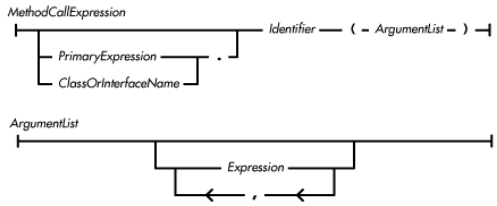
4. Expressions

Method Call Expression

A *method call expression* is a primary expression that invokes a method:



A method call expression produces the pure value returned by the method; the type of this value is specified by the return type in the method declaration. But if the method has the return type `void`, the expression does not produce a value.

The *PrimaryExpression*, if present, is evaluated first. Then expressions provided as method arguments are evaluated from left to right. Finally, the method is invoked.

When a method call is made to a method that is not `static`, the call is made through an object reference:

- If the method call expression does not contain a *PrimaryExpression* or *ClassOrInterfaceName* before the method name, the method call is made implicitly through the object referenced by the keyword `this`. This form of a method call expression is treated as if it were written:

  ```
  this.Identifier(...)
  ```

- If the method call expression contains a *PrimaryExpression* before the method name, the call is made through the object reference produced by the *PrimaryExpression*.

- If the method call expression contains a *ClassOrInterfaceName* before the method name, then the specified class must either be the same class in which the method call expression appears or a superclass of that class. In this case, the method call is made through the object referenced by the keyword `this`. This form of a method call expression is treated as if it were written:

  ```
  ((ClassOrInterfaceName)this).Identifier(...)
  ```

When a method call is made to a `static` method, the call is made through a class or interface type:

- If the method call expression does not contain a *PrimaryExpression* or *ClassOrInterfaceName* before the method name, the method call is made implicitly through the class that contains the call.

- If the method call expression contains a *PrimaryExpression* before the method name, the call is made through the class of the object reference produced by the *PrimaryExpression*.

- If the method call expression contains a *ClassOrInterfaceName* before the method name, the method call is made through the specified class or interface type.

The rules for supplying actual values for the formal parameters of a method are similar to the rules for assignment. A particular value can be specified as the actual value of a formal parameter if and only if it is assignment-compatible with the type of the formal parameter. You can use a type cast to make a value assignment compatible with a formal parameter.

The process that the Java compiler uses to select the actual method that will be invoked at runtime is rather involved. The compiler begins by finding any methods that have the specified name. If the method call has been made through an object reference, the compiler searches in the class of that object reference. If the call has been made through a specific class or interface name, the compiler searches in that class or interface. The compiler searches all of the methods defined in the particular class or interface, as well as any methods that are inherited from superclasses or super-interfaces. At this point, the compiler is searching for both `static` and non-`static` methods, since it does not know which type of method is being called.

If the compiler finds more than one method, that means the method is overloaded. Consider this example:

```
public class MyMath {
    public int square(int x) { return x*x; }
    public long square(long x) { return x*x; }
```

```
    public float square(float x) { return x*x; }
    public double square(double x) { return x*x; }
    public double hypotenuse(double x, double y) {
        return Math.sqrt(x*x + y*y);
    }
}
```

In the above example, the `square()` method is overloaded, while `hypotenuse()` is not.

If the method is overloaded, the compiler then determines which of the methods has formal parameters that are compatible with the given arguments. If more than one method is compatible with the given arguments, the method that most closely matches the given parameters is selected. If the compiler cannot select one of the methods as a better match than the others, the method selection process fails and the compiler issues an error message. Note that the return types of overloaded methods play no part in selecting which method is to be invoked.

After the compiler successfully selects the method that most closely matches the specified arguments, it knows the name and signature of the method that will be invoked at runtime. It does not, however, know for certain what class that method will come from. Although the compiler may have selected a method from `MyMath`, it is possible that a subclass of `MyMath` could define a method that has the same name and the same number and types of parameters as the selected method. In this case, the method in the subclass overrides the method in `MyMath`. The compiler cannot know about overriding methods, so it generates runtime code that dynamically selects the appropriate method.

Here are the details of the three-step method selection process:

Step One

> The method definitions are searched for methods that, taken in isolation, could be called by the method call expression. If the method call expression uses an object reference, the search takes place in the class of that object reference. If the expression uses a specific class or interface name, the search takes place in that class or interface. The search includes all of the methods defined in the particular class or interface, as well as any methods inherited from superclasses or super-interfaces. The search also includes both `static` and non-`static` methods.
>
> A method is a candidate if it meets the following criteria:
>
> - The name of the method is the same as the name specified in the method call expression.
> - The method is accessible to the method call expression, based on any access modifiers specified in the method's declaration.
> - The number of formal parameters declared for the method is the same as the number of actual arguments provided in the method call expression.
> - The data type of each actual parameter is assignment-compatible with the corresponding formal parameter.
>
> Consider the following expression that calls a method defined in the preceding example:
>
> ```
> MyMath m;
> m.square(3.4F)
> ```
>
> Here is how the Java compiler uses the above criteria to decide which method the expression actually calls:
>
> - The name `square` matches four methods defined in the `MyMath` class, so the compiler must decide which one of those methods to invoke.
> - All four methods are declared `public`, so they are all accessible to the above expression and are thus all still viable candidates.
> - The method call expression provides one argument. Since the four methods under consideration each take one argument, there are still four possible choices.
> - The method call expression is passing a `float` argument. Because a `float` value cannot be assigned to an `int` or a `long` variable, the compiler can eliminate the versions of `square()` that take these types of arguments. That still leaves two possible methods for the above expression: the version of `square()` that takes a `float` argument and the one that takes a `double` argument.

Step Two

> If more than one method meets the criteria in Step One, the compiler tries to determine if one method is a more specific match than the others. If there is no method that matches more specifically, the selection process fails and the compiler issues an error message.
>
> Given two methods, `A()` and `B()`, that are both candidates to be invoked by the same method call expression, `A()` is

more specific than `B()` if:

- The class in which the method `A()` is declared is the same class or a subclass of the class in which the method `B()` is declared.
- Each parameter of `A()` is assignment-compatible with the corresponding parameter of `B()`.

Let's go back to our previous example. We concluded by narrowing the possible methods that the expression `m.square(3.4F)` might match to the methods in `MyMath` named `square()` that take either a `float` or a `double` argument. Using the criteria of this step, we can further narrow the possibilities. These methods are declared in the same class, but the version of `square()` that takes a `float` value is more specific than the one that takes a `double` value. It is more specific because a `float` value can be assigned to a `double` variable, but a `double` value cannot be assigned to a `float` variable without a type cast.

There are some cases in which it is not possible to choose one method that is more specific than others. When this happens, the Java compiler treats the situation as an error and issues an appropriate error message.

For example, consider a situation where the compiler needs to choose between two methods declared as follows:

```
double foo(float x, double y)
double foo(double x, float y)
```

Neither method is more specific than the other. The first method is not more specific because the type of its second parameter is `double` and `double` values cannot be assigned to `float` variables. The second method is not more specific because of a similar problem with its first parameter.

Step Three

After successfully completing the previous two steps, the Java compiler knows that the expression in our example will call a method named `square()` and that the method will take one `float` argument. However, the compiler does not know if the method called at runtime will be the one defined in the `MyMath` class. It is possible that a subclass of `MyMath` could define a method that is also called `square()` and takes a single `float` argument. This method in a subclass would override the method in `MyMath`. If the variable `m` in the expression `m.square(3.4F)` refers to such a subclass, the method defined in the subclass is called instead of the one defined in `MyMath`.

The Java compiler generates code to determine at runtime which method named `square()` that takes a single `float` argument it should call. The Java compiler must always generate such runtime code for method call expressions, unless it is able to determine at compile time the exact method to be invoked at runtime.

There are four cases in which the compiler can know exactly which method is to be called at runtime:

- The method is called through an object reference, and the type of the reference is a `final` class. Since the type of the reference is a `final` class, Java does not allow any subclasses of that class to be defined. Therefore, the object reference will always refer to an object of the class declared `final`. The Java compiler knows the actual class that the reference will refer to, so it can know the actual method to be called at runtime.
- The method is invoked through an object reference, and the type of the reference is a class that defines or inherits a `final` method that has the method name, number of parameters, and types of parameters determined by the preceding steps. In this case, the compiler knows the actual method to be called at runtime because `final` methods cannot be overridden.
- The method is a `static` method. When a method is declared static, it is also implicitly declared final. Thus, the compiler can be sure that the method to be called at runtime is the one defined in or inherited by the specified class that has the method name, number of parameters, and types of parameters determined by the preceding steps.
- The compiler is able to deduce that a method is invoked through an object reference that will always refer to the same class of object at runtime. One way the compiler might deduce this is through data flow analysis.

If none of the above cases applies to a method call expression, the Java compiler must generate runtime code to determine the actual method to be invoked. The runtime selection process begins by getting the class of the object through which the method is being invoked. This class is searched for a method that has the same name and the same number and types of parameters as the method selected in Step Two. If this class does not contain such a definition, its immediate superclass is searched. If the immediate superclass does not contain an appropriate definition, its superclasses are searched, and so on up the inheritance hierarchy. This search process is called *dynamic method lookup*.

Dynamic method lookup always begins with the class of the actual object being referenced. The type of the reference being used to access the object does not influence where the search for a method begins. The one

exception to this rule occurs when the keyword `super` is used as part of the method call expression. The form of this type of method call expression is:

```
super.Identifier(...)
```

In this case, dynamic method lookup begins by searching the superclass of the class that the calling code appears in.

Now that we've gone through the entire method selection process, let's consider an example that illustrates the process:

```
class A {}
class B extends A {}
class C extends B {}
class D extends C {}
class W {
    void foo(D d) {System.out.println("C");}
}
class X extends W {
    void foo(A a) {System.out.println("A");}
    void foo(B b) {System.out.println("X.B");}
}
class Y extends X {
    void foo(B b) {System.out.println("Y.B");}
}
class Z extends Y {
    void foo(C c) {System.out.println("D");}
}
public class CallSelection {
    public static void main(String [] argv) {
        Z z = new Z();
        ((X) z).foo(new C());
    }
}
```

In the class `CallSelection`, the method `main()` contains a call to a method named `foo()`. This method is called through an object reference. Although the object refers to an instance of the class `Z`, it is treated as an instance of the class `X` because the reference is type cast to the class `X`. The process of selecting which method to call proceeds as follows:

1. The compiler finds all of the methods named `foo()` that are accessible through an object of class `X`: `foo(A)`, `foo(B)`, and `foo(D)`. However, because a reference to an object of class `C` cannot be assigned to a variable of class `D`, `foo(D)` is not a candidate to be invoked by the method call expression.

2. Now the compiler must choose one of the two remaining `foo()` methods as more specific than the other. Both methods are defined in the same class, but `foo(B)` is more specific than `foo(A)` because a reference to an object of class `B` can be assigned to a variable declared with a type of class `A`.

3. At runtime, the dynamic method lookup process finds that it has a reference to an object of class `Z`. The fact that the reference is cast to class `X` is not significant, since dynamic lookup is concerned with the class of an object, not the type of the reference used to access the object. The definition of class `Z` is searched for a method named `foo()` that takes one parameter that is a reference to an object of class `B`. No such method is found in the definition of class `Z`, so its immediate superclass, class `Y`, is searched. Such a method is found in class `Y`, so that method is invoked.

Here is another example that shows some ambiguous and erroneous method call expressions:

```
class A {}
class B extends A {}
class AmbiguousCall {
    void foo(B b, double x){}
    void foo(A a, int i){}
    void doit() {
        foo(new A(), 8);   // Matches foo(A, int)
        foo(new A(), 8.0); // Error: doesn't match anything
        foo(new B(), 8);   // Error: ambiguous, matches both
        foo(new B(), 8.0); // Matches foo(B, double)
    }
}
```