2 *Fundamental Concepts of Object-Oriented Languages*

In this chapter we review the fundamental concepts of object-oriented languages. We assume the reader has some experience with object-oriented languages, so our main purpose here is to establish consistent terminology for the rest of the text.

The concepts of object-oriented languages discussed here include objects, classes, methods, instance variables, dynamic method invocation, subclasses and inheritance, and subtypes. Other features include mechanisms to allow the programmer to refer to the current object and to access methods of its superclass. These concepts are described briefly below. In later chapters we will go into much more detail as to their meanings. For now, we also avoid discussion of most issues involving types. We will devote a substantial amount of attention to typing issues later.

2.1 Objects, classes, and object types

OBJECT
INSTANCE VARIABLE
METHOD
MESSAGE

Objects encapsulate both state and behavior. In particular, they consist of a collection of *instance variables*, representing the state of the object, and a collection of *methods*, representing the behavior that the object is capable of performing. We sometimes refer to instance variables as the *fields* of an object. The methods are routines that are capable of accessing and manipulating the values of the instance variables of the object. When a *message* is sent to an object, the corresponding method of the object is executed. (In C++, instance variables are referred to as *member fields* or variables and methods as *member functions*.)

SHARING SEMANTICS

As is the case in Java and Smalltalk, we will assume that all objects are implicitly references. This results in a *sharing semantics* for assignment. That is, if `o` and `o'` are objects of the same type, execution of the assignment state-

ment, `o := o'`, will result in `o` referring to the same object as `o'`.[1] Similarly, the equality test, `o = o'`, will be true if and only if both have the same reference (*i.e.*, both point to the same object). Also as in Java and Smalltalk, we will assume that the language implementation is provided with a garbage collector. Thus programmers do not have to worry about disposing of objects when they are no longer needed or accessible. The value `nil` is used as a null reference and is considered to be an element of all object types.

NIL

CLASS

Classes are extensible templates for creating objects, providing initial values for instance variables and the bodies for methods. All objects generated from the same class share the same methods, but contain separate copies of the instance variables. New objects can be created from a class by applying the `new` operator to the name of the class.

NEW

The following is an example of a class written in the notation to be used throughout this text.

```
class CellClass {
   x: Integer := 0;

   function get(): Integer is
   { return self.x }

   function set(nuVal: Integer): Void is
   { self.x := nuVal }

   function bump(): Void is
   { self ⇐ set(self ⇐ get()+1) }
}
```

The name of the class is `CellClass`. It has a single instance variable named `x` that holds integer values. When a new object is created by evaluating `new CellClass`, the initial value of its instance variable `x` will be 0.

The class contains three methods: `get`, `set`, and `bump`. The method `get` takes no parameters and returns an integer. The methods `set` and `bump` are procedures (a function that does not return a value), which is indicated by a return type of `Void`. The method `set` takes a single integer parameter, `nuVal`, while `bump` takes no parameters.

1. In this text we will use "`:=`" for assignment and "`=`" for the equality operator. While this differs from the conventions for the languages C, C++, and Java, we find this notation more sensible in relation to common mathematical usage.

SELF

The keyword `self` (written `this` in C++ and Java) is used in method bodies to indicate the object currently executing the method. The "dot" notation is used with `self` to get access to instance variables of the current object. Thus in the bodies of methods `get` and `set`, `self.x` refers to the instance variable `x` of the object executing the method.

Adopting notation from Smalltalk, we use the symbol "⇐" to represent sending a message to an object. While most languages don't bother to distinguish notationally between accessing an instance variable and sending a message, they are quite different operations, so we use different symbols. In the body of `bump`, the message sends `self ⇐ set` and `self ⇐ get` indicate that the corresponding methods in the current object should be executed.

In most object-oriented languages, it is possible to omit the prefix `self` when used in accessing instance variables or performing message sends. For example `CellClass` could be written:

```
class CellClass {
   x:Integer := 0;

   function get(): Integer is
   { return x }

   function set(nuVal:Integer): Void is
   { x := nuVal }

   function bump(): Void is
   { set(get()+1) }
}
```

In order to keep meanings as clear as possible, we will generally include the prefix `self` as in the first version of `CellClass`.

For simplicity, and as an aid to abstraction, we will use the default that instance variables of an object are not accessible from outside of that object's methods. We will also assume that methods are by default publicly accessible from outside of the object.

Later we will introduce notation to allow an object's methods to be hidden from other objects. Obviously we may provide access to an instance variable accessible from outside of the object by writing appropriate "get" and "set" methods that access or update the variable.

In many object-oriented languages, class names are used for three distinct purposes: as a name for the class, as a name for a constructor of the class, and as a name for the type of objects generated from the class. In order to make it easier to describe the different meanings of these three interpretations, as well as to provide better support for abstraction, we choose not to conflate these uses.

For now, we omit discussion of constructors. Instead, we will depend on classes to provide initial values of instance variables when new objects are created with the `new` operator.[2] However, we do wish to distinguish classes from the types of objects.

We consider types as abstractions that represent sets of values and the operations and relations applicable to them. In order to better support abstraction, and hence later program modification, we believe that object types should not carry implementation information. Instead they should reveal only the names and types (signatures) of the messages that may be sent to them.[3] From this point of view, classes provide too much information to users. The identity of instance variables should not be revealed to the user, nor should hidden methods. The bodies of existing methods should also not be part of the type of an object. To support this degree of abstraction, we provide a new kind of type expression to represent the types of objects.

The type of objects with public methods $\mathtt{m}_1$ of type $\mathtt{T}_1$, up to $\mathtt{m}_n$ of type $\mathtt{T}_n$, will be written as `ObjectType` $\{\mathtt{m}_1 \colon \mathtt{T}_1, \ldots, \mathtt{m}_n \colon \mathtt{T}_n\}$. For example, the type of objects generated by class `CellClass` is:

```
CellType = ObjectType { get: Void → Integer;
                        set: Integer → Void;
                        bump: Void → Void }
```

The notation $\mathtt{A} \to \mathtt{B}$ denotes the type of all functions from type A to type B. Thus a function of type $\mathtt{A} \to \mathtt{B}$ may be applied to arguments of type A and returns values of type B. We follow Java and C++ in using `Void` in the domain of a function to indicate a parameterless function and, as stated earlier, `Void` in the range is used to indicate a procedure.

Because object types do not mention instance variables, distinct classes that have the same public methods and types can generate objects with the same object types. For example, we can define

2. We will see later that we can create functions that provide the initial values of instance variables in classes. This will allow us to more closely model constructors.

3. While it is common in some programming languages to refer to method signatures as though they were distinct from types, we shall find it convenient to refer to all such notations as types.

```
class DimCellClass {
   z:Integer := -1;

   function get(): Integer is
   { return self.z + 1 }

   function set(nuVal:Integer): Void is
   { self.z := nuVal - 1 }

   function bump(): Void is
   { self ⇐ set(self ⇐ get()+1) }
}
```

Class `DimCellClass` has a different instance variable than `CellClass`, and the methods `get` and `set` have bodies that are distinct from those in `CellClass`. However, because the public methods are the same as those in `CellType`, and they have the same types as specified there, `DimCellClass` also generates objects of type `CellType`. While, in this case, objects generated from `CellClass` and `DimCellClass` exhibit identical observable behaviors, two classes can generate objects of the same type even if the methods result in different behaviors.

The following is a a simple program using these two classes. Notice that the pair `//` indicates the beginning of a comment, which extends to the end of the line.

```
program CellExample;
    ... // Definitions of CellClass, DimCellClass,
        // and CellType omitted
var
    c:CellType := nil;     // (1)
{
    c := new CellClass;    // (2)
    c ⇐ set(17);           // (3) set from CellClass
    c ⇐ bump();            // (4)
    writeln(c ⇐ get());    // (5)
    c := new DimCellClass; // (6)
    c ⇐ set(17);           // (7) set from DimCellClass
    c ⇐ bump();            // (8)
    writeln(c ⇐ get())     // (9)
}
```

The program begins with a `program` statement containing the name of the program. The next several lines should contain the definitions of `CellType`, `CellClass`, and `DimCellClass`, but we have omitted them here to avoid repeating code. Line (1) contains a declaration that the variable `c` is of type `CellType`.

The main body of a program, like that of a class or method, is enclosed in curly brackets. The `new` expression on line (2) creates a new object from class `CellClass`, which is assigned to variable `c`. As mentioned earlier, we use `:=` rather than `=` to represent assignment. Lines (3) and (4) send the messages `set(17)` and `bump()` to `c`, while line (5) prints the value of the cell obtained as the result of sending the message `get()` to `c`. Because `c` was generated by class `CellClass`, the actual code executed during these message sends is that of the corresponding methods in `CellClass`.

Line (6) results in `c` being reassigned a value created from class `DimCell-Class`, with lines (7), (8), and (9) sending the same messages as lines (3), (4), and (5). Because the object held in `c` is now generated from `DimCellClass`, the code executed during these message sends is now that of the corresponding methods in `DimCellClass`. Executing the above code will result in the number 18 being printed out twice, despite the fact that different method bodies are executed as a result of the method invocations of `set` and `bump` in the first and second half of the listing.

Because `c` is declared to have type `CellType`, it can be sent the messages `get`, `set`, and `bump`, as they are specified in the type. Objects generated by classes `CellClass` and `DimCellClass` can both be assigned to `c` because both have public methods with the signatures required by `CellType`.

DYNAMIC METHOD INVOCATION

The mechanism by which the object receiving a message is responsible for knowing which method body to execute is often called *dynamic method invocation*. This facility provides an enormous amount of flexibility for object-oriented programming. It allows a program to send messages to an object of unknown origin as long as the object has a type that guarantees it has a method with the appropriate signature. Thus objects generated by different classes may be used interchangeably and simultaneously as long as they have the same object type.

2.2 Subclasses and inheritance

SUBCLASS

One of the important features of object-oriented languages is the ability to make incremental changes to a class by creating a *subclass* (called a derived

```
class ClrCellClass inherits CellClass modifies set {

   color:ColorType := blue;

   function getColor(): ColorType is
   { return self.color }

   function set(nuVal:int): Void is
   { self.x := nuVal;
     self.color := red }
}

ClrCellType =
   ObjectType { get:Void → Integer;
                set:Integer → Void;
                bump:Void → Void;
                getColor:Void → ColorType
              }
```

Figure 2.1 `ClrCellClass` defined as a subclass of `CellClass`.

class in C++). A subclass may be defined from a class by either adding to or modifying the methods and instance variables of the original class. (We will see later that restrictions on the modification of the types of methods and instance variables in subclasses are necessary in order to preserve type safety). If class `B` is a subclass of `C`, we say that `C` is a *superclass* of `B`.

SUPERCLASS

Figure 2.1 includes an example of a subclass of `CellClass` that adds color to the objects. The *inherits* clause specifies the class from which the new class inherits, while the `modifies` clause indicates which methods from the superclass will be overridden in the subclass. Because `ClrCellClass` inherits from `CellClass`, it automatically has all of the instance variables and methods from `CellClass`, including both those defined in `CellClass` and any that were inherited. `ClrCellClass` adds a new instance variable `color` of type `ColorType` with initial value `blue`. It also adds a new method `getColor`, and *overrides* the method `set` with a new method body that not only updates `x`, but also sets `color` to be `red`. That is, the body of the method `set` contained in the class definition replaces the inherited body from `Cell-`

INHERITANCE

METHOD OVERRIDE

`Class`. We insist that names of methods that are intended to be overridden in the subclass be listed in the "`modifies`" clause as a double-check in order to ensure that methods are not accidentally overridden. Our type-checking rules will generate an error if a method is overridden, yet its name is not listed in the `modifies` clause.

Because of inheritance, objects generated from `ClrCellClass` contain the instance variable `x` declared in `CellClass` as well as the new instance variable `color`. Similarly, they contain methods `get` and `bump` as well as the new `getColor` and the redefined `set`. The type of objects generated from `ClrCellClass` is `ClrCellType`.

When new definitions are given to methods in a subclass, it is useful to be able to refer to the methods of the superclass. For instance, in overriding a method, one often wishes to apply the method body from the superclass, and then perform a few more operations before returning from the redefined method. We provide a keyword, `super`, to provide access to the methods of the superclass. Using `super`, we can rewrite the body of `set` above to be

SUPER

```
{ super ⇐ set(nuVal);
  self.color := red }
```

While this does not provide enormous savings in this case, in other cases it can. Moreover it can guarantee that any changes made later to the method body in the superclass will automatically be carried over to the subclass.

Dynamic method invocation plays an important role during inheritance. Recall that the body of the `bump` method in `CellClass` involves the message send `self ⇐ set(...)`. Inside `CellClass`, the method `set` simply updates the instance variable `x`. However, inside `ClrCellClass`, `set` also updates `color`. When `bump` is inherited in `ClrCellClass`, the message send of `self ⇐ set(...)` now results in invoking the method `set` of `ClrCellClass`. Thus sending a `bump` message to an object generated from `ClrCellClass` will end up *both* incrementing that object's `x` instance variable, and updating the value of `color` to be `red`. (In the terminology of C++, all methods are *virtual*.)

2.3 Subtypes

SUBTYPE

We say type `T` is a *subtype* of `U`, written `T <: U`, if a value of type `T` can be used in any context in which a value of type `U` is expected. That is, a value of type `T` can *masquerade* as an element of type `U` in all contexts if `T <: U`. We say `U` is a *supertype* of `T` if `T` is a subtype of `U`.

SUPERTYPE

SUBTYPE POLYMORPHISM

Because values may have multiple types in languages supporting subtyping, we say these languages support *subtype polymorphism* (we shall examine other sorts of polymorphism later). If `v` has type `T`, subtype polymorphism allows it to be used in any context that expects a value of some type `U` as long as `T` is a subtype of `U`.

Subtyping depends only on the types or interfaces of values, while inheritance depends upon their implementations. In most simple object-oriented languages, the type of objects generated by a subclass is a subtype of objects generated by the superclass. For example, `ClrCellType` $<:$ `CellType`. However, we provide examples later that show that if one enriches the language slightly, a class may inherit from another, yet the type of the objects generated by the subclass may not necessarily be a subtype of the type of the objects generated by the superclass.

It is also not necessary to restrict subtypes to those relationships that arise from subclasses. As long as object type `T` has at least all of the methods of object type `U`, and the corresponding methods have the same type,[4] then an object of type `T` can successfully masquerade as an object of type `U`. For example, the type of objects generated by `ClrCellClass` is a subtype of the type of objects generated by `DimCellClass`, even though they are not subclasses.

By separating object types from classes, we hope to make it clearer that subtyping is a relation between types, while subclassing is a relation between classes. The first has to do with public interfaces, while the second has to do with the inheritance of implementations.

STRUCTURAL SUBTYPING

We remark that we are using *structural subtyping* here. Thus subtyping is determined by the structure of the type rather than its declaration. By contrast, Java requires the programmer to explicitly specify when one type is to be a subtype (*extension* in Java terminology) of another.

In Chapter 5 we examine the subtyping relation very carefully and see that it can allow more variations in object types than we have allowed here. In Chapter 16 we will also define later another relation, *matching*, between object types that is similar to subtyping, but may be even more important in languages with a keyword for the type of `self`.

4. We will loosen this restriction later.

```
class C {
    v: T1 := ...;

    function m(p: T2): T3 is { ... }
}

class SC inherits C modifies v, m {
    v: T1' := ...;

    function m(p: T2'): T3' is { ... }
}
```

Figure 2.2 Covariant and contravariant changes in types.

2.4 Covariant and contravariant changes in types

One of the most confusing things in understanding the type systems of statically typed object-oriented languages has been understanding restrictions on changing the types of methods and instance variables inherited from superclasses when defining subclasses. Most statically typed object-oriented languages allow no changes to types in subclasses. However, as we shall see in the next chapter, this puts real restrictions on the expressiveness of the language. As a result there has been pressure on language designers to allow changes to types.

In Chapter 5 we will see why some restrictions on changing types are necessary in order to preserve type safety. For now, we simply introduce the terminology that will allow us to discuss the issue. Figure 2.2 includes the definitions of a class, `C`, and its subclass, `SC`. In the subclass we have changed the type of the instance variable `v`, and the parameter and return types of method `m`.

COVARIANT

If the types in `C` are replaced by subtypes in `SC` (*i.e.*, `T1'` $<:$ `T1`, `T2'` $<:$ `T2`, and `T3'` $<:$ `T3`), then the changes are referred to as *covariant*. This is the most obvious change to make to types in subclasses. For example, C++ now allows the programmer to make covariant changes in the return types of methods, though it allows no changes to the type of instance variables or the parameters of methods.

On the other hand, we will see later that we can preserve type safety if we allow the types of parameters of methods to be replaced by supertypes in

subclasses. Thus we may allow `T2 <: T2'`. Replacing a type by a supertype in a subclass is referred to as a *contravariant* change.

CONTRAVARIANT

The terms covariant and contravariant come from category theory. The simplest way to remember the difference is that *contra*variant changes to types are *contra*dictory to one's intuition.

We will see later that type safety is preserved in subclasses if we allow only covariant changes to the return types of parameters (`T3' <: T3`), contravariant changes to parameter types (`T2 <: T2'`), and no changes at all to types of instance variables (`T1 = T1'`).

2.5 Overloading versus overriding methods

OVERLOADED METHOD

Some languages, including both Java and C++, allow programmers to *overload* method names in classes. A method name is overloaded in a context if it is used to represent two or more distinct methods, and where the method represented by the overloaded name is determined by the type or signature of the method.

Look at the following excerpt from the definition of a class `Rectangle`.

```
class Rectangle {
    ...
    function contains(pt:Point): Boolean is
        { ... }
    function contains(x, y:Integer): Boolean is
        { ... }
}
```

The method name `contains` is listed twice. The two versions of `contains` have slightly different signatures as one takes a parameter of type `Point` while the other takes two parameters of type `Integer` that represent the coordinates of a point. It is convenient to use the same name for these methods because they represent essentially the same operation, even though the parameter types are different.

As long as their signatures are different, both Java and C++ treat methods with overloaded names as though they had completely different names. In particular, if code is written using these methods, the language processor statically determines what method body is to be executed. Examine the following code fragment:

```
var
   r: Rectangle;
   pt: Point;
   x, y: Integer;
function m(...): ... is  {
   ... r ⇐ contains(pt) ... r ⇐ contains(x, y) ...
}
```

The language processor can easily determine that the first invocation corresponds to the first definition of `contains` in `Rectangle`, while the second invocation corresponds to the second definition.

Let us compare and contrast overloaded and overridden methods. message sends involving overloaded methods are resolved statically. By contrast, overridden methods always occur in different classes, typically when one of the classes is a subclass of the other. They typically have the same signatures, though some languages allow the overriding method in the subclass to have a signature that is a subtype of the method in the superclass. Message sends involving overridden methods are resolved at run time.

Languages have different rules for when method names may be overloaded. In C++, overloaded methods must be defined in the same class, while in Java, the overloading can happen when a method in a superclass is inherited in a subclass that has a method with the same name, but different signature.

Many programmers believe that overloaded method names make it much easier to understand programs, because they allows the user to reuse names that suggest the operation being performed. For example, in the `Rectangle` class, both versions of method `contains` determined whether a particular location, represented alternatively as a `Point` or as a pair of integer coordinates, was contained in the rectangle.

Moreover, allowing overloading seems to have few consequences for a language, as the language processor automatically renames all overloaded methods with distinct names before runtime.

However, the interaction between overloaded method names with static resolution, and overridden methods with dynamic resolution can result in great confusion as to what methods are called when in a program. We illustrate this with an example.

Class `C` and subclass `SC` are defined in Figure 2.3. Class `C` has an `equals` method taking a parameter of type `CType`. Class `SC` has two `equals` methods. Thus `equals` in class `SC` is overloaded.

```
class C {
    ...

    function equals(other:CType): Boolean is
        { ... }          // equals 1
}

class SC inherits C modifies equals {
    ...

    function equals(other:CType): Boolean is
        { ... }          // equals 1

    function equals(other:SCType): Boolean is
        { ... }          // equals 2
}
```

Figure 2.3 Classes with overridden and overloaded method `equals`.

The first definition of `equals` in `SC` takes a parameter of type `CType`, overriding the `equals` method of class `C`. As a result, the comments label them both with *equals 1*.

The second `equals` method in class `SC` takes a parameter of type `SCType`. Because the parameter type is different from the other two definitions of `equals`, this method is treated as being statically different from the others. As a result, we label it with `equals 2`.

As usual, let `CType` and `SCType` be the types of objects created from those classes. Clearly `SCType` $<:$ `CType`.

Let `c` and `c'` be variables with declared type `CType`, and let `sc` be a variable with declared type `SCType`. Consider the following code:

```
c := new C;
sc := new SC;
c' := new SC;

c ⇐ equals(c);
c ⇐ equals(c');
```

```
c ⇐ equals(sc);

c' ⇐ equals(c);
c' ⇐ equals(c');
c' ⇐ equals(sc);

sc ⇐ equals(c);
sc ⇐ equals(c');
sc ⇐ equals(sc);
```

The variable `c` is assigned an object created from class `C` and `sc` is assigned an object created from class `SC`. Variable `c'` is also assigned an object created from class `SC`. This is legal because `SCType` is a subtype of `CType`.

The 9 message sends shown above correspond to all possible combinations of receiver and parameter. Which `equals` method is actually executed as a result of each of the sends? Think carefully about each case before looking at the answers in the next paragraph.

The answer to this question is rather surprising.

- All 3 message sends to `c` result in the execution of method *equals 1* from class `C`.
- All 3 message sends to `c'` result in the execution of method *equals 1* in class `SC`.
- The first two message sends to `sc` also result in the execution of method *equals 1* from class `SC`.
- Only the last message send, `sc ⇐equals(sc)`, results in the execution of method *equals 2* from class `SC`.

Most people get this wrong, even when they understand the rules for overloading given above. Usually the error is thinking that method *equals 2* is selected for some or all of the message sends to `c'`, and for two or more of the message sends to `sc`. The key to understanding which method body is selected in these examples is to remember what is resolved statically and what is resolved dynamically.

The overloading of `equals` is resolved statically. That is, the selection of *equals 1* versus *equals 2* is resolved solely on the static types of the receiver and parameters.

Because the type of both variables `c` and `c'` is `CType`, when the `equals` message is sent to `c` or `c'`, the type system examines the methods in class `C` at

compile time to determine if there is an appropriate method `equals`. There is only one method `equals` in `C`, and it has a parameter of type `CType`. That method is appropriate for each of the three actual parameters to `equals`. The actual parameters `c` and `c'` are clearly of the appropriate type. The parameter `sc` is also fine because its type, `SCType`, is a subtype of `CType`. Thus the first 6 method calls are all to *equals 1*.

The first two message sends to `sc` have parameters with static type `CType`. This is an exact match with the signature of *equals 1* in `SC`, so they resolve to that method. The last message send has a parameter of type `SCType`, so its best match is method *equals 2*.

In summary, the first 8 message sends resolve statically to method *equals 1*, while the last resolves to method *equals 2*.

Now all we have to do is figure out which of the first 8 message sends execute the body of *equals 1* from class `C`, and which execute the body from class `SC`. Because we know that all of these resolve statically to method *equals 1*, we determine which version of *equals 1* is executed by determining the class that generated the receiver of the message.

This is now easy because the receiver of the first 3 message sends is a *value* generated from class `C`. Therefore those 3 message sends result in the execution of the body of *equals 1* from class `C`. The receivers of the rest of the message sends are *values* generated from class `SC`. Hence all of those message sends result in the execution of method bodies from class SC.

Exercise 2.5.1 *Suppose that class `SC` did not include the first `equals` method – the one with parameter of type `CType` that overrode the `equals` from class `C`. Determine which of the two remaining method bodies is executed for each of the 9 message sends given above.*

*Interestingly, the answer is not the same for C++ and Java because C++ does not allow overloading methods across class boundaries. Look up the rules for each of these languages and determine the correct answer for each. (*Hint*: The only difference between the two languages with this example is that one or more message sends in C++ result in static type errors.)*

We hope that this example provides convincing evidence that the combination of static overloading and dynamic method invocation in object-oriented languages is likely to result in confusion on the part of programmers. While many programmers believe that overloading makes it easier to understand programs, we have seen very experienced programmers working on real code incorrectly believe that one method is being executed, when static res-

olution of overloading resulted in a different version of the method being selected.

We strongly recommend that object-oriented languages *not* support static overloading of method names. As a result we will not further consider static overloading in the languages discussed in this book.

2.6 Summary

In this chapter we defined the fundamental concepts of object-oriented languages, including classes, objects, methods, messages, and instance variables. We also introduced object types, which include the types of public methods, but do not include the names or types of instance variables.

We also discussed the use of inheritance in the definition of subclasses, and noted its differences from the notions of subtype and subtype polymorphism. We also discussed covariant versus contravariant changes of types that may arise when modifying object types. We shall see later that careful identification of locations in type expressions allowing covariant versus contravariant changes in types is very important in obtaining type safety in object-oriented languages.

Finally we discussed static overloading. As a result of the possible confusion in resolving static overloading and dynamic method invocation, we strongly recommend that programmers avoid using static overloading.