Consider the object drawn to the right. It’s an object of class S, which is declared as a subclass of class C. Three different variables contain a pointer to this object: s, c, and ob. They were declared like this:

Object ob;
C c;
S s;

They point at the same object, but because of how the variables were declared, each looks at the object differently:

Variable ob views it as an Object.
Variable c views it as a C.
Variable s views it as an S.

Now note that execution of the assignment

```
ob = new Object();
```

will change variable ob to point to an object of class Object, as shown to the right. Consider a call ob.c(5). With ob pointing at this object of class Object, there is no method to call!

The Java developers wanted to ensure that such a situation could never happen at runtime. They wanted to ensure that a call like ob.c(5) would compile only if it was guaranteed that method c is always available at runtime. To ensure this property, they imposed this reference rule:

**Compile-time reference rule.** For a variable p declared as

```
P p;
```

The variable reference p.v (or method call p.m(…)) is syntactically correct and can be compiled only if v (or method m) is declared in P or is inherited by P.

**Examples**

- `ob.equals(...)` and `ob.toString()` are syntactically correct.
- `ob.b`, `ob.c(5)`, `ob.getB()`, `ob.f`, `ob.setF(5)` and `ob.s(int)` are syntactically incorrect and will not compile.
- `c.equals(...)`, `c.toString()`, `c.b`, `c.c(5)`, and `c.getB()` are syntactically correct.
- `c.f`, `c.setF(5)`, and `c.s(int)` are syntactically incorrect and will not compile.
- `s.equals(...)`, `s.toString()`, `s.b`, `s.c(5)`, `s.getB()`, `s.f`, `s.setF(5)`, and `s.s(int)` are syntactically correct.

Please remember that this reference rule is a syntactic rule, to be checked at compile-time, and the program will not compile if some variable or method reference violates this rule.