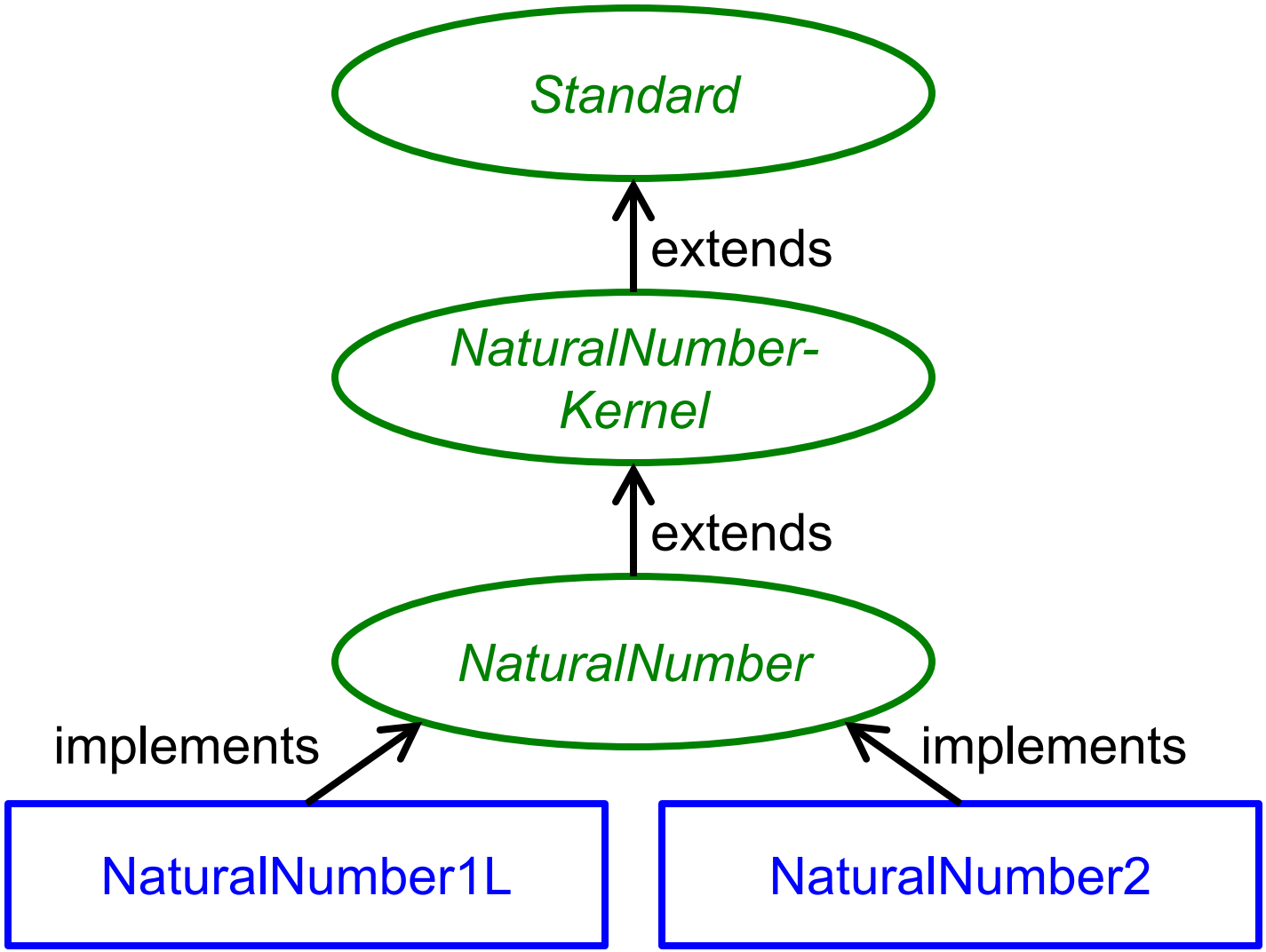
Concepts of Object-Oriented Programming

Recall...


Standard
extends
NaturalNumber-
Kernel
extends
NaturalNumber
implements
implements
NaturalNumber1L
NaturalNumber2

The “Implements” Relation

- The ***implements*** relation may hold between a class and an interface
- If `C` ***implements*** `I` then class `C` contains code for the behavior specified in interface `I`
  - This means `C` has method bodies for instance methods whose contracts are specified in `I`
  - The code for `C` looks like this:

```
class C implements I {
  // bodies for methods specified in I
}
```

The "Implements" Relation

- The ***implemen*** n a class and an
- If `C` ***implemen*** code for the be `I`
  - This means `C` has d bodies for instance methods whose c acts are specified in `I`
  - The code for `C` lo ks like this:

```
class C implements I {
  // bodies for methods specified in I
}
```

The implements relation allows you to separate contracts from their implementations — a **best practice** for component design.

The "Implements" Relation

- The ***implemen*** a class and an
- If `C` ***implemen*** code for the be
  - This means `C` has d bodies for instance methods whose c acts are specified in `I`
  - The code for `C` lo ks like this:

```
class C implements I {
  // bodies for methods specified in I
}
```

The Java compiler checks that `C` *contains bodies* for the methods in `I`, but does not check that those bodies *correctly* implement the method contracts!

The “Extends” Relation

- The ***extends*** relation may hold between:
  - Two interfaces (as on the earlier slide), or
  - Two classes
- In either case, if `B` ***extends*** `A` then `B` ***inherits*** all the methods of `A`
  - This means `B` implicitly *starts out* with all the method contracts (for an interface) or all the method bodies (for a class) that `A` has
  - `B` can then *add more* method contracts (for an interface) or method bodies (for a class)

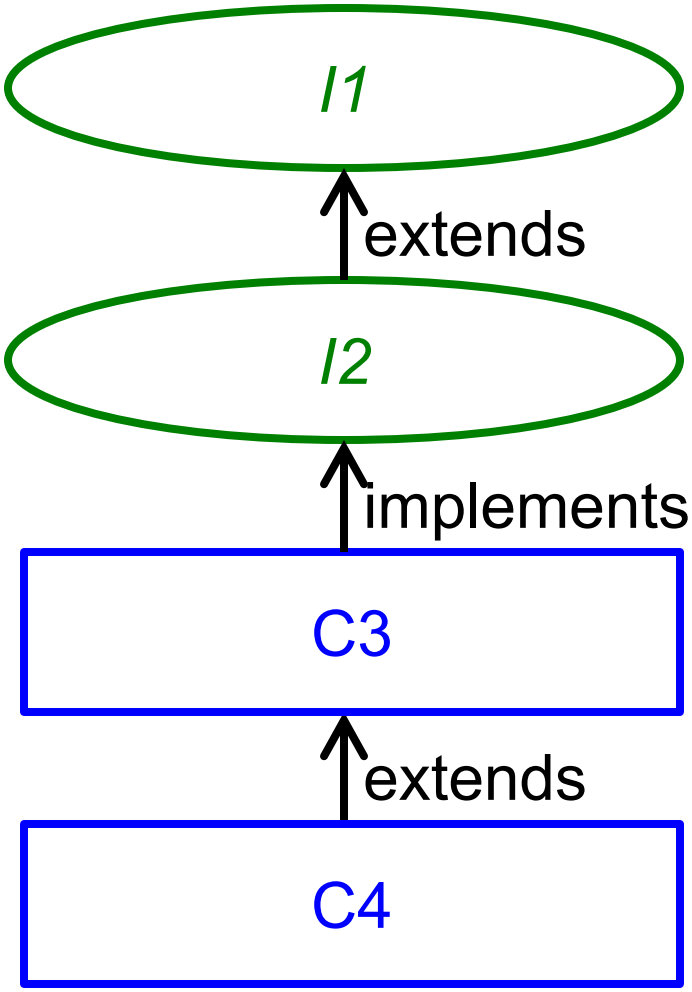
Caveats About Java Interfaces

- "If `B` ***extends*** `A` then `B` ***inherits*** all the methods of `A`"
  - Interfaces cannot have constructors
    - So there is no good place to write separate contracts for the constructors of classes that implement an interface
  - Interfaces cannot have *static* method contracts without also providing corresponding method bodies
    - So there is no good place to write separate contracts for public static methods of classes that implement an interface

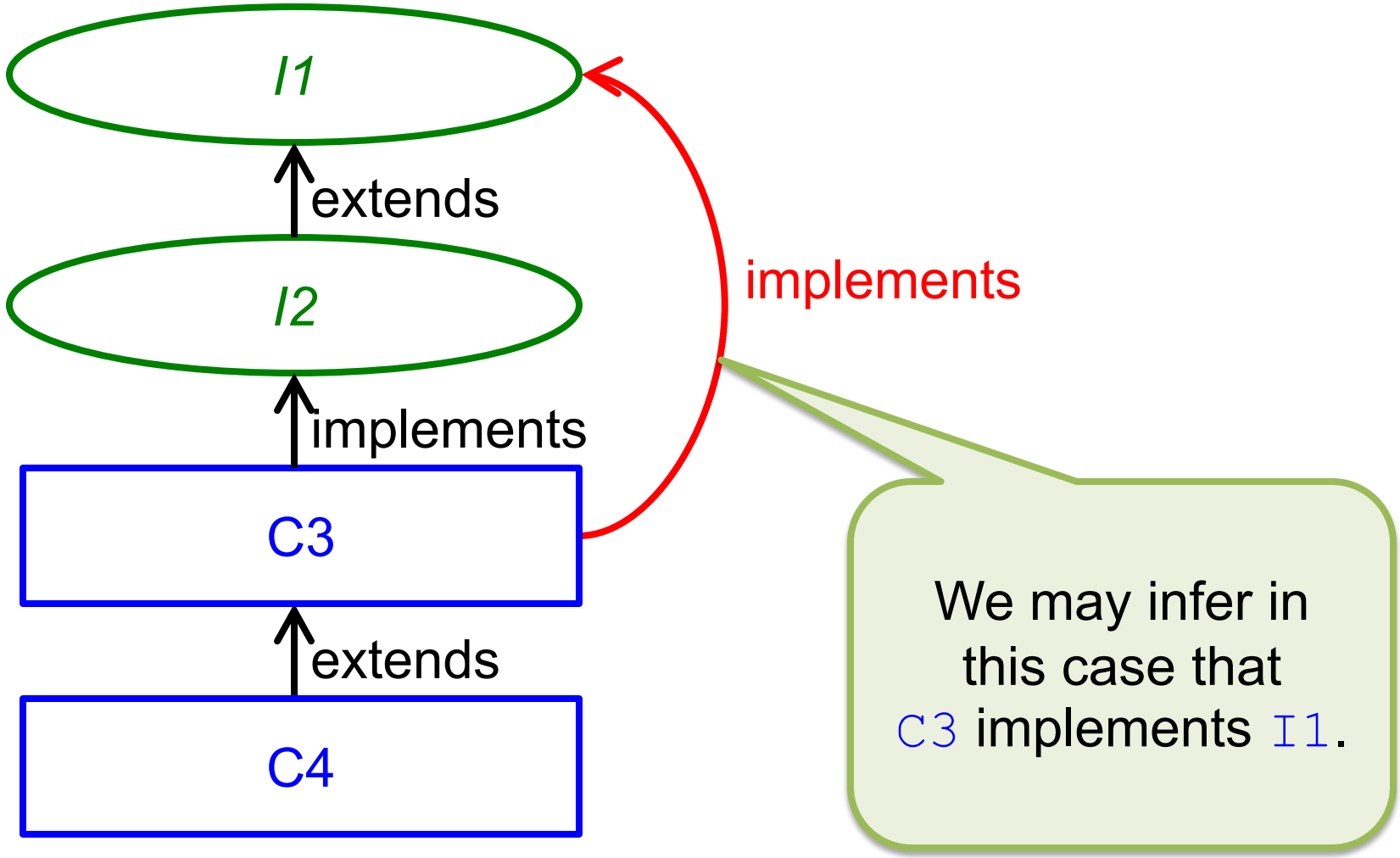
Caveats About Java Classes

- "If `B` ***extends*** `A` then `B` ***inherits*** all the methods of `A`"
  - Constructors *are not* inherited
    - So in the situation above, the class `B` must include bodies for any constructors that are expected, even if they would be identical to those of `A`
    - The bodies of the constructors in `B` generally would simply invoke the constructors of `A`, which is done using the special notation **`super`**`(…)`
  - Static methods *are* inherited

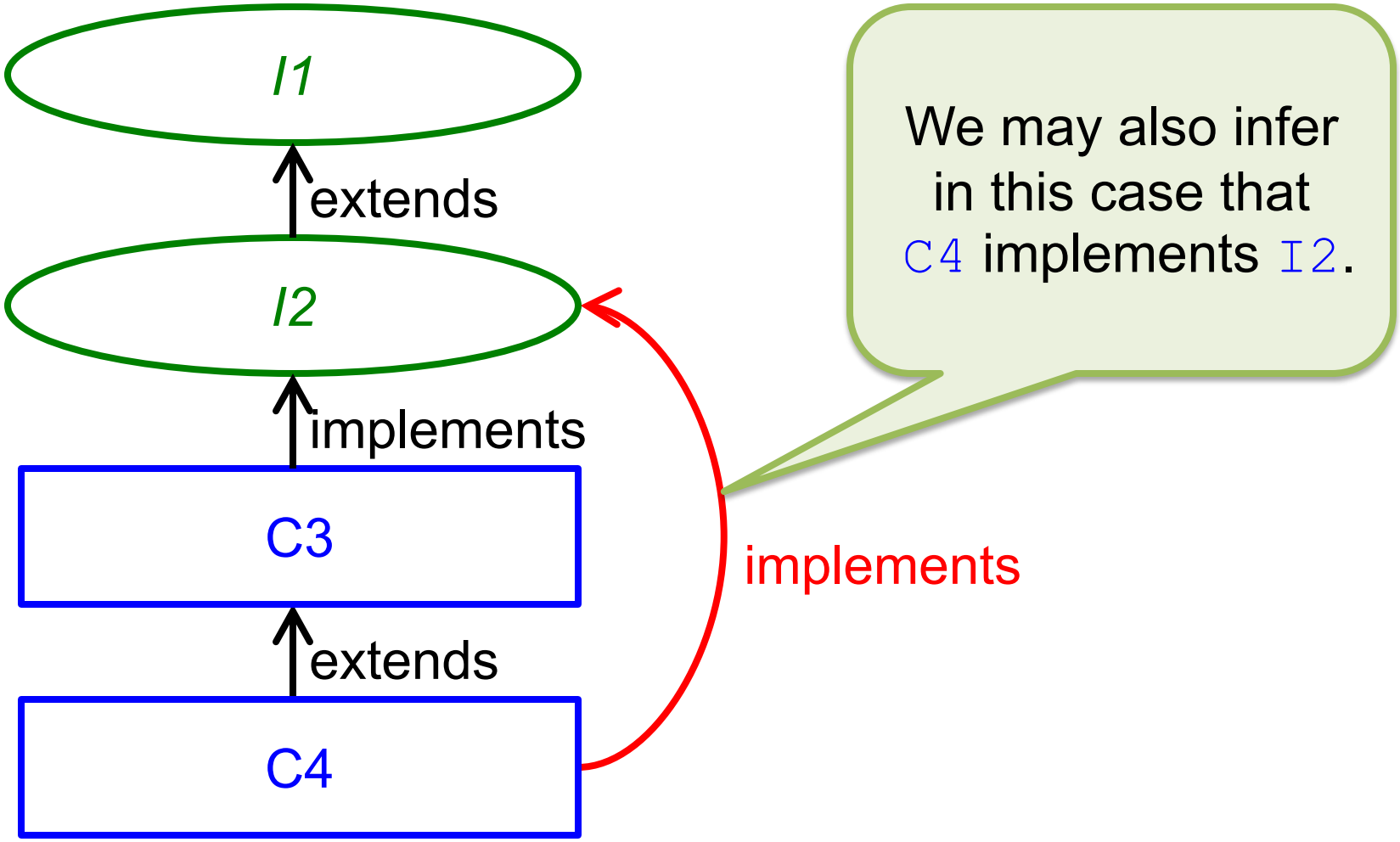
“Implements” May Be Inferred



“Implements” May Be Inferred



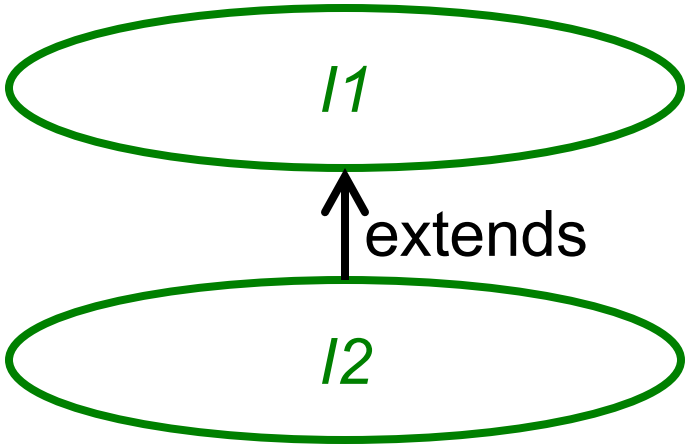
“Implements” May Be Inferred



Interface Extension

- If `I1` and `I2` are interfaces and `I2` ***extends*** `I1`, then the code for `I2` looks like this:

```
interface I2 extends I1 {
  // contracts for methods added in I2
}
```

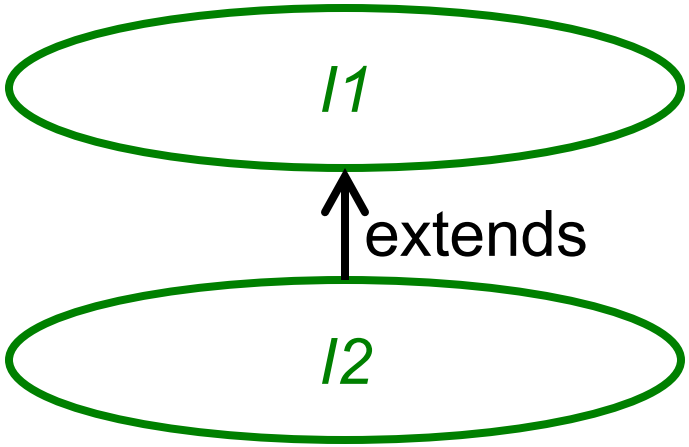


Interf

Remember, for interfaces all such methods are instance methods!

- If `I1` and `I2` are inte s and `I2` ***extends*** `I1`, then the code for ooks like this:

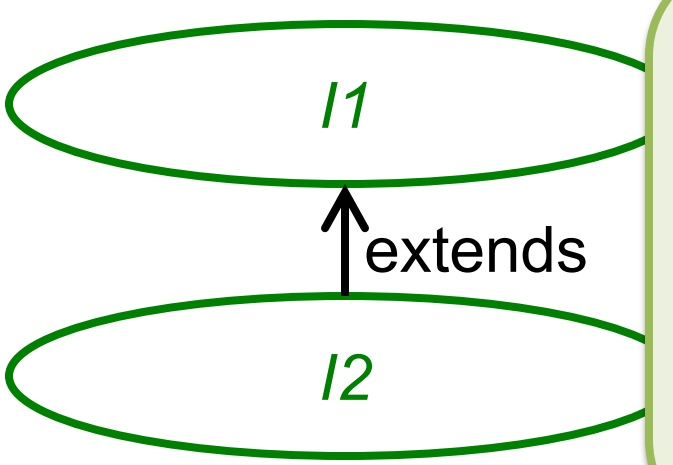
```
interface I2 extends I1 {
  // contracts for methods added in I2
}
```



Interface Extension

- If `I1` and `I2` are interfaces and `I2` ***extends*** `I1`, then the code for `I2` looks like this:

```
interface I2 extends I1 {
  // contracts for methods added in I2
}
```

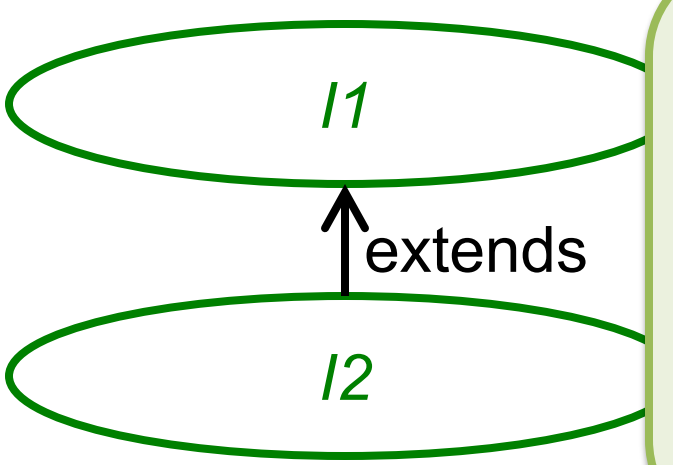


Other terminology for this situation:
`I2` is a ***subinterface*** of `I1`
`I2` is a ***derived interface*** of `I1`
`I2` is a ***child interface*** of `I1`

Interface Extension

- If `I1` and `I2` are interfaces and `I2` ***extends*** `I1`, then the code for `I2` looks like this:

```
interface I2 extends I1 {
  // contracts for methods added in I2
}
```



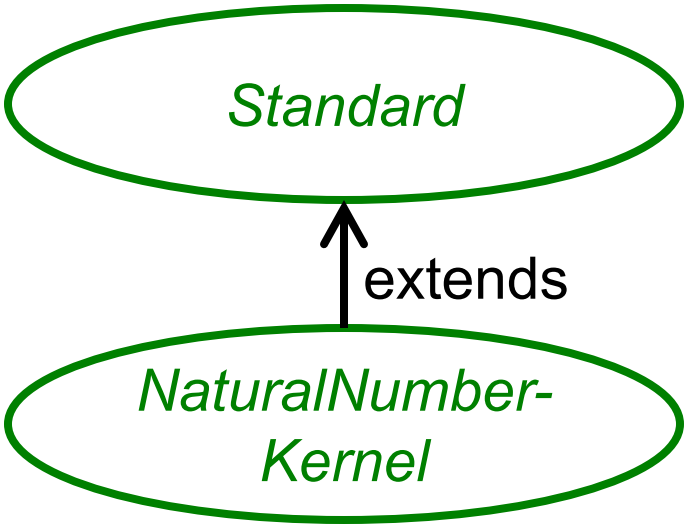
Other terminology for this situation:

`I1` is a ***superinterface*** of `I2`

`I1` is a ***base interface*** of `I2`

`I1` is a ***parent interface*** of `I2`

Example: Interface Extension




```
clear
newInstance
transferFrom
```

+

```
multiplyBy10
divideBy10
isZero
```

=

```
clear
newInstance
transferFrom
```

```
multiplyBy10
divideBy10
isZero
```

Example: Interface Extension

`NaturalNumberKernel` actually has all these methods, even though their contracts are in two separate interfaces.

NaturalNumber-Kernel

`lear`
`stance`
`ferFrom`
+
`ultiplyBy10`
`divideBy10`
`isZero`
=

`clear`
`newInstance`
`transferFrom`

`multiplyBy10`
`divideBy10`
`isZero`

Example: Interface Extension

The extends relation for interfaces allows you to separate contracts into smaller chunks — arguably a **best practice** for component design.

NaturalNumber-Kernel

```
lear
stance
erFrom
+
ultiplyBy10
divideBy10
isZero
=
```

```
clear
newInstance
transferFrom
```

```
multiplyBy10
divideBy10
isZero
```

Class Extension

- For classes, extension can serve two different purposes:
  - To add method bodies that *are not* already in the class being extended (similar to the use of extension for interfaces)
  - To ***override*** methods that *are* already implemented in the class being extended, by providing *new* method bodies for them

Class Extension

- For classes, [illegible] different purp[illegible]
  - To add metho[illegible] *are not* already in the class being [illegible]ded (similar to the use of extension for [illegible]terfaces)
  - To ***override*** methods that *are* already implemented in the class being extended, by providing *new* method bodies for them

When pronounced, this may sound like "overwrite", but that is not a correct interpretation!

Class Extension

- For classes, [illegible] different purp[illegible]
  - To add metho[illegible] the class being [illegible]ded (similar to the use of extension for [illegible]terfaces)
  - To ***override*** methods that *are* already implemented in the class being extended, by providing *new* method bodies for them

For now, we are concerned only with this use of class extension.

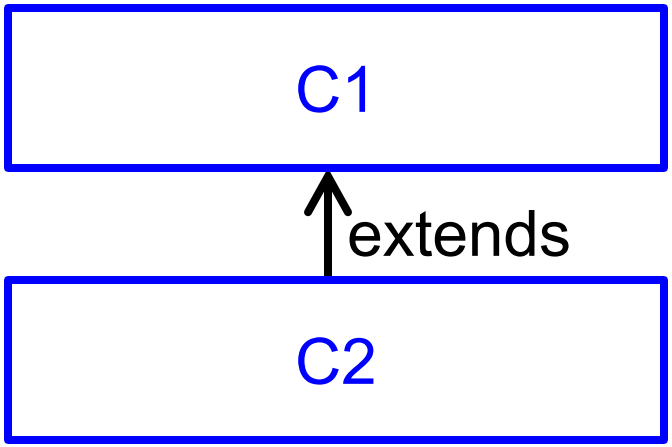
- For c
  differ
  - To a
    the class
    extensio
    interfaces)
  - To ***override*** methods that *are* already implemented in the class being extended, by providing *new* method bodies for them

Important note: ***Overriding*** a method is different from ***overloading*** a method!
A method (name) is ***overloaded*** when two or more methods have the same name, in which case the methods must differ in the number and/or types of their formal parameters (which the compiler uses to disambiguate them).

Class Extension

- If `C1` and `C2` are classes and `C2` ***extends*** `C1`, then the code for `C2` looks like this:

```
class C2 extends C1 {
  // code for methods added or
  // overridden in C2
}
```

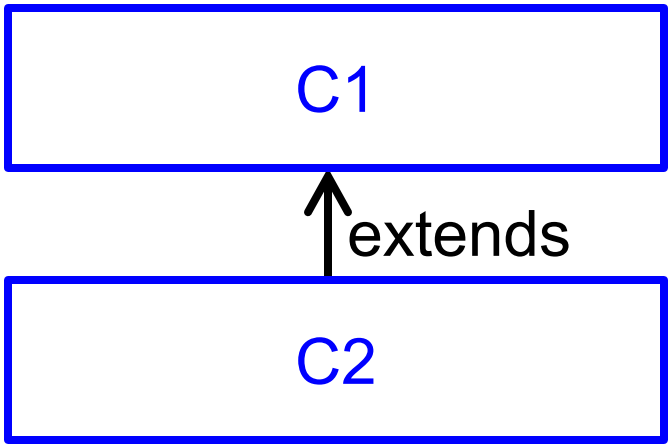


Cla

Remember, for classes these may be either static methods or instance methods.

- If `C1` and `C2` are cl… and `C2` **extends** `C1`, then the code fo… looks like this:

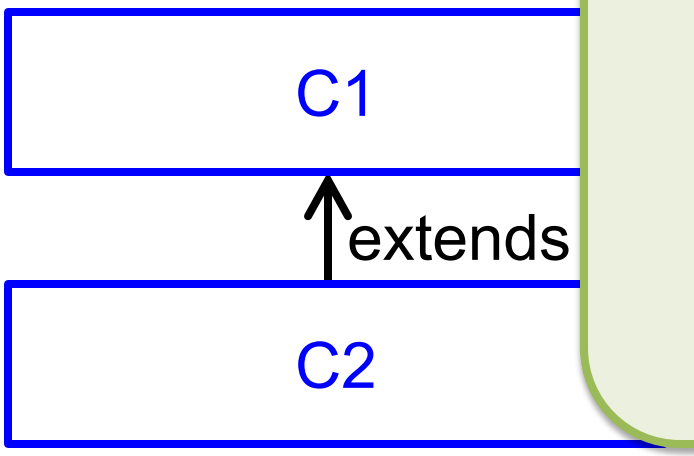
```
class C2 extends C1 {
  // code for methods added or
  // overridden in C2
}
```



Class Extension

- If `C1` and `C2` are classes and `C2` ***extends*** `C1`, then the code for `C2` looks like this:

```
class C2 extends C1 {
   // code for methods added or
   // overridden
}
```

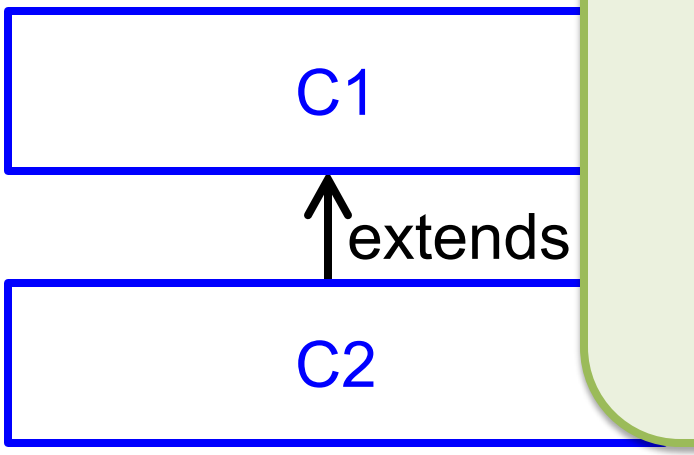


Other terminology for this situation:
`C2` is a ***subclass*** of `C1`
`C2` is a ***derived class*** of `C1`
`C2` is a ***child class*** of `C1`

Class Extension

- If `C1` and `C2` are classes and `C2` ***extends*** `C1`, then the code for `C2` looks like this:

```
class C2 extends C1 {
   // code for methods added or
   // overridden
}
```



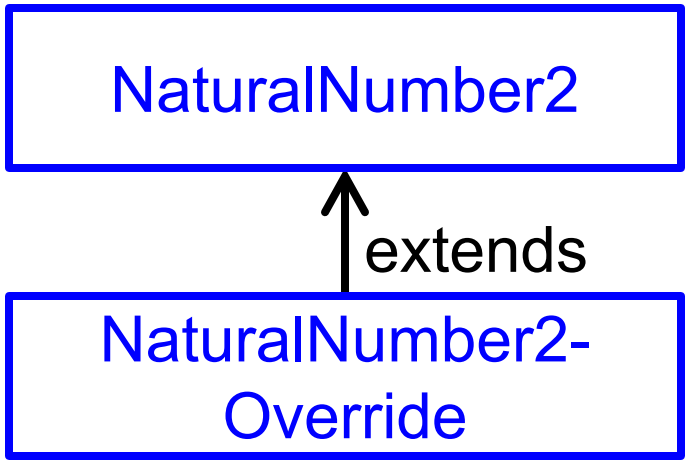
Other terminology for this situation:

`C1` is a ***superclass*** of `C2`

`C1` is a ***base class*** of `C2`

`C1` is a ***parent class*** of `C2`

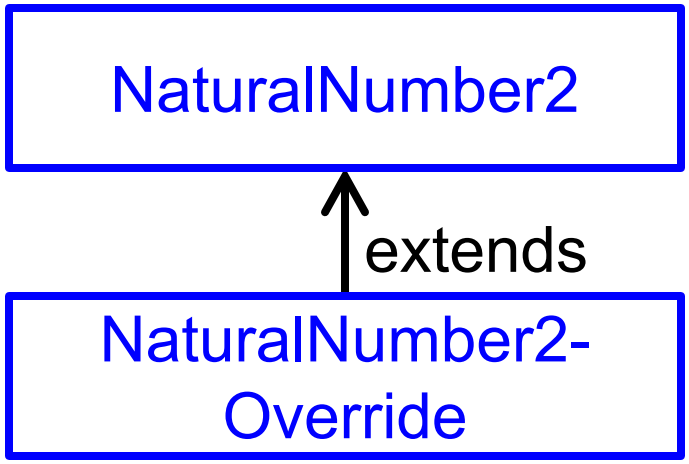
Example: Overriding a Method




```
...
power
...
```

```
power
```

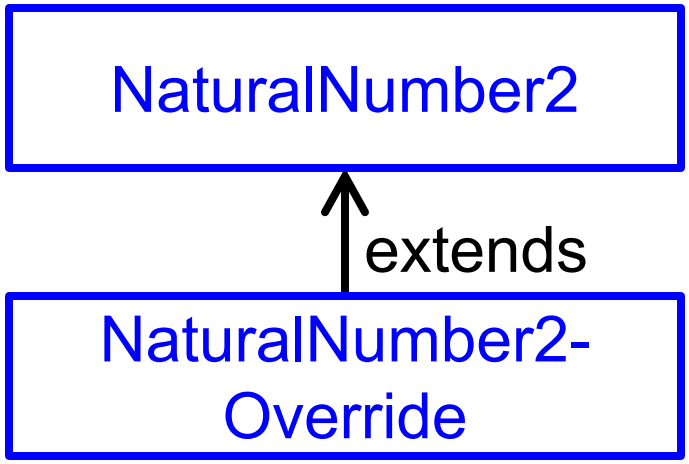
Example: Overriding a Method




```
...
power
...
power
```

There is a method body for `power` in `NaturalNumber2` ...

Example: Overriding a Method




```
...
power
...
```

```
power
```

... and there is *another* method body for `power` in `NaturalNumber2Override`.

“@Override” Annotation

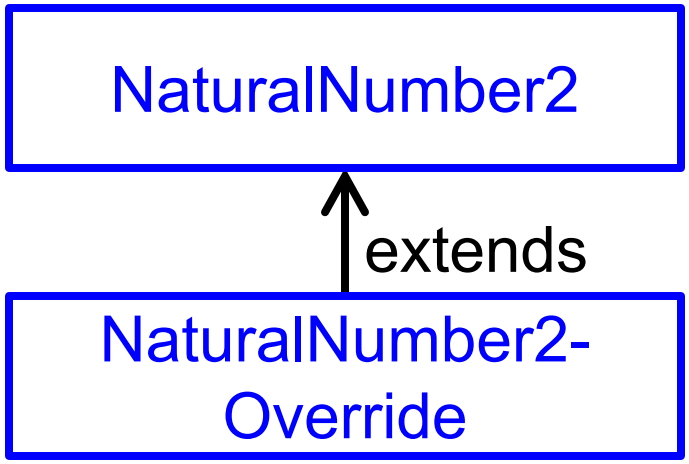
- When writing the code for the body of *either* a method
  - whose contract is from an interface being *implemented*, or
  - that overrides a method in a class being *extended*

  you preface the method body with an ***@Override annotation***

Example of “@Override”

```
@Override
public void power(int p) {
   ...
}
```

Which Method Body Is Used?




```
...
power
...
```

?

```
power
```

This raises the question:
Which method body for `power` is used when `power` is called in a client program?

Interface as Declared Type

- When a variable is declared using the name of an ***interface*** as its type, e.g.:

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

then its ***declared type*** (or ***static type***) is said to be an ***interface type***

Interface as Declared Type

- When a variable is declared using the name of an ***interface*** as its type, e.g.:

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

then its ***declared type*** (or ***static type***) is said to be an ***interface***

Here, the declared type of `k` is `NaturalNumber`.

Interface as Declared Type

- When a variable is declared using the name of an ***interface*** as its type, e.g.:

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

then its ***declared type*** (or ***static type***) is said to be an ***interface***

Best practice is for variables to be declared using an *interface* type, as shown here.

Class as Declared Type

- When a variable is declared using the name of a ***class*** as its type, e.g.:

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

then its ***declared type*** (or ***static type***) is said to be a ***class type***

Class as Declared Type

- When a variable is declared using the name of a ***class*** as its type, e.g.:

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

then its ***declared ty*** (or ***static type***) is said to be a ***cla***

Here, the declared type of `k` is `NaturalNumber2`.

Class as Declared Type

- When a variable is declared using the name of a ***class*** as its type, e.g.:

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

then its ***declared type*** (or ***static type***) is said to be a ***class***

Best practice is for variables to be declared using an *interface* type, but Java will let you use a *class* type, as shown here.

Object Type

- When a variable is ***instantiated*** (an object for it to reference is constructed), e.g.:

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

  then its ***object type*** (or ***dynamic type***) is the class type from which the constructor comes

Object Type

- When a variable
  for it to referenc

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

  then its ***object type*** (or ***dynamic type***) is the class type from which the constructor comes

Here, the object type of `k` is `NaturalNumber2`.

Declared/Object Type Rule

- Suppose we follow best practices, and:
  - The *declared type* of some variable is the interface type `I`
  - The *object type* of that variable is the class type `C`
- Then the relation `C` ***implements*** `I` must hold
  - Java enforces this rule!

Polymorphism

- Finally, back to overriding... Java and other object-oriented languages decide which method body to use for any call to an instance method based on the ***object type*** of the ***receiver***
  - This type, because it is the *class* of the constructor, is always a *class* type
- This behavior for calling methods is known as ***polymorphism***: “having many forms”

Example of Polymorphism

```
NaturalNumber k =
  new NaturalNumber2();
NaturalNumber n =
  new NaturalNumber2Override();
...
k.power(2);
n.power(2);
...
```

Example of Polymorphism

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
...
k.power(2);
n.power(2);
...
```

This call of `power` uses the method body for `power` from `NaturalNumber2` (which is the object type of `k`).

Example of Polymorphism

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
...
k.power(2);
n.power(2);
...
```

This call of `power` uses the method body for `power` from `NaturalNumber2Override` (which is the object type of `n`).

Example of Polymorphism

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
...
k.power(2);
n.power(2);
...
```

Note that the declared type of *both* `k` and `n` is `NaturalNumber`, and it does *not* determine which method body is used.

Another Example

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
NaturalNumber j = k;
...
j.power(2);
...
```

Another Example

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
NaturalNumber j = k;
...
j.power(2);
...
```

This call of `power` uses the method body for `power` from `NaturalNumber2` (which is the object type of `j`) ...

Another Example

```
NaturalNumber k =
   new NaturalNumber2();
NaturalNumber n =
   new NaturalNumber2Override();
NaturalNumber j = n;
...
j.power(2);
...
```

... but this call of `power` uses the method body for `power` from `NaturalNumber2Override` (which is the object type of `j`).