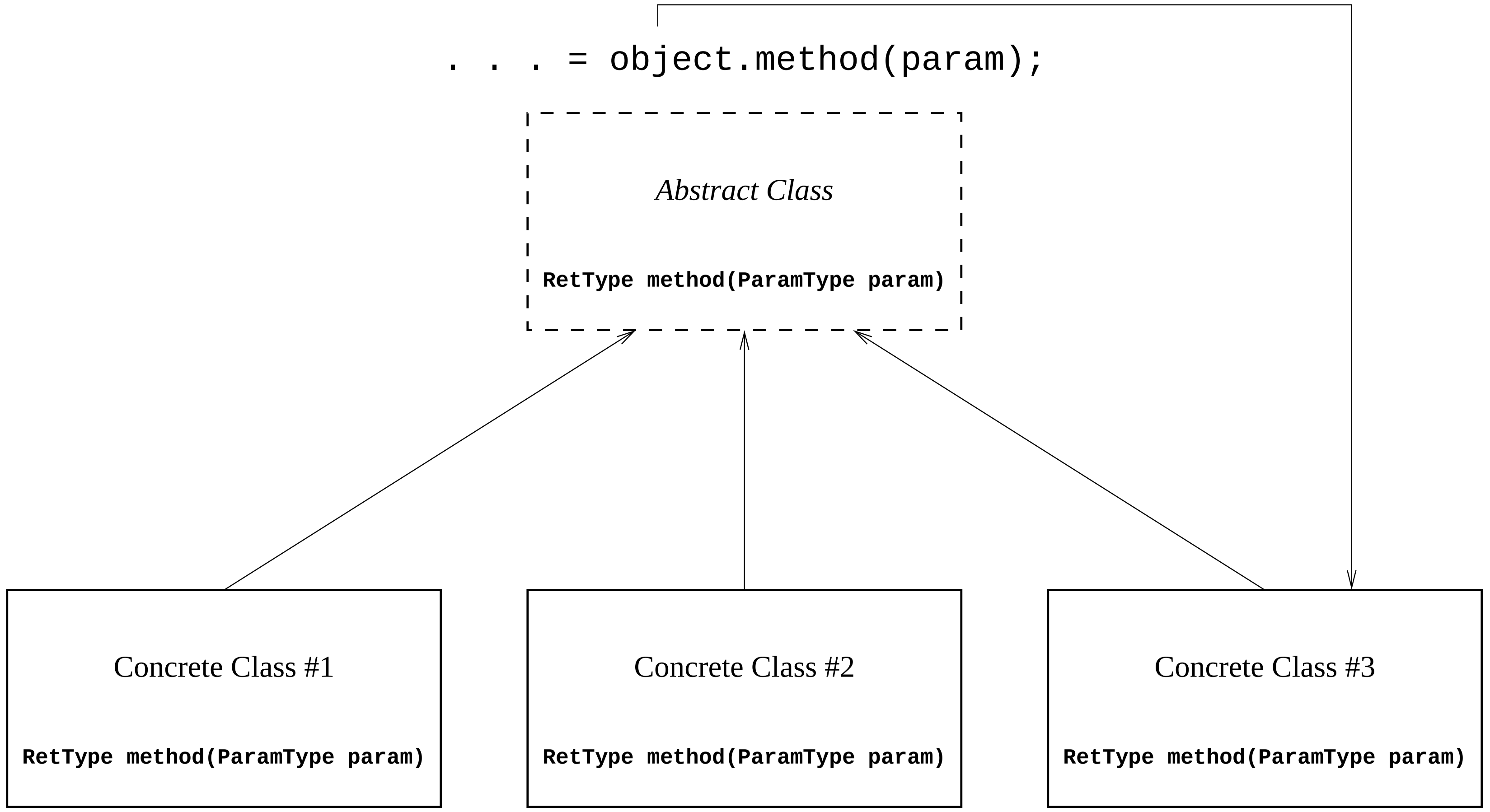
Standard Method Invocation Via *Single-Dispatching*

- When a method is performed on an object, the resultant behaviour is defined by the implementation of that method in the object's class.



```
. . . = object.method(param);
```

Abstract Class
RetType method(ParamType param)
Concrete Class #1
RetType method(ParamType param)
Concrete Class #2
RetType method(ParamType param)
Concrete Class #3
RetType method(ParamType param)

The Limitation

- Sometimes the behaviour must also be determined by the type of the parameter object.

The Situation

- Suppose that there is a class hierarchy with superclass SuperClass and subclasses SubA, SubB and SubC.

- Suppose that we have a piece of code that reads:

```
SuperClass a = getTarget();
SuperClass b = getParameter();
a.commonMethod(b);
```

getTarget() and getParameter() can return any of the subclasses.

The Situation Continues . . .

- Each of the subclasses has its own implementation of `commonMethod()`.
- But there are several versions of `commonMethod()`, each taking one of the sibling classes as input:

```
void commonMethod(SubA param) { . . . }
void commonMethod(SubB param) { . . . }
void commonMethod(SubC param) { . . . }
```

The Situation Continues . . .

- We know that the `commonMethod()` that executes will be determined by the type of the object referenced by `a`.
  - If `a` references an object of type `SubA`, then `SubA`'s `commonMethod()` will be called;
  - If `a` references an object of type `SubB`, then `SubB`'s `commonMethod()` will be called;
  - *etc.*

The Problem

- But, the Java compiler cannot necessarily determine the type of the object referenced by `b`. So, it cannot choose the appropriate method implementation.
  - For example,

```
. . .
void method(SuperClass a, SuperClass b)
{
    . . .
    a.commonMethod(b);
    . . .
```

A *Strawman* Solution

- One solution would be to declare a single `commonMethod()` with parameter type `SuperClass`, to test the type of the parameter object, and to execute different code as a result:

```
void commonMethod(SuperClass param)
{
    if (param instanceof SubA) {
        . . .
    } else if (param instanceof SubB) {
        . . .
    } else if (param instanceof SubC) {
        . . .
    } else
        . . . // probably throws an exception
}
```

Don't do that!

- This is, however, contrary to the principles of object-oriented programming.
  - If a new class of parameter object was added, then the test code would also have to be modified to accomodate the new class (a maintenance issue).

A Better Solution: *Double-Dispatching*

- A better solution is to make use of the polymorphic nature of the language and to use a technique known as double dispatching.
  - This involves adding a new method (we'll call this a secondary method) to the classes of all the potential parameter objects and then calling this from the original method with the receiver as a parameter.
    - The secondary method's name is typically constructed from the primary method's name followed by the class name of the original receiver.

Double-Dispatching (cont.)

- For example,

```
class SubA extends SuperClass {
    . . .
    void commonMethod(SuperClass param)
    {
        param.commonMethodFromSubA(this);
    }

    void commonMethodFromSubA(SubA param) { . . . }

    void commonMethodFromSubB(SubB param) { . . . }

    void commonMethodFromSubC(SubC param) { . . . }
    . . .
```