```
void doIt( Base obj )
{
    obj.f( );
}
```

Since `f` is a final method, it does not matter whether `obj` actually references a `Base` or `Derived` object; the definition of `f` is invariant, so we know what `f` does. As a result, a compile-time decision, rather than a run-time decision, can be made to resolve the method call. This is known as *static binding*. Because binding is done during compilation rather than at run time, the program should run faster. Whether this is noticeable would depend on how many times we avoid making the run-time decision while executing the program.

Static binding is used when the method is invariant over the inheritance hierarchy.

A corollary to this observation is that if `f` is a trivial method, such as a single field accessor, and is declared `final`, the compiler can replace the call to `f` with its inline definition. Thus the method call will be replaced by a single line that accesses a data field, thereby saving time. If `f` is not declared `final`, then this is impossible, since `obj` could be referencing a derived class object, for which the definition of `f` could be different. Static methods have no controlling object and thus are resolved at compile time using static binding.

Static methods have no controlling object and thus are resolved at compile time using static binding.

Similar to the final method is the *final class*. The final class cannot be extended. As a result, all of its methods are automatically final methods. As an example, the `Integer` class is a final class. Notice that the fact that a class has only final methods does not imply that it is a final class. Final classes are also known as *leaf classes* because in the inheritance hierarchy, which resembles a tree, final classes are at the fringes, like leaves.

A *final class* may not be extended. A *leaf class* is a final class.

4.2.4 Overriding a Method

Methods in the base class are overridden in the derived class by simply providing a derived class method with the same signature. The derived class method must have the same return type and may not add exceptions to the throws list.

The derived class method must have the same return type and signature and may not add exceptions to the throws list.

Sometimes the derived class method wants to invoke the base class method. Typically, this is known as *partial overriding*. That is, we want to do what the base class does, plus a little more, rather than doing something entirely different. Calls to a base class method can be accomplished by using `super`. Here is an example:

Partial overriding involves calling a base class method by using `super`.

```
public class Workaholic extends Worker
{
    public void doWork( )
    {
        super.doWork( );   // Work like a Worker
        drinkCoffee( );    // Take a break
        super.doWork( );   // Work like a Worker some more
    }
}
```