Virtual methods in C# cheat sheet

(version 2 - 2025/11/19)

Assume:

```
class A {
    public ... void Update(X x) { α }
}
A a = ...;
X x = ...;
```

concept 1: method ≠ concept 2: implementation

↯ What does `a.Update(x)` mean in this context?

✓ **Answer:** C# compiler always chooses method based on compile-time expression type of `a` ~ type `A` here. So, it always means "call `A.Update(X)` method" here.

↯ Does the C# compiler choice depend on content of `a` or `x` variables?

✓ **Answer: No**, it never depends on runtime content of variables `a` or `x`!

Assume the `A.Update(X)` method is **non-virtual**:

```
public void Update(X x) { α }
```

✓ **Observation:** `a.Update(x)` always means "call method `A.Update(X)`" **!!!**

✓ **Observation:** Because `A.Update(X)` is a non-virtual method, *static dispatch* is always used to call it ~ CLR calls `A.Update(X)` implementation provided in type `A` ~ always code α.

Assume the `A.Update(X)` method is **virtual**:

```
public virtual void Update(X x) { α }
```

✓ **Observation:** `a.Update(x)` always means "call method `A.Update(X)`" **!!!**

✓ **Observation:** `virtual` always means **new virtual method** ~ ***new entry*** in type `A` VMT.

✓ **Observation:** Because `A.Update(X)` is a virtual method, *dynamic dispatch* is always used to call it ~ CLR looks up `A.Update(X)` implementation at runtime according to actual content of `a` variable.

‼ **Note:** There is one exception – if we call `A.Update(X)` via `base.Update(X)`, *static dispatch* is always used to call it ~ CLR calls `A.Update(X)` implementation provided in "base" class; if our base class is type `A` ~ always code α.

Variant 0) Assume**:**

```
A a = new A();
X x = ...;
```

✓ **Observation:** `a.Update(x)` calls code α.

Variant 1) Assume**:**

```
class B : A {
    public new void Update(X x) { β1 } or alternatively: public void Update(X x) { β1 }
}
A a = new B();
X x = ...;
```

↯ Is β1 a new implementation of `A.Update(X)`?

✓ **Answer: No** – in both cases it is a **new non-virtual** method with implementation β1.

↯ So, does `a.Update(x)` call code β1 now?

✓ **Answer: No** – it will still call the only `A.Update(X)` implementation available in type `B` ~ implementation α.

Variant 2) Assume**:**

```
class B : A {
    public override void Update(X x) { β2 }
}
A a = new B();
X x = ...;
```

↯ Is β2 a new implementation of `A.Update(X)`?

✓ **Answer: Yes** – it is **not a new method**, it is a **new implementation** β2 of **inherited virtual** method `A.Update(X)`.

↯ So, does `a.Update(x)` call code β2 now?

✓ **Answer: Yes** – it will call the `A.Update(X)` implementation β2.

✓ **Observation:** `override` always means providing new implementation for inherited virtual method ~ ***overwriting existing entry*** in type `B` VMT.

Interfaces in C# cheat sheet

(version 2025/11/18)

Assume:

```
interface I {
    public void Update(X x);
}
I i = ...;
X x = ...;
```

↯ What does `i.Update(x)` mean in this context?

✓ **Answer:** C# compiler always chooses method based on compile-time expression type of `i` ~ type `I` here. So, it always means "call `I.Update(X)` method" here.

↯ Does the C# compiler choice depend on content of `i` or `x` variables?

✓ **Answer: No**, it never depends on runtime content of variables `i` or `x`!

Assume the `I.Update(X)` contract is fulfilled by `A.Update(X)` method:

```
class A : I {
    public void Update(X x) { α }
}
I i = ...;
X x = ...;
```

✓ **Observation:** Implementing interface by "`: I`" always means **fulfill the contract** ~ filling entry in type `A` method table for interface `I`.

✓ **Observation:** `i.Update(x)` always means "call method `I.Update(X)`" **!!!**

✓ **Observation:** Because `I.Update(X)` is an interface method, CLR looks up at runtime **what method** fulfills the contract for actual content of `i` variable (it is a variant of *dynamic dispatch*).

Variant 0) Assume**:**

```
I i = new A();
X x = ...;
```

✓ **Observation:** `i.Update(x)` calls code α.

Variant 1) Assume**:**

```
class B : A {
    public new void Update(X x) { β1 } or alternatively: public void Update(X x) { β1 }
}
I i = new B();
X x = ...;
```

↯ Does the new **non-virtual** method `B.Update(X)` fulfill the `I.Update(X)` contract?

✓ **Answer: No** – in both cases the contract given by interface `I` is fulfilled by `A` and inherited without change to `B`.

↯ So, does `i.Update(x)` call code β1 now?

✓ **Answer: No** – it will still call the `A.Update(X)` method, as `A.Update(X)` fulfills `I.Update(X)` contract for both `A` and `B` types ~ implementation α gets called.

Variant 2) Assume**:**

```
class B : A, I {
    public new void Update(X x) { β1 } or alternatively: public void Update(X x) { β2 }
}
I i = new B();
X x = ...;
```

↯ Does the new **non-virtual** method `B.Update(X)` fulfill the `I.Update(X)` contract?

✓ **Answer: Yes** – in both cases the contract given by interface `I` is fulfilled in `B` with methods available in `B` (declared or inherited) ~ filling (overwriting) entry in type `B` method table for interface `I`.

↯ So, does `i.Update(x)` call code β2 now?

✓ **Answer: Yes** – it will call the `B.Update(X)` method, as `B.Update(X)` fulfills `I.Update(X)` contract for `B` type ~ implementation β2 gets called.

✓ **Observation:** If `B.Update(X)` is a **virtual** method, CLR **always** uses *dynamic dispatch* to call it ~ it will look up the virtual method implementation at runtime in VMT of the target instance type.