Static Methods vs. Instance Methods
Common Features

• Static and instance methods:
  – May have formal parameters, of any types
  – May return any type, or nothing (\textit{void})
  – May be \textit{public} or \textit{private}
  – May compute the same things

• Arguments are passed to all calls using the same parameter-passing mechanism
Common Features

• Static and instance methods:
  – May have formal parameters of any types
  – May return any type, or nothing (\texttt{void})
  – May be \texttt{public} or \texttt{private}
  – May compute the same things

• Arguments are passed to all calls using the same parameter-passing mechanism

This is the mechanism described earlier, termed call-by-copying or call-by-value.
Static Methods

• Are declared with the keyword `static`
  – Suppose `power` is a `static method` declared in the class `NNStaticOps`
  – Its declaration might look like this:

```java
public static void power(
    NaturalNumber n, int p)
{...}
```
Static Methods

• Are declared with the keyword `static`  
  – Suppose `power` is a `static method` declared in the class `NNStaticOps`  
  – Its declaration might look like this:

```java
public static void power(
    Natural n,
    int p)
{
    ...
}
```

Whether it is `public` or `private` is unrelated to whether it is a static or an instance method.
Static Methods

• Are called *without* a receiver
  – A call to `power` from within the class `NNExtraOps` might look like this:
    ```
    power(m, k);
    ```
  – A call to `power` from outside the class `NNExtraOps` might look like this; i.e., before a dot, the method name is *qualified* with the name of the *class* where it is declared:
    ```
    NNExtraOps.power(m, k);
    ```
Instance Methods

• Are declared *without* the keyword `static`
  – Suppose `power` is an *instance method* declared in the class `NNExtraOps`
  – Its declaration might look like this:

```java
public void power(int p)
{
...
}
```
Instance Methods

• Are declared *without* the keyword `static`
  – Suppose `power` is an *instance method* declared in the class `NNExtraOps`
  – Its declaration might look like this:

```java
public void power(int p)
{
...
}
```

Whether it is `public` or `private` is unrelated to whether it is a static or an instance method.
Instance Methods

• Are declared *without* the keyword `static`
  – Suppose `power` is an *instance method* declared in the class `NNExtraOps`
  – Its declaration might look like this:

    ```java
    public void power(int p)
    {{...}}
    ```

Why is there only one formal parameter now? The other formal parameter is `this`, which is implicit because it is an instance method.
Instance Methods

• Are called *with* a receiver
  – Suppose \( m \) is a variable of dynamic/object type `NNExtraOps` (or, it turns out, any type that extends `NNExtraOps`)
  – Then a call might look like this; i.e., before a dot is the name of the *receiver* of the call:

\[
\text{m.power}(k);
\]
Check Your Understanding

• It is easy to tell from the method’s declaration whether it is a static or instance method; how?

• If you see the following call, how can you tell whether it is a call to a static method or an instance method?

    foo.bar(x, y, z);
Why Have Two Kinds of Methods?

• There is one main reason to have instance methods: *polymorphism*

• An instance method that has exactly the same functional behavior as a static method simply *distinguishes* one formal parameter by placing it “out front”
  
  – It is the implicit formal parameter called *this*
  
  – It means there must be a *receiver* of a call to that method
Why Have Two Kinds of Methods?

- There is one main reason to have instance methods: **polymorphism**
- An instance method that has exactly the same functional behavior as a static method simply **distinguishes** one formal parameter by placing it “out front”
  - It is the implicit formal parameter called **this**
  - It means there must be a **receiver** of a call to that method

This is why an instance method seems to have one less formal parameter than a static method with exactly the same functional behavior.
Why Have Two Kinds of Methods?

- There is one main reason to have instance methods: polymorphism.
- An instance method that has exactly the same functional behavior as a static method simply distinguishes one formal parameter by placing it “out front”
  - It is the implicit formal parameter called `this`.
  - It means there must be a `receiver` of a call to that method.

Recall that polymorphism is the mechanism that selects the method body to be executed based on the dynamic/object type of the receiver.
Implications for Contracts

• Unfortunately, in Java, you may not declare a static method in an interface!
• This limitation, along with the flexibility added by polymorphism, is a good reason to (generally) prefer instance methods to static methods in Java, all other things being equal
Implications for Contracts

• Unfortunately, in Java, you may not declare a static method in an interface!
• This limitation, along with the flexibility added by polymorphism, is a good reason to (generally) prefer instance methods to static methods in Java, all other things being equal.

You might think about why this is the case... Googling it will lead you to the (valid) conclusion that “experts” find the limitation hard to explain, even to each other!
Implications for Method Bodies

• The variables in scope in a static method’s body are its formal parameters
• The variables in scope in an instance method’s body are its explicit formal parameters, plus the implicit formal parameter this
• The bodies do not otherwise differ