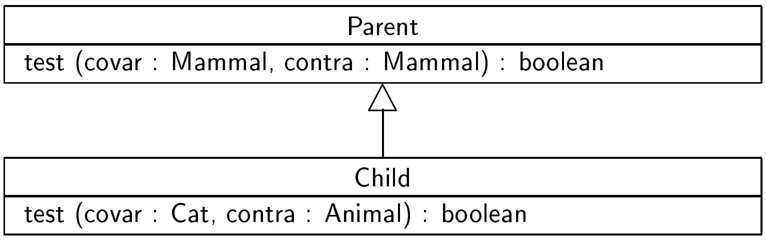
## 11.6 Covariance and Contravariance

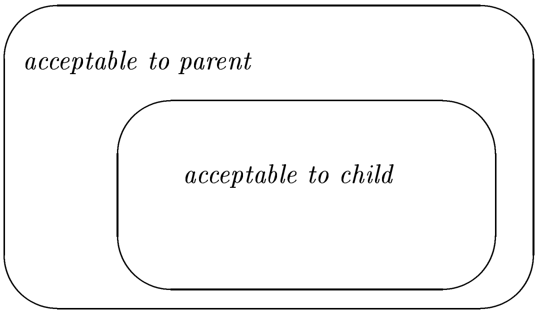
Frequently it seems as if it would be useful if the type signature of a method in a child class could be different from that found in the parent class. A common example, one we will explore in a subsequent section, is a method equals used to compare two objects. Since normally one is only interested in comparing objects of the same type, it would seem to make sense that the argument in the child class should be the child class type. Unfortunately, the whole concept of changing type signatures is fraught with subtle difficulties, as we will explore in this section.

One seldom wants to change the type signatures arbitrarily, but typically move either up or down the type hierarchy. The term *covariant* change is used when a type moves down the type hierarchy, in the same direction as the child class. The term *contravariant* is used for the opposite, when a type moves up the class hierarchy, in the opposite direction as subclassing. These two are shown in the following class hierarchy, where in the child class the first argument (named covar) moves from Mammal to the more specific type Cat, while the second argument is changed in a contravariant fashion to the more general type Animal:




The impact of covariant or contravariant change is complicated by the fact that types can be used in a variety of different ways. The effect of changing a pass-by-value parameter is different from the effect of changing a pass-by-reference parameter, which is in turn different from the effect of changing the return type for a procedure. For this reason it is slightly easier to think about the problem using sets, and consider the relationships between the set of values are are acceptable (for example, as an argument) to the parent in relationship to the set of values that are acceptable to the child.

We first consider what can happen if the set of values acceptable to the child is smaller than the set of values acceptable to the parent. This can happen, for example, in a covariant change to a pass-by-value parameter. (The parent class has a parameter of type Mammal, and the child class restricts the same parameter to the type Cat).



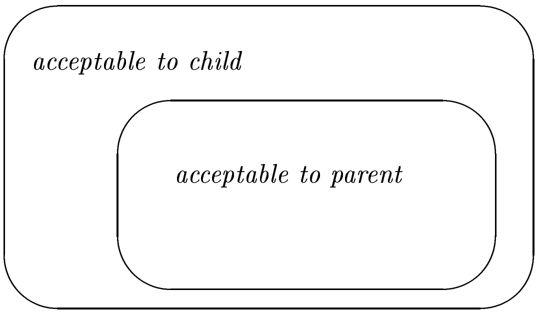

A problem occurs when this covariant change runs into the principle of substitution. According to the principle of substitution, we should be able to create a value using as declaration the parent type, but assign it a value from the child class:

```
Parent aValue = new Child();
aValue.test(new Dog(), new Mammal()); // note type of first argument
```

As far as the compiler is concerned, the first argument is perfectly acceptable, since the declaration insists only that the value be type Mammal. But the invocation will bind the message to the method in the child class, which is prepared only to accept values of type Cat. The result will almost certainly result in completely erroneous and catastrophic results.

It is occasionally proposed that run-time checks could be used to detect this condition, allowing at least a graceful error reporting, albeit at run-time rather than at compile time. However, note that such checks are never necessary when the child is being used as an instance of the child class, but are only necessary when the child is being used as an instance of the parent class. Thus, in a large percentage, perhaps the majority, of cases such run-time checks would be superfluous.

An error will not occur in the opposite case, where a contravariant change to a by-value parameter increases the range of values the child class is willing to handle. In this case neither the parent class nor the child class can pass an unacceptable value to the method.



However, that is only true for pass-by-value parameters. When we consider a change to the result type of a method the situation is exactly the reverse. Suppose a method in the parent class returns a value of type Mammal, and the child class includes a contravariant change that extends this to Animal, thereby making the set of values acceptable to the child *larger* than the set of values acceptable to the parent. Once again we run into problems with the principle of substitution. It would be perfectly legal for the child class to return a value of type Bird, since locally a bird satisfies the typing restrictions.

```
class Parent {
    Mammal test ( ) {
        return new Cat();
    }
}

class  Child extends Parent {
    Animal test () {
        return new Bird();
    }
}
```

But it is also legal for a variable of the parent class to hold an instance of the child class, and for the result of executing the method in question to be assigned to a variable of type Mammal, since as far as the compiler is concerned the result of the method fits that class designation.

```
Parent aValue = new Child();
Mammal result = aValue.test();
```

The consequence will be the variable of type Mammal holding a non-mammal value, with no typing error having been reported.

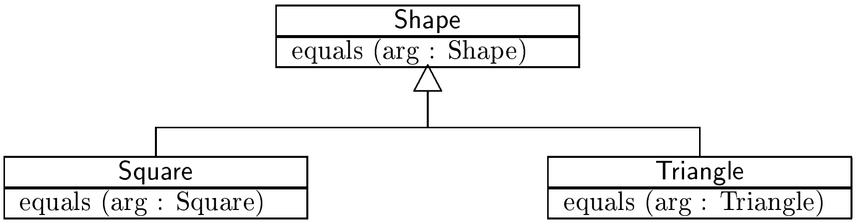
A pass-by-reference parameter can be used to pass information both into and out of a procedure. Therefore both of these errors could arise if any change whatsoever is permitted in such a value.

It is possible to specify a language so that both covariant and contravariant overriding is permitted in certain situations. Example of languages that permit this include Eiffel [Rist 1995] and Sather [**?**]. However, most language designers have opted to avoid this problem altogether by using a rule that might be termed *novariance*, namely, that a child class is not allowed to change the type signature of an overridden method in any fashion.

### Equality Testing

The example where covariant modification of type signatures often seems most compelling is in the implementation of equality and comparison tests. Examining this situation will provide a concrete illustration of the problems that can arize.

Imagine we have the following class hierarchy, where the equals method in the parent class always returns false, whereas the overridden method in the child classes correctly handle the comparison of triangles to triangle and of squares to squares.




In the imaginary language we envision here, consider what meaning might be assigned to an attempt to compare a triangle to a square:

```
Triangle aTriangle;
Square aSquare;

if aTriangle.equals(aSquare) ...
```

There are two possibilities:

- The search for a method is based solely on the receiver, a triangle, and yields the trangle method, which requires a triangle as argument. Thus, the use of a square as argument results in a compiler error.
- The search for a method is based both on the receiver and on the argument type signatures. Since the argument does not match the method in class Triangle, the method in the parent class Shape is then invoked.

Selecting either interpretation leads to trouble as soon as we consider the principle of substitution. Suppose we create a shape variable and assign it the triangle value. Comparing the triangle and the shape is then effectively comparing a value to itself. But the first interpretation results in an error, and the second interpretation results in the nonsensical result that the value is not equal to itself.

```
Shape aShape = aTriangle;
if aTriangle.equals(aShape) ...
```

A similar incongruity occurs if the parent variable is used as the receiver. Since the only reasonable method to use is the one defined in the parent class, the test yields an unexpected false value:

```
if aShape.equals(aShape) ...
```

Both because the implementation of either covariant or contravariant overriding is complex and because the semantics are cloudy, almost all object-oriented languages prohibit any modification of argument types in overridden methods. To get around this restriction, programmers most often result to explicit tests and casts, as in the following C++ example:

```
boolean Triangle.equals (Shape & aShape)
{
    Triangle & right = dynamic_cast<Triangle>(aShape);
    if (right) { // it was a triangle
        // ... do triangle comparison
    } else // it was not a triangle
        return false;
}
```

It is important to point out that we have been here discussing *overriding*, where there is a polymorphic relationship between the methods in the parent and child classes. Some languages, such as C++ and Java, permit function *overloading*, wherein the programmer is allowed to write two functions using the same name, as long as the function signatures are different. The disambiguation of a function call on an overloaded method uses an entirely different mechanism from the object-oriented disambiguation of an overridden method. We will consider overloading in more detail in Chapter **??**.