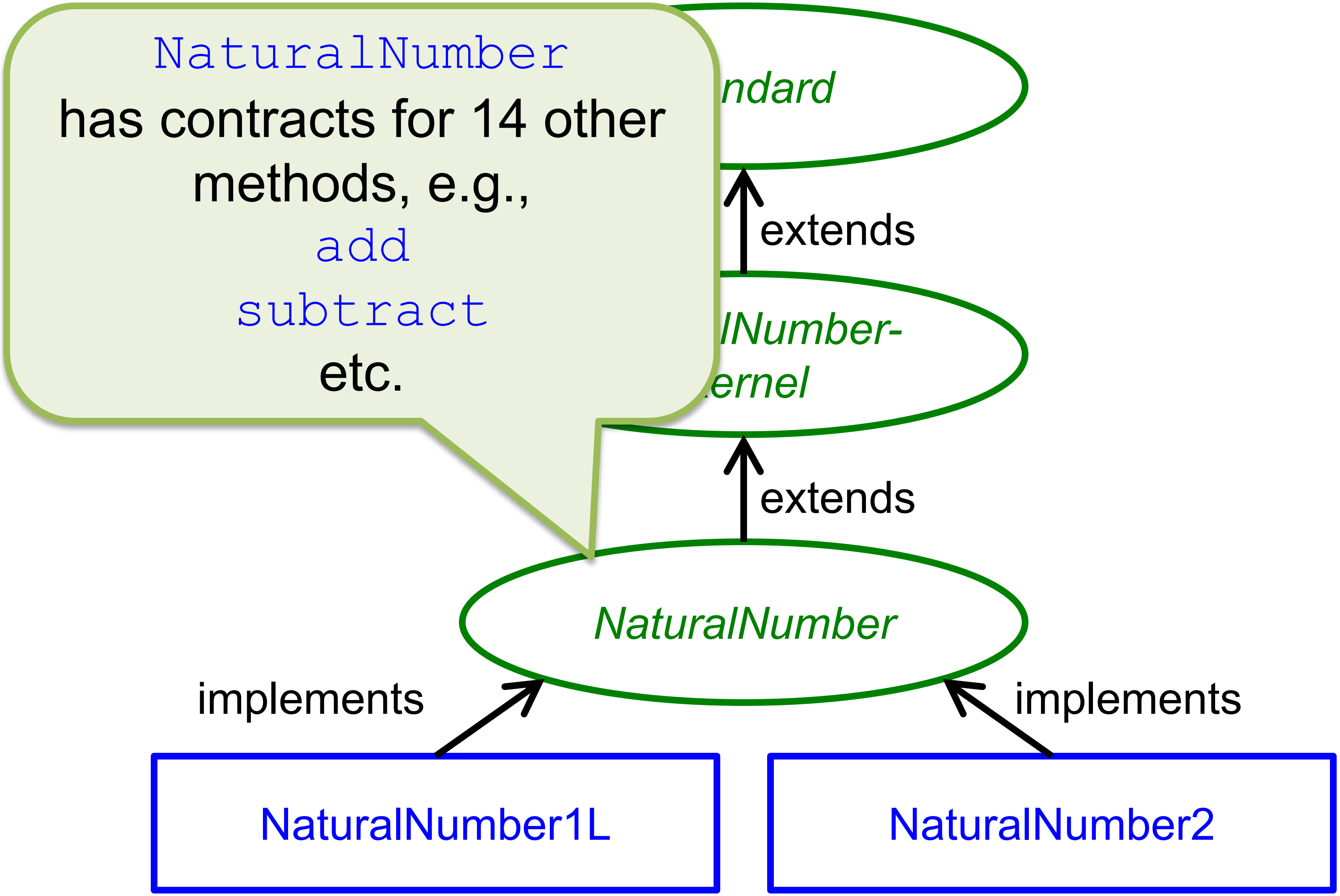
`NaturalNumberKernel` has contracts for three methods:

`multiplyBy10`

`divideBy10`

`isZero`

Interfaces and Classes



`NaturalNumber` has contracts for 14 other methods, e.g.,

`add`

`subtract`

etc.

ndard

extends

lNumber-
ernel

extends

NaturalNumber

implements

implements

NaturalNumber1L

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 `Standar`

- The interface `Standa` are part of most (nearl component families
  - Separating the ***standard*** ***ethods*** into their own interface means tha 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 `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 Kernel

- The interface Natura
  has a minimal set of m
  primitive in the Natural er
  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 `NaturalNumber` has all other methods that are *convenient to have* in the `NaturalNumber` component family
  – These ***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:

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

Mathematical Model

- The value of a Natur
is modeled as a non-
- Formally:

```
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 `Natur`
  is modeled as a non-
- Formally:

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

No-argument Constructor

- A constructor with no parameters is called a ***no-argument constructor***
- Ensures:

```
this = 0
```

Example

Code	***State***
`NaturalNumber n = new NaturalNumber2();`	

Example

Code	State
`NaturalNumber n = new NaturalNumber2();`	
	n = 0

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:

  this = *n*

Example

Code	***State***
	k = 12345678909

`NaturalNumber m = new`
`NaturalNumber2(k);`	

Example

Code	***State***
	k = 12345678909
`NaturalNumber m = new NaturalNumber2(k);`	
	k = 12345678909 *m = 12345678909*

Constructor from `int`

- There is a constructor with one parameter `int i`
- Requires:
  i >= 0
- Ensures:
  ***this** = i*

Example

Code	***State***
	j = 13
`NatualNumber n = new NaturalNumber2(j);`	

Example

Code	*State*
	j = 13

`NaturalNumber n = new`
`NaturalNumber2(j);`	

	j = 13 *n = 13*

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)
```

Constructor from String

- There is a cons
  String s
- Requires:

```
there exists n:   TURAL
      (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 cons
  `String s`
- Requires:

```
there exists n: I        AL
      (s = TO_STRI      n))
```

- Ensures:

```
s = TO_STRING(this)
```

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

Example

Code	***State***
	`s = "265"`
`NatualNumber n = new NaturalNumber2(s);`	

Example

Code	***State***
	s = "265"
`NaturalNumber n = new NaturalNumber2(s);`	
	s = "265" *n* = 265

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`

Methods for `NaturalNumber`

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

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

where is an i

`NaturalNu`

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

```
void add(NaturalNumber n)
```

- Adds `n` to **`this`**.
- Updates: **`this`**
- Ensures:

  this $= \#$***this*** $+ n$

`add`

```
void add(NaturalNumber n)
```

- Adds `n` to **`this`**.
- Updates: **`this`**
- Ensures:

  `this = #this`

The ***parameter mode*** called ***updates*** in a contract means the variable’s value *might be changed* by a call to the method.

add

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***.

add

```
void add(Natu
```

- Adds n to **this**.
- Updates: **this**
- Ensures:

```
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 **this**.

Example

Code	State
	m = 143 *k = 70*
`m.add(k);`	

Example

Code	***State***
	m = 143 *k = 70*
`m.add(k);`	
	m = 213 *k = 70*

subtract

```
void subtract(NaturalNumber n)
```

- Subtracts `n` from **`this`**.
- Updates: **`this`**
- Requires:

  this *>= n*

- Ensures:

  this *= #****this*** *- n*

subtract

```
void subtract(NaturalNumber n)
```

- Subtracts `n` from
- Updates: **`this`**
- Requires:

  this *>= n*
- Ensures:

  this *= #****this*** *- n*

Important! It could have been written as:

*#****this*** *=* ***this*** *+ n*

`subtract`

```
void subtract(NaturalNumber n)
```

- Subtracts `n` from
- Updates: **`this`**
- Requires:

  this *>= n*
- Ensures:

  this *= #****this*** *– n*

Or even as:

this *+ n = #****this***

Example

Code	State
	m = 143 *k = 70*
`m.subtract(k);`	

Example

Code	**State**

	m = 143
	k = 70

`m.subtract(k);`	

	m = 73
	k = 70

`multiply`

```
void multiply(NaturalNumber n)
```

- Multiplies **`this`** by `n`.
- Updates: **`this`**
- Ensures:

  $\textbf{\textit{this}} = \#\textbf{\textit{this}} * n$

Example

Code	State
	m = *143* *k* = *70*
`m.multiply(k);`	

Example

Code	State
	m = *143* *k* = *70*
`m.multiply(k);`	
	m = *10010* *k* = *70*

`divide`

`NaturalNumber divide(NaturalNumber n)`

- Divides **`this`** by `n`, returning the remainder.
- Updates: **`this`**
- Requires:

  `n > 0`

- Ensures:

  *`#`**`this`**` = n * `**`this`**` + divide  `**`and`***

  `0 <= divide < n`

Example

Code	**State**
	m = *143* *k* = *70*

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

Example

Code	State

	m = 143
	k = 70

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

	m = 2
	k = 70
	r = 3

power

```
void power(int p)
```

- Raises **this** to the power p.
- Updates: **this**
- Requires:

  p >= 0

- Ensures:

  ***this** = #**this** ^ (p)*

`power`

```
void power(int p)
```

- Raises **`this`** to t
- Updates: **`this`**
- Requires:

  p >= 0

- Ensures:

  this *= #**this** ^ (p)*

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

Example

Code	***State***
	m = 143 *k = 4*
`m.power(k);`	

Example

Code	State
	m = *143* *k* = *4*
`m.power(k);`	
	m = *418161601* *k* = *4*

root

void root(**int** r)

- Updates **this** to the r-th root of its incoming value.
- Updates: **this**
- Requires:

  r >= 2

- Ensures:

  ***this** ^ (r) <= #**this** < (**this** + 1) ^ (r)*

Example

Code	State

	m = 143
	k = 2

`m.root(k);`	

Example

Code	***State***
	m = 143 *k = 2*
`m.root(k);`	
	m = 11 *k = 2*

Example

Code	State

	m = 144
	k = 2

`m.root(k);`	

	m = 12
	k = 2

`copyFrom`

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

- Copies `n` to **`this`**.
- Replaces: **`this`**
- Ensures:

  this = *n*

copyFrom

```
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

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

- Copies `n` to **`this`**.
- Replaces: **`this`**
- Ensures:

  this = *n*

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

Example

Code	State
	m = 143 *k = 70*
`m.copyFrom(k);`	

Example

Code	State
	m = 143 *k* = 70
`m.copyFrom(k);`	
	m = 70 *k* = 70

compareTo

`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

Code	***State***
	m = *143* *k* = *70*

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

Example

Code	***State***
	m = 143 *k = 70*

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

	m = 143 *k = 70* *comp = 1*

Example

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

	State
	3

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

	m 143 *k = 0* *comp = 1*

multiplyBy10

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

- Multiplies **`this`** by 10 and adds `k`.
- Updates: **`this`**
- Requires:

  0 <= k < 10

- Ensures:

  this *= 10 * #****this*** *+ k*

multiplyBy10

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

- Multiplies **this** by 10 and adds
- Updates: **this**
- Requires:

  0 <= k < 10
- Ensures:

  this *= 10 * #**this** + k*

This is a kernel method.

Example

Code	State
	m = 143 *d = 7*
`m.multiplyBy10(d);`	

Example

Code	*State*
	m = 143 *d = 7*
`m.multiplyBy10(d);`	
	m = 1437 *d = 7*

`divideBy10`

`int` `divideBy10()`

- Divides **`this`** by 10 and returns the remainder.
- Updates: **`this`**
- Ensures:

  *#**this** = 10 * **this** + divideBy10 **and***
  0 <= divideBy10 < 10

divideBy10

```
int divideBy10()
```

- Divides **this** by 10 and returns e remainder.
- Updates: **this**
- Ensures:

```
#this = 10 * this + divideBy10 and
0 <= divideBy10 < 10
```

This is a kernel method.

Example

Code	***State***
	m = 1437

`int r =`
`  m.divideBy10();`	

Example

Code	***State***
	m = 1437

`int r =`
`  m.divideBy10();`	

	m = 143 *r = 7*

isZero

```
boolean isZero()
```

- Reports whether **this** is zero.
- Ensures:

*isZero = (**this** = 0)*

`isZero`

`boolean` `isZero()`

- Reports whether **`this`** is ze
- Ensures:

  *`isZero = (`**`this`**` = 0)`*

This is a kernel method.

Example

Code	*State*
	m = 143
boolean z = m.isZero();	

Example

Code	***State***
	m = 143

`boolean z =`
`m.isZero();`	

	m = 143 *z =* ***false***

clear

```
void clear()
```

- Resets **this** to an initial value.
- Clears: **this**
- Ensures:

  this *= 0*

`clear`

```
void clear()
```

- Resets **`this`** to an initial v e.
- Clears: **`this`**
- Ensures:

  this = *0*

This is a standard method.

`clear`

`void clear()`

- Resets **`this`** to an
- Clears: **`this`**
- Ensures:

  `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

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***.

`clear`

```
void clear()
```

- Resets **`this`** to an
- Clears: **`this`**
- Ensures:

  `this = 0`

The ensures clause is redundant in this case because **`this`** is a ***clears-mode parameter***.

Example

Code	State
	m = 143
`m.clear();`	

Example

Code	State
	$m = 143$
`m.clear();`	
	$m = 0$

`newInstance`

`NaturalNumber newInstance()`

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

  `newInstance = 0`

newInstance

```
NaturalNumber newIns    nce()
```

- Returns a new object with   e same implementation as **this**, h   g 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

Code	***State***
	m = 143

`NaturalNumber k =`
`  m.newInstance();`	

Example

Code	**State**
	m = 143

`NaturalNumber k =`
`  m.newInstance();`	

	m = 143 *k = 0*

transferFrom

```
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

```
void transferFrom(NaturalNu ber n)
```

- Sets **this** to the incoming va of n, and resets n to an initial value; n m be of the same implem
- Replaces: **this**
- Clears: n
- Ensures:

  this = *#n*

This is a standard method.

`transferFrom`

```
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: **`thi`**
- 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

Code	***State***
	m = 143 *k = 70*
`m.transferFrom(k);`	

Example

Code	State
	m = *143* *k* = *70*
`m.transferFrom(k);`	
	m = *70* *k* = *0*

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:

```
toString = [the string
             representation of this]
```

Example

Code	***State***
	m = 143

`String s =`
`m.toString();`	

Example

Code	***State***
	m = 143

`String s =`
`  m.toString();`	

	m = 143 *s = "143"*

Resources

- OSU CSE Components API: `NaturalNumber`
  - http://cse.osu.edu/software/common/doc/