The Mysidia Programming Language

Preface

Mysidia is a pure object oriented language with Smalltalk's Object Model and C Family's syntax. It is a dynamically typed with optional static type hint for methods. Mysidia is a general purpose programming language that can be used in a large variety of application domains, though it has a strong focus on web development. Unlike Smalltalk, Mysidia is file based rather than image based, which makes it easy to adapt to for whoever unfamiliar or uncomfortable with image based development environment.

The first Mysidia intepreter is written in JVM, though it has been ported to C later on. In future the language may become bootstrapped once the performance is close to native C speed. Mysidia can also transpile into PHP and Javascript, as well as support for Web Assebly, making it useful for both front end and back end web development.

Mysidia's OO model follows closely to Smalltalk's barring some syntactical differences, which can be summarized in 10 rules below:

- Everything is an object
- Everything happens by sending messages
- Every object is an instance of a class
- Every class has a superclass
- Method lookup follows the inheritance chain
- Every class is an instance of a metaclass
- The metaclass hierarchy parallels the class hierarchy
- Every metaclass inherits from Class or Behavior
- Every metaclass is an instance of Metaclass
- The metaclass of Metaclass is an instance of Metaclass

However, there are some minor deviations from the original Smalltalk OO Model. For instance, Mysidia's classes are not singleton objects with their corresponding metaclasses like they are in Smalltalk. On the other hand, Mysidia expands the message system in Smalltalk by introducing the concepts of primary/secondary/tertiary messages and more. These differences will be addressed in language specs.

Below are the chapters for Mysidia's language specs:

Chapter 1: A quick tour of Mysidia

A simple Hello World Program

Mysidia is a pure object oriented language with similar object model to Smalltalk, while the syntax is vastly similar to C family languages. Below is a sample Hello World program in Mysidia:

```
namespace Demo;

class Program{
    methods: {
        main(argc, argv){
            Transcript.show("Hello World");
        }
    }
}
```

It may seem a little verbose for a simple hello world script, but it is a clear demonstration of how Mysidia programs work. Similar to C family languages such as Java and C#, Mysidia has a single entry point for its application. This entrance class is defined in a file called program.mys in Desktop Applications, or index.mys in Web Applications. The name by convention is Program(or Index for Web), but can be named anything else. Behind the scene the VM's context sends a message *.main()* or *.main(argc: intArg, argv: strArg)* to the entry class to start the application. The class' namespace is declared to be **Demo**, which will be further explore in Chapter 6 about namespaces and traits.

The entrance method is defined as **main(argc, argv)**, while **argc**(integer) and **argv**(string) are command line parameters passed to Mysidia Program. To print **Hello World** on the command prompt, we use a receiver Object called **Transcript** and pass the message **.show("Hello World")** to it. This forms an expression in Mysidia Program, and the semicolon converts this expression into a statement and ends this statement. If we run the program right now, the text **Hello World** will be displayed on the screen.

```
Hello World
```

Here **Transcript** is a built-in Class Object that is pre-defined in Mysidia's kernal. It has several worth-mentioning methods on the class itself, including the method **show(arg)** that we use in HelloWorld example. Mysidia uses message passing to a receive object which will respond to the given message by executing a corresponding method, a message consists of a selector and a few arguments.

In this example, **.show("Hello World")** is a message passed to the receiver object **Transcript**. Transcript has a method **show(arg)** that matches the message signature, thus it will find and execute this corresponding method. If such a method cannot be found, the receiver object does not understand the message passed to itself. This results in a **MessageNotUnderstoodException** thrown, it needs to be handled properly to prevent fatal error that crashes the program.

If no command line parameters are needed, and the entry class contains only a main method. It is also possible to use the below syntactic sugar to simplify the above program:

}

```
namespace Demo;

main{
    Transcript.show("Hello World");
}
```

Will prints the text *Hello World* on the Transcript as well, which saves a fair amount of boilerplate code for code demonstrations. Behind the scene Mysidia will automatically assigns the main class name to be the same as the file name(Program in this example). This syntactic sugar only works if the entry class has no other fields/methods, as well as no arguments are taken from command line. For a more complex use case, it is still recommended to define an entry class with an explicitly specified main method.

A slightly more complex Program

Of course, the power of Mysidia lies in composing larger and more sophiscated object oriented software system. The case study: Point below demonstrates a more complex and useful program that can be written with Mysidia:

```
namespace Demo;

class Point{
    fields: {
        x, y
    }

    methods: {
        init(){
            x = 0;
            y = 0;
        }

        init(x, y){
            this.x = x;
            this.y = y;
        }

        x() => x;
        y() => y;

        move(dx, dy){
            x += dx;
            y += dy;
        }

        distance(point){
            var dx, dy;
            dx = x - point.x();
            dy = y - point.y();
            return (dx.square() + dy.square()).sqrt();
        }
        toString() => "(${x}, ${y})";
    }
}

main{
    var point = Point.new();
    Transcript.show("Creating Point: " + point.toString());
    var point2 = Point.new(x: 3, y: 4);
    Transcript.show("Creating Point 2: " + point2.toString())
              ;.show("The distance between the two points is:
${point.distance(point2)}");
}
```

This example creates a two points objects and calculates the distance between these two points. Variables are declared using **var**, and multiple variables can be declared by separating each other with comma. Note that Mysidia is different from Smalltalk that the variable declaration can happen at any lines of the method body. This is because under the hood variable declaration is just sending a message to the current executing context object. We will explore this concept further in the next chapter.

The class **Point** is declared using the standard Java/C#-like syntax, though under the hood class declaration block is a syntactic sugar which translates into defining a class object with the given name, fields and methods. Inside the class block one may declare three inner blocks for **instance fields**, **instance methods** and **metaclass** block. The **Point** class has both instance fields and instance methods.

Instance fields are declared inside fields block, and separated by comma. Mysidia follows the same standard as Smalltalk and Ruby, that instance fields are always private to instances of the class and its subclasses(which is equivalent to protected fields for some languages that support explicit access modifiers). In order to access/mutate instance fields from outside, there are two ways to achieve this: 1. *define accessor/mutator methods*, 2. *use reflection library*.

Instance methods are defined inside methods block, which have **selectors**, **argument lists** and a **method bodies** to execute when the method selector matches the message's selector passed to the receiver object. In the **Point** class, two initializers are defined which take 0 or 2 arguments. Mysidia appears to support method overloading, but in fact init and init(x, y) are effectively two distinct methods. The former is stored as *init()* in method dict, while the latter is stored as *init(x:, y:)*. Initializer methods are executed automatically when an equivalent message that matches *new* is sent to the class object, which will create a new instance of the given class.

Below the initializer methods are the accessor methods to return the x and y coordinate values, note the short cut => x is equivalent to { return x; }. This is a **syntactic sugar** for methods with single line return statements. The other methods are typical behaviors for a Point object, which can respond to messages with selectors *move: dy: *, *distance:* and *toString*. Note message selectors are expressed in the form of secondary messages without arguments, which will be explained in chapter 3. These methods together form the public API for Point Class.

An interesting fact about Mysidia is that the operators are themselves just **binary messages** passed to a receiver object. The expression 2 + 3 is equivalent to to 2.+(3), which passes the message **+ 3** to integer object 2. Since 2 has a method **+ number** which can respond to this message, it is executed and returns a new integer object 5, the result of arithmetic plus operation. This beauty of consistency also incurs a cost of breaking the well-known arithmetic operation principle, as +, * and ^ are all treated as binary messages with same presidence. It is recommended to make use of parenthesis to overcome this underlying issue. The subtle difference between Mysidia and Smalltalk's message precedence will be explained in chapter 4.

Mysidia supports **Message Chaining** and **Message Cascading**. Message Chaining works the same way as most Object Oriented Languages, as one can write a.b().c().d() to chain a series of messages together. Message Cascading in Mysidia is similar to Smalltalk and Dart, which allows passing a series of messages on the same receiver object. This can be achieved by using *semicolon-dot ;.* followed by the next message, as demonstrated in the above example to pass *.show(text)* message to **Transcript** class object twice. The end result is to print two lines of text.

What to go from here?

The above two sample code demonstrates how to write very basic programs, but they are far not enough to explain everything Mysidia offers. The rest of the language specs book will explore all the features for Mysidia, ranging from the simplest objects such as variables and expressions, to complex features such as metaclasses and decorators. Mysidia is a rather heavy weight programming language compared to the simple Smalltalk language that influences its Object Model, especially with the syntactic sugars available. These are not requisite to use Mysidia as a language tool, though they aim for helping developers write more concise and elegant code.

As Mysidia language is still in early alpha stage and some of the language specs are subject to change. There may be language features and syntactic sugars added or removed before the official release, so this specs book will keep receiving updates whenever available. The first version of Mysidia will be written in JVM, though there is plan to migrate to C in future for improved performance. There is also possibility of JIT compiler to be integrated to improve performance of the language, which will be addressed in future development as well. Mysidia's support for optional static type hinting is also an experimental feature that will evolve quickly over time, we hope to give developers choices to choose the most preferable way to write programs. However, it is necessary to establish a standardized coding convention to prevent too much flexibility leading to write-only code. The one principle that will never change is: Mysidia is a pure Object Oriented Language, everything is an object, and every action happens by sending messages.

Chapter 2: Variables and Expressions

Introduction

Mysidia is a pure Object Oriented Language, so logically it makes sense to introduce the concept of objects and message passing at the very beginning. However, this proves to be a difficult task since to explain Mysidia's Object Model, it is necessary to understand concepts such as literals, variables, expressions, statements and some other very basic features. For this reason, a detailed description of objects and messages will be deferred until a brief introduction of the fundamental building blocks for Mysidia Program. Once we are done with the tour of variables and expressions, the next chapter will explain Mysidia's Object Model in detail.

Literals

Literals are the simplest building block for Mysidia's language syntax. Under the hood they are special objects that are automatically created by their corresponding classes when they are first used/referenced. The most basic literals in Mysidia are summarized in the below list:

- **null**: The literal for UndefinedObject class, it represents an uninitialized value in Mysidia.
- **true**: The literal for True class, it represents boolean true in Mysidia.
- **false**: The literal for False class, it represents boolean false in Mysidia.
- **5**: A literal of Int object, it represents an integer number in Mysidia.
- **1.6**: A literal of Float object, it prepresents a floating point number in Mysidia.
- **'a'**: A literal of character object, it represents a single unicode character in Mysidia.
- **"hello world"**: A literal of string object, it represents a string/character array in Mysidia.
- **#arg**: A literal of symbol object, it represents a special string used as identifiers for variables.
- **[1, 2, 3, 4]**: a literal of array object, it represents a fixed-size indexed collection in Mysidia.
- **["a": "apple", "b": "banana"]**: a literal for dictionary object, it represents a key-value pair collection in Mysidia.

The Literals **null**, **true** and **false** are special instances from UndefinedObject, True and False classes. They are global immutable objects created from Mysidia's virtual machine, and they are available for the entire lifecycle of Mysidia program. Since these classes are not abstract, it is theoretically possible to create other instances of UndefinedObject, True and False, though there is no practical reason for this. These literals are also reserved keywords in Mysidia, as one cannot declare variables as null, true or false.

The 5 and 1.6 are examples of Mysidia's number literals. Numbers are immutable and messages passed to them to 'mutate' them will simply create a new number object instead. Here 5 is a literal of Int object, and 1.6 is a literal for Float object. Number class provides most numeric methods that will respond to messages such as *.abs()*, *.toString()*, *.round(precision)*, *.pow(exponent)*, etc. **Int** and **Float** classes provide specialized methods for their instances to respond to messages such as *.factorial()* and *.times()* for Int objects, and ceil, floor for Float objects. If an integer literal grows larger than 2 ^ 63 - 1, the max limit for Int Object, it will be automatically converted to **BigInt** object. This behavior ensures that Mysidia does not suffer Integer overflow issues as Java and C#.

In Mysidia, single quoted text represents unicode **character** literals, the character 'a' is a good example of a character literal. It is also possible to use arbitrary unicode character such as '\u2202'. Characters have internal numerical value that can be compared from one to another, in fact both **Number** and **Character** are subclasses of the abstract **Magnitude** class. The double quoted text represents strings in Mysidia, the above list shows a sample string "Hello World". A **String** are infact a syntactic sugar for an array of characters, and thus they behave like arrays/collections. Note both characters and strings are immutable literals, a message that appears to 'mutate' a character or string, will result in creation of a new **Character** or **String** Object. The **Symbol** class is a specialized subclass of **String** used for creating identifiers for variables/slots, it only allow alphanumeric characters and must not start with a number.

Enclosed in square brackets are array or dictionary literals, as syntactic sugar for creating **Array** and **Dictionary** objects. They provide concise and fast way to instantiate the corresponding collection objects. Arrays are fixed sized numeric indexed collections starting from index 0 to the array's length - 1. This parallels the array index in C family languages, but different from Smalltalk in which array index begins at 1. Both array and dictionary literals are in fact immutable collection types, it is not possible to remove or replace elements once they are added. To create mutable versions of them, use standard object instantiation syntax with collection classes **ArrayList** and **HashMap**.

Other than these basic literal objects in Mysidia, there are other more advanced literals such as **Range** literal: *1...2*, **Closure** literal: *{ Transcript.show("Hello World"); }*, **Message** literal: *.pow(3)*, **Class** literal: *class Map* etc. Details of these complex literals will be provided in later chapters.

Variables

Variables are another fundamental building block for Mysidia's language syntax, it is near impossible to write an application of Mysidia Program without using them. Variables are special proxy objects that hold references of another object, which can be simple literal objects or other complex objects(ie. Date Object, Class, MetaClass, etc). One way to look at variables is that they are pointers to the objects they reference, though it is quite different from the idea of memory address concept in C pointers.

One of the most common types of variables are **Local Variables**(or *Temporary Variables*), which can be declared in a method or closure body. Before using a local variable, it must be declared first. The **var** syntactic sugar can be used to declare local variables. Using an undeclared local variable is a compile time error. The below example demonstrates how to declare local variables in Mysidia:

```
var a;
// declare one local variable
var b, c, d;
// declare multiple local variables
```

The first line declares a single local variable, while the second line declares a list of 3 local variables separated by comma. Variables declared with **var** are automatically assigned by default value **null**, and they can be used upon declaration. This is usually not very interesting to us, so we need to assign values to variables. In Mysidia, variable assignments work similarly in C family languages, as demonstrated in the code snippet below:

```
var a, b;
a = 10;
b = "Hello World";
```

In the above example, we declare two local variables a and b. The int literal 10 is assigned to variable a, and the string literal "Hello World" is assigned to variable b. The equal sign = is used to assign an object to a variable, so it can be referenced by the name of the variable in subsequent code execution. Variable assignment passes the message **= value** to the variable object on its left, variable objects responds to this message by activating the corresponding assignment method to set value on a variable. This conforms to the fundamental rule in Mysidia Programming Language, that everything happens by sending messages.

It is also possible to initialize the local variables right after they are declared, the above code snippet can be rewritten as below. It is recommended to declare local variables with var at the very first line of a method body, as behind the scene the VM will be able to optimize the code for maximum performance and minimum memory consumption.

```
var a = 10;
var b = "Hello World";
// or:
var a = 10, b = "Hello World";
```

Mysidia is a gradually typed language that supports **dynamically typing** by default, so variables can change to any types during the course of the application(optional static typing will be introduced in later chapters). However, Mysidia is also a **strongly typed** language, this means implicit conversion of variable types is not supported. A boolean is a boolean, a number is a number, a string is a string, and so on. On this regard, variables in Mysidia work very similarly to Smalltalk, Python and Ruby. The below snippet shows Mysidia's dynamically typed variables in action:

```
var a = 2;
Transcript.show(a);
a = "Hello World";
Transcript.show(a);
```

We can reassign a string object to variable a after it has been assigned to an integer object, a direct result of Mysidia's support for dynamic typing. If we run this program, we see these two lines on the command line prompt:

```
2
Hello World
```

It is worth noting that variable declarations using **var** in Mysidia are simply syntactic sugar for passing a message *.var(name)* to a special receiver object called *thisContext*. The details of thisContext is beyond the scope of this chapter, but this does imply that the concept of message passing is everywhere in Mysidia, even when it comes to variable declaration.

```
thisContext.var(#a);
thisContext.var(#b);.var(#c);.var(#d);
// declare local variables with thisContext implicit variable.
thisContext.var(name: #a, value: 2);
// send message .var(name:, value:) to change variable's value.
```

By passing message *.var(#a)* to a special object *thisContext*, it creates a local variable with the supplie symbol as its identifier, and initializes this variable to *null*. It is equivalent to using var syntactic sugar to declare variables. With this being said, the syntactic sugar for variable declaration is the preferred way to declare variables since it is not only more concise/elegant, but also optimized by the VM for speed improvement. Similarly sending message *.var(name: #a, value: 2)* is exactly the same as a = 2, and we can see that both variable declaration and assignment are in fact message passing under the hood.

Other than local variables, Mysidia also provides instance variables accessible by instance methods of a given class. They are declared within the field block, and separated by comma. They will be introduced in more details in *chapter 4: Classes and Methods*. In fact, the local varaiables themselves are a special kind of instance variables that belong to the lexical scope object that *thisContext* references to.

Mysidia does not provide ways to declare global variables in methods, although class objects themselves behave like global objects once declared. It is possible to find a workaround using metaclasses, but it is not recommended as global variables are proven to be bad practices. Note that in Mysidia, every variable(a local variable, an instance variable, a method/closure parameter etc) is an instance of the Slot class. The *var a* syntactic sugar as well as sending message *.var(#a)* to thisContext will both create a Slot object with the symbol #a as its identifier. This reification makes it possible to inspect and manipulate variables more conveinently, which will be explained in depth at later chapters.

Expressions and Statements

When combining a receiver object with the message passed to it, we produce a **DirectMessage** object, which is synonym to **Expression**. Mysidia's expressions can be used as arguments to form another message with its selector, which can then be sent to yet another receive object to create a more complex expression. The code snippet below shows some typical expressions for a Mysidia Program:

```
true == false
2 + 3
DateTime.now()
a = "Hello World"
```

On the first line we sends message == *false* to receiver object *true*, this produces an expression(directed message) that evaluates to false. The second line has message + *3* passed to an integer object 2, which returns the result 5. The message *.now()* is sent to DateTime class object to form another expression, which fetches a DateTime object that represents the current date/time. The last line is about assigning a value to a variable, its understandable that Variable declaration and assignments are both expressions(or directed mesages) as well. They are in fact passing a message to a receiver object(Slot class for declaration, and the slot objects for assignment).

The above examples are the simplest form of expressions, as they involve passing one message to one receiver object. It is possible to write far more complex expressions, as a series of multiple messages can be passed to the same receiver object(message cascading), or to the return value of an expression. It is also possible for arguments of a message to be an expression. The below example shows how composite messages and expressions can be produced using various techniques:

```
1 + 2 * 4 / 3
DateTime.now().timestamp()
Transcript.show("Daria");.show(" is");.show(" an");.show(" angel")
a = "Helloworld".toLower()
```

The first expression from the code above is an arithmetic operation. It passes the message + 2 to object 1, then message * 4 to the returned object 3(1 + 2), and finally / 3 to the result 12(3 * 4) from the last message passing, the expression evaluates to 4 in the end. The second expression is a standard method chaining in which message *.now()* is sent to DateTime class object, and the message *.timestamp()* is passed to the returned current datetime object to yield a integer value for current timestamp. The third expression uses method cascading to pass .show(arg) multiple times to the same Transcript receiver object. The last one assigns variable a by the result of passing message *.toLower()* to string object "Hello World".

Mysidia does not follow arithmetic precedence rule in C family language, as it is evident from the first expression. It will actually send message + *2* before sending *3 since + *2* is to the left of *3. All binary messages such as +, -, , /, %, * have same presidence, messages on the left are sent before messages on the right. In order to send messages on the right before messages on the left, we may send a higher presidence message. For instance, the last example toLower message it sent first to "Hello World", before assignment to variable a takes place. It happens because assignment messages have the lowest message presidence in Mysidia.

Alternatively, using parenthesis creates a primary directed message that will be sent at the highest presidence:

```
1 + (2 * 3) / 4
1 + (2 * 3 / 4)
(1 + 2) * (3 / 4)
```

On the first line of the above code snippet, enclosing the primary expression (2 * 3) creates a directed message that will be evaluated first. Then the expression becomes 1 + 6 / 4, and the result is the float object 1.75. On the second line, the primary expression (2 * 3 / 4) is created and evaluated to the float object 1.5, then message + 1.5 is passed to receiver object 1 to yield result 2.5. Lastly, two primary expression (1 + 2) and (3 / 4) are created and executed consecutively, which effectively becomes 3 and 0.75. Passing message * 0.75 to int object 3 leads to the final result 2.25. To avoid confusion, it is recommended to always use parenthesis to create primary expressions whenever a message needs to be past before another.

If a semicolon ; is appended at the end of an expression, the expression is converted into a statement. In fact, semicolon is yet another special unary message(called end-statement message) like assignment sent to the receiver object. And just like assignment message, the end-statement message is also at the lowest precedence. If positioned at the end of the line, it will be guaranteed to be the last message to send. Note the syntax choice for semicolon is similar to C family languages again, while in Smalltalk it is dot sign. A sample list of statements in Mysidia are shown in the code snippet below:

```
var a, b;
a = 2;
b = a + 3 * 5;
Transcript.show("The value of b is: " + b);
```

A statement is a unit of code execution, the end-statement message signals the end of message sending for the expression preceding it. Without finding a semicolon, Mysidia will continue to look for the next message to be sent(until reaching the end of a block), and the expression propagates. This can lead to unexpected behaviors that will usually cause a syntax error. For this reason, it is necessary for Mysidia Programmers to remember to terminate an expression with a semicolon(except when writing a method or closure body with single statement). Statements separated by semicolons will not affect each other, hence why a statement is considered a unit of code execution.

Summary

This chapter introduces the very basics of Mysidia's language syntax, they are the bare minimum to understand a Program written in Mysidia. With them out of the way, it will be possible to introduce the rest of language specs such as objects, classes, metaclasses, etc. The next few chapters will provide full description for the syntax and concepts for Mysidia's Object Model, we wiill start with a general picture of Objects and Messages.

Chapter 3: Objects and Messages

Introduction

The previous two chapters take readers to brief tour of Mysidia's Program and its basic syntax. From this moment on, we will introduce all the language specs in logical order. The 10 fundamental Rules for Mysidia's Object Model are specified at the beginning of the book, they apply uniformly across the entire language. Among them are the two most fundamental concept which state: everything is an Object, and everything happens by sending Messages. This chapter is an introduction of the idea, syntax and semantics of Objects and Messages in Mysidia Programming Language.

Objects

In Mysidia, everything is an object. Every object may hold its own private set of data fields, and can respond to messages passed to itself. A direct consequence of Mysidia's object model is that, the primitive values such as booleans, numbers are objects, and can respond to messages passed to them. The below code snippet shows how messages can be passed to these simple primitive objects to produce the result we need:

```
var a, b, c;
a = 2 * 3;
b = -4.abs();
b % a == 0 ifTrue: { c = "multiple" } else: { c = "not multiple" };
var d = DateTime.now();
Transcript.show("a is: " + a)
          ;.show("b is: " + b)
          ;.show("b is ${c} of a")
          ;.show("current date is: ${d.year()} - ${d.month()} - ${d.day()}");
```

The variable a is assigned by the object returned by passing message * 3 to a receiver object 2. The object 2 has a method to respond to this message, which returns the calculated value 6, another integer object. The variable b is assigned by the object returned by passing message *.abs()* to the integer object -4, which results in a new integer object 24. The next statement evaluates an expression b % a == 0 first, which results in a boolean object. The message *ifTrue: else:* is passed to this boolean object, and will assign variable c to be one of the two possible strings depending on whether *b % a == 0* is true or not. Since a is 6 and b is 24, b is clearly a multiple of a, the expression will return a true object, c will become string "a multiple". At last, the variable d is assigned as the current date time object.

Run the above code and the following will show on command line prompt, which is exactly as we expect:

```
a is 6
b is 24
b is a multiple of a
current date is: 2018-10-31
```

Deep in the implementation of the VM, there are 4 variations of objects in Mysidia:

1. **Literal Value Objects**: ie. Int Object
2. **Ordinary Reference Objects**: ie. DateTime object

3. **Indexable Collection Objects**: ie. Array object
4. **Slot Objects**: ie. Variable object

The first three kinds of objects closely mirror the Smalltalk object model. Literal value objects are stored directly on the stack in the VM, they are passed by values when used as arguments for messages. Most Mysidia objects are the second kind(Ordinary Reference Objects), these objects are created by sending a message *.new()* to their classes, and are passed by reference when used as arguments for messages. The third kind is for special collection objects like arrays, in which one can refer their elements by integer or string indexes using square bracket [] notation. Mysidia's numeric index for indexable objects begin at index 0.

The last kind(Slot objects) is an idea closely resembles the Self and Newspeak object models(it has also made its way to Pharo Smalltalk lately), which have reified variables as instances of Slot class. Variables themselves can be considered as proxies of the values they hold, developers are usually unaware of this, as messages passed to variables(slot objects) are directly sent to the actual objects they hold. The Slot objects can only be revealed by using reflection, which may be needed to design IDE/Debugger. It is also possible to create a variable by sending instantiation messages to the class Slot, details about Slots is beyond the scope of this section and will be explored further in later chapters.

As everything is an object, it is possible to imply that messages, classes, namespaces, block closures, and any other built-in elements in Mysidia. It also means that everything can be used as arguments to compose messages, and passing messages to another object will execute the code. Objects are the fundamental unit of Mysidia programs, just like cells for human body. This objects to cells analogy was originally proposed by Alan Kay, the creator of smalltalk and one of the greatest pioneers of OOP. Mysidia is designed as a pure Object Oriented Language, it will conform to Alan Kay's OO definition however the language may evolve in future.

Messages

If objects are the most important concept for Mysidia's OO Model, then the second most important will go down to messages. An object by itself is not interesting at all, it needs communicate with other objects to complete useful tasks. In Mysidia, objects 'talk' to each other by sending messages, contrast to procedural languages and impure OO languages which focus on calling functions or methods. The terminology is vital here as we do not tell an object to make procedural calls, instead we ask them to do some tasks by sending messages to them as receivers. The receiver objects will decide if they understand the message, and respond by selecting a method to execute. For this reason, Mysidia does not have concepts for functions and procedures, only objects and messages(which are actually objects themselves).

A Message consists of a selector and may optionally have arguments. A **Selector** is the identifier of a message, which has one or multiple symbols separated by colon. The symbols can be either alphanumeric or special chars, but not both. **Arguments** are objects attached to each symbol of the selector, which are optional since some messages do not accept arguments for their selectors. Once a Message is attached next to a receiver object, it forms a DirectedMessage(or Expression), and the result of the expression will be executed to return another object. This happens by activating a corresponding method that matches the selector of the message, which will be explained in more details at next chapter.

Messages in Mysidia work similarly to messages in Smalltalk, but with some differences. Mysidia has the same concept for **Unary Messages**, **Binary Messages** and **Keyword Messages**, the syntax is similar to Smalltalk as well. A unary message consists of only a selector symbol, a binary message consists of a a selector symbol and an argument which can be an object, another message, etc. A keyword message consists of a series of symbols followed by colon and arguments(here each symbol is called a keyword). In Mysidia, keywords can either be explicit specified for the each argument, or implicitly default to the names of the arguments. The first keyword can be omitted for the reason of convenience(as the method name serves as first keyword). Also non-alphanumeric message selectors can be placed adjacent to the receiver object or argument object, makes code more concise and elegant. Examples of Unary, Binary and Keyword Messages are shown in the below code snippet:

```
4 factorial
2 + 3
a == b ifTrue: { Transcript.show("equal") } ifFalse: { Transcript.show("not equal") }
```

On the first line, *4 factorial* is an example that passes unary message factorial to integer object 4, which evaluates to a new integer object 24. The second line is an example that passes binary message + 3 to integer object 2, this evaluates to a new integer object 5. Note binary mnessages are all characterized by special characters instead of alphanumeric letters. The last line is a **composite message**(message composition will be explained in next section), in which the binary messages == b is sent to receiver object a first, which returns a Boolean object. Then a keyword message is passed to the resulted Boolean object, it will print "equal" or "not equal" depending on the return value of a == b.

In additional to Smalltalk's 3 message types, there are two new types of messages that can be sent to Mysidia. One new type of Mysidia messages are **Primary Messages**, which have the highest precedence among all messages. Primary messages are created when using the dot followed by a selector identifier and parenthesis. Primary messages may take no argument, which will be just an empty parenthesis. They may also take one argument, in which keyword symbols may be omitted. Primary messages with zero or one argument look just like Java method calls. To pass more than one argument to a primary message, one may pass a series of keywords and arguments in the parenthesis as well, this mirrors C# and Python's named arguments in method calls. Note the first keyword may be safely omitted in the case of multiple arguments.

Another new type of Mysidia messages are **tertiary messages**, which are predefined messages such as variable declaration message*(var arg)*, variable assignment message*(= arg)*, and end-statement message*(;)*. Tertiary messages may be explicit(ie. assignment, end-statement messages) or implicit(send to *thisContext* object without specifiying the receiver). These messages have unique selectors that are handled specifically by the VM. The three earlier messages are collectively called **Secondary Messages**. In the following example, the first three lines are primary messages, while the last two lines are tertiary messages:

```
16.sqrt()
Transcript.show("Hello World")
Date.new(year: 2018, month: 10, day: 31)
var a
a = [1, 2, 3, 4]
```

Message Compositions

The last section explains message concept, syntax and types. Everything happens by objects sending messages to each other. Mysidia has five different types of messages, and they have varying precedence. These types of messages are listed by descending order of precedence, note again that Unary, Binary, and Keyword Messages are collectively referred to as *Secondary Messages*.

- Primary Messages
- Unary Messages
- Binary Messages
- Keyword Messages
- Tertiary Messages

However, at this moment the messages we have see are just single messages passed to receiver objects. It is possible to pass a series of messages to their corresponding receiver objects, which is referred to as Message Composition. Message composition follows three basic rules:

1. Messages of same precedence are evaluated in forwarding order of appearance, messages on the left are sent prior to messages on the right.
2. Messages of different types are sent in descending order of precedence, higher precedence messages will be sent prior to lower precedence messages.
3. Messages enclosed in parenthesis are evaluated prior to any kinds of messages, inner parenthesis are evaluated prior to outer parenthesis.

With the above message composition rules, we will be able to evaluate the results for each line on the follow code snippet:

```
DateTime.now().timeStamp()
2 + 3 * 4 / 5
6 * 5 factorial
3.cube() + 4.square()
a = "Hello " + "world".capitalize()
1 + (2 ^ (3 - 6).abs())
```

The first line is a simple message composition with a series of 2 Primary Messages *.now()* and *.timeStamp()*. Evaluation follows rule #1 from left to right as both Primary Messages have the same precedence. The first message *.now()* is passed to receiver DateTime class object, which returns a DateTime object representing the current date that we call it curDate for now. Then the second message *.timeStamp()* is passed to the returned curDate object, which further returns an integer object that represents the current timestamp value.

The second line is also a simple message composition with a series of 3 Binary Messages. Evaluation again follows rule #1 from left to right, the result is actually integer 4(2 + 3 * 4 / 5 => 5 * 4 / 5 => 20 / 5 => 4). This may seem a little odd as Mysidia does not follow the standard mathematic arithmetic operator precedence rules, in fact it does not have the concept of operators at all. The 'operator' like symbols are in fact message selectors for Binary Messages, and they all have the same precedence.

The third line is a slightly more complex message composition, as the rule #2 for different message precedence come into play. This expression consists of a Binary Message on the left, and a Unary Message on the right, but Unary Message has higher precedence. The unary message *factorial* is passed to integer object 5 first, whose result is integer object 120. Then the Binary Message * 120 is passed to integer object 6 next, the final result is 720.

The fourth line is yet another example of message composition in which the precedence rule comes into play. It consists of two Pimary Messages *.cube()* and *.square()*, and a Binary Message(Secondary Message) + *arg*. In this case, the two Primary Messages have higher precedence over the Binary Message, and the Primary Message on the left(cube) is sent before the Primary Message on the right(square). After the left Primary Message cube is sent to 3, the expression becomes 27 + 4.square(), then the right Primary Message is sent to 4, resulting the expression 27 + 16. Finally the Binary Message + *16* is sent to receiver object 27, the outcome is integer object 43.

The fifth line has Primary Message *.capitalize()*, Secondary Message + *arg*, and Tertiary Message = *arg*. The Primary Message has highest precedence, so message capitalize is sent to string object "world", and returns new string object "World". Then comes the Secondary Messsage, as Binary Message + *"World"* is sent to another string object "Hello ", which produces concatenated string object "Hello World". At last, the Tertiary Message = *"Hello World"* is passed to variable object a, which assigns new value "Hello World" to variable a.

The last line is a good example of rule #3 in action, it makes uses of parenthesis to alter the standard message precedence. Messages inside parenthesis are sent prior to messages outside of parenthesis, while parenthesis is composable so inner messages will evaluate first. It indicates that the message - *6* will be sent to integer object 3 first before anything else happens, as it is enclosed inside the innermost parenthesis. The expression then becomes *1 + (2 ^ -3.abs())*, and the outer parenthesis will evaluate now. Within the outer parenthesis, the previous 2 message composition will apply accordingly. As a result, the next message to sent is *.abs()* to receiver object -3(returns integer object 3), then message *^ 3* is sent to receiver object 2(returns integer object 8). The expression now becomes *1 + 8*, and at the very end we get integer object 9.

Message chaining and cascading:

It is a common practice to chain methods in a series, Mysidia supports Message Chaining in the same way as most OO languages. The chain happens usually from left to right, and the message on the left are sent to the first receiver object, which returns a new object. The next message is then sent to the returning object, which returns yet another new object that can accept the next message. Message chaining also follows message precedence rules, and can continue all the way until the end of file, block or statement. Message chaining is usually applied for fluent interfaces, the below code snippet shows method chaining in action:

```
DateTime.now().timeStamp();
1 + 2 * 4 / 3;
a < 0 ifTrue: { b = a.abs().cube().toString() };
```

The first line is a typical example of message chaining, in which the first message *.now()* is passed to DateTime class object, which returns a DateTime Object that represents current date time(call it curDate), then we chain the next message *.timeStamp()* to the returned object, and the result is an integer object that represents the current datetime's timestamp value. The second line is also message chaining, as it is a series of binary messages sent to the receiver objects to their left. The message + *2* is sent to 1, next the message * 4 is sent to 3(1 + 2), and then */ 3* is sent to 12(3 * 4) to give integer object at the end. It is possible to rewrite this series of message as 1.+(2).*(4)./(3), which will resemble the first message well. Though the binary messages are converted to primary messages, which may affect precedence for message sending.

The last line has a rather complex composite message, in which binary message < 0 is sent to variable a first, which returns a boolean True or False object depending on a's value. Then keyword message ifTrue is sent to this boolean object, with a closure object as argument. Within the closure body, it assigns variable b to the result of chained messages on variable a. Evaluations of expressions/statements inside closures are deferred, which will be explained in Chapter 7: Closures and Contexts. Closures are in fact literal objects that can be sent as argument in a message, or returned as value from method body. The above message that contains a closure object will only execute if passed to a boolean True object, it will be skipped if sent to boolean False object.

Message Cascading on the other hand, allows a series of messages to be sent to the same receiver object, rather than the returned object from last messages. It is a technique common used in Smalltalk, but is not available in most C family languages. Just like Message Chaining, Message Cascading can be applied in Fluent Interfaces, and proper use of Message Cascading leads to more concise, readable and elegant code. The below snippet shows a few use cases for Message Cascading in Mysidia:

```
Transcript.show("Hello");.show("World");
point.x(1);.y(2);.toString();
button.id("submitButton");.type("submit");.text("Send Email");.render();
```

The first line has two messages *.show("Hello")* and *.show("World")* both sent to receiver object Transcript, a typical example of Message Cascading. The second line uses message cascading to set the X and Y coordinate values for Point object, and then gets the string representation of this point. The last line applies Builder Design Pattern, to constructs an HTML button object with Fluent Interfaces, in which Message Cascading is applied to pass a series of messages to the button object. Both Message Chaining and Message Cascading are powerful tool to write elegant and readable Mysidia Program.

Summary

This chapter introduces the two most fundamental concepts in Mysidia's OO Model: Objects and Messages. Mysidia strictly conforms to OO purism, everything is an object, and every object communicates by sending messages to each other. Primitive literal values are objects, Variables are objects, and Messages themselves are objects too. Mysidia expands Smalltalk's message types by adding two new types of messages, with this addition every expression can be decomposed into objects receiving messages from each other, even assignment and end statement symbols are themselves messages. In the next few chapters will further explore other core language features of Mysidia Programming Languages.

Chapter 4: Classes and Methods

Introduction

With a detailed description of how objects and messages work in Mysidia, we can already build simple Mysidia Programs/Applications. However, we still need a way to allow creation of user defined objects, and specify how receiver objects will respond to messages sent to them. An object may or may not be able to understand certain type/format of messages. For this, we need to understand *Classes* and their data(fields)/behaviors(methods). Mysidia has a class-based OO Model and it is impossible to achieve anything without declaring/using Classes. The language specs for Mysidia Classes and Methods will be explained in this chapter.

Classes

The third rule for Mysidia's OO Model specifies that *Every object is an instance of a class*. A class can be viewed as a factory that creates objects, it defines the structure of the corresponding type of objects. Below is a (incomplete) list of classes for a few common objects in Mysidia:

- 2 : Int
- 3.5 : Float
- 'a' : Character
- "Hello World" : String
- [1, 2, 3] : Array
- *.toString()* : Message
- { b = 4 * 5 } : Closure

To create a class, one common way in Mysidia is to declare it with *class className*, or send message with selector *new* to a special object called **Class**. Mysidia Objects consist of **instance fields** that store reference of other objects, and **instance methods** that will respond to messages sent to them. Both Instance fields and Methods are declared inside the class declaration block. The following code snippet demonstrates how to create classes in Mysidia, along with instance fields:

```
class Fraction {

    fields: {
        numerator, denominator
    }
}

// or:
Class.new(name: #Fraction, fields: [#numerator, #denominator]);
```

In the above code we have declared a class called *Fraction*, with 2 instance fields *numerator* and *denominator*. Mysidia supports a declarative way to create *class literals*, which are in fact syntactic sugar for sending message the equivalent *new* message to object *Class* as seen from the second approach. The declarative/literal approach automatically assigns *Fraction* as a global object that can be accessed anywhere by the client code. Note we can just use Fraction instead of symbol #Fraction when using class literals.

Mysidia does not support creation of global variables in userland, although classes themselves are global variables to some extent. However, they are also very different from standard global variables in other programming languages, as developers cannot dereference a class. For instance, attempting to assign global identifier Fraction = 2, will lead to a compile-time error in Mysidia.

The **Instance Fields** are owned by each individual objects, and their values may differ for objects of the same class. A fraction object(1/2) may have different numerator/denominator from another fraction object(3/5). They are private to the owning objects and cannot be accessed by the outside code. In the above example, it is not possible to create a new instance of Fraction object and access/modify the instance fields like this:

```
den = fraction.denominator;
fraction.denominator = 1;
// Error: Accessing instance fields from outside directly is disallowed in Mysidia
```

We still need a way to access instance fields within a given object so they will be useful. In Mysidia, the instance fields of an object can only be accessed by the instance methods defined within its corresponding class or the class' subclasses(as will be discussed in next chapter for Inheritance). Instance fields may contain any other objects, which may in turn have instance fields of smaller objects. This is referred to as *Object Composition*, which are at the heart of Object Oriented Design.

The process of creating an object from a given class is called *instantiation*. For a simple Mysidia class(without specialized initializer methods), we can send the message *.new()* to the Class object directly with message passing syntax as demonstrated in the following examples:

```
var f;
f = Fraction.new();
// Or send a unary secondary message:
f = Fraction new;
```

Both the two variations are valid and are equivalent to each other in the above code snippet. The message *.new()* is sent to class Fraction, which handles instantiation of an object of Fraction. The newly created fraction objects are ready to use and can receive messages passed to them, but they will not respond to most messages we expect them to yet. To make objects understand messages and respond to them accordingly, we need to define the corresponding methods for their classes.

Methods

Instance methods are message handlers which may respond to certain types/formats of messages. A **Method Literal** may have three components: **Selectors**, **Parameters**, and a **Method Body**. The Selectors are identifiers for instance methods, they are usually alphanumeric though some special symbols are also allowed(ie. +, !, ==, ?). Numbers cannot be used as the beginning of selectors, they may however appear in the middle/end of selectors. Parameters are objects passed to the method body, they are optional. The Method Bodies are a block of code what will be executed if the corresponding methods are activated. We can declare methods inside the method block of a class, as the example for Fraction Class:

```
class Fraction {

    fields: {
        numerator, denominator
    }

    methods: {
        numerator(){
            return numerator
        }

        numerator(numerator){
            this.numerator = numerator
        }

        denominator(){
            return denominator
        }

        denominator(denominator){
            this.denominator = denominator
        }

        toFloat(){
            return numerator / denominator
        }

        toString(){
            return "${numerator.toString()} / ${denominator.toString()}"
        }
    }
}
```

Here we have defined 6 methods for the Fraction Class. Once a message is passed to a receiver object(a fraction object in this example), it will attempt to find a a method to respond to the message. If such a match is found, the object will activate the corresponding method and executes the code inside this method's body. Otherwise, it will activate a unique method called *messageNotUnderstood(message)* and throws a *MessageNotUnderstoodException* error.

Note the special variable **this** in the two setter methods, it is an implicitly declared variable that references to the current instance responding to messages. If a method body accepts arguments that has the same name as instance fields. ie. numerator and denominator in the methods *numerator(numerator)* and *denominator(denominator)*, the instance fields are **shadowed** in these methods. It means the variables will now hold reference to the argument passed to method body. In order to access the shadowed instance fields, it is possible to use the notation *this.fieldName* instead to avoid collision.

It is also possible to pass message to **this**, and another method for the same object will be activated to respond to this message. It is mandatory to explicitly specify **this** as the receiver object if a message should be sent to the same object itself. This semantics is the same as Smalltalk in which messages need to be sent to *self* explcitly, but different from Java/C# in which messages are implicitly sent to *this*. When unspecified, Mysidia's implicit receiver is actually *thisContext*, which will be explored in later chapters.

To use the class declared above, see the example code below:

```
var fraction = Fraction.new();
fraction.numerator(1);
fraction.denominator(2);
Transcript.show("Numerator: ${fraction.numerator()} , Demominator:
${fraction.denominator()}");
Transcript.show(fraction.toFloat());
Transcript.show(fraction.toString());
```

A variable fraction is declared in the first line, then the message *.new()* is passed to class Fraction to create a new instance Fraction Object. In Mysidia classes are also objects themselves which can understand certain messages sent to themselves. Once a fraction object is created, we pass two *setter* messages *.setNumerator(1)* and *.setDenominator(2)* to it to mutate its instance fields. The next line prints out the result using *getter* messages, both getters and setters are common ways to access/modify instance variables for Mysidia Objects. The last two lines converts fraction object to its float and string representation. If we run the code above, the command line prompt displays:

```
Numerator: 1, Denominator: 2
0.5
1/2
```

Which is exactly as we expect. The getter and setter messages are standard ways to ask an object for its internal instance fields, though they should be used with caution as abusing them will break **Encapsulation**, which is important concept of Object Oriented Programming. Good Object Oriented Design usually only exposes the instance fields that should be accessed by outside code, and getters/setters are used sparingly.

Return Values

In Mysidia, every method body(as well as closure body) has a return value. A return value is the resulting object from execution of a method, it may appear at anywhere inside the block. It can be specified by sending a tertiary message *return* with one argument, which is the return value. This special message is implicitly sent to the current executing context *thisContext*, the below 2 expressions are equivalent:

```
thisContext.return(a);
return a;
```

The second variant is preferred as it is concise and optimized by the VM for speed/performance, though The first variant is still useful for the consistency of message passing syntax. Many programming languages have dedicated return statement, but in Mysidia everything happens by sending messages. Here *return* is nothing more speical than a message sent to the implicitly declared variable called *thisContext*. When a return message is received by the special variable thisContext, method execution stops and the return value is sent back to the outside script in which the message was sent. More about thisContext will be explained in chapter 9.

Since *return* is a tertiary message, it will be sent after all the primary and secondary messages are sent already. The implication is that if the argument for return message consists of an expression with many other messages, they will surely be sent/evaluated before the return message is sent. For instance, in the below code snippet all expressions will be evaluated as expected. There is no need to use parenthesis to wrap up the expressions, return will be sent last:

```
return a + b;
return a * b.square();
return a > b ifTrue: { a - b } else: { b - a };
```

Similar to Smalltalk, Mysidia has no void return value/type. If a method has no explicit return value, it will implicitly *return this* at the end of method execution. It is also possible to explicitly *return this* at any point of a method body, incase the method needs to return the current object itself. Methods that *return this*, whether explictly or implicitly, will make it possible for client coders to use message chaining(fluent interface).

For methods whose body has a single line of return statement, Mysidia offers a convenient syntactic sugar with double arrow (=>) follow by an expression and semicolon. The getters and two conversion methods in Fraction Class can be rewritten as:

```
numerator() => numerator;
denominator() => denominator;
toFloat() => numerator / denominator;
toString() => "${numerator.toString()} / ${denominator.toString()}";
```

which is more concise and compact. However, it is necessary to note that trying to force only a return statement in method body may not work always. The above syntactic sugar aims at making coding in Mysidia more fun and convenient, but it should not get in the way of readability. For method with complex logic, it is still recommended to write multiple statements in a block closure, instead of the one-line double arrow expression format.

Initializers

The above class can accept setter messages to change the state of the object(manipulating the two instance fields). This is a nice feature to have, but the downside is that the object is created with invalid state, until they receive setter messages to initialize their instance fields to appropriate values. Mysidia provides special methods called **Initializers** which are activated right upon the creation of an object, they can be used to initialize objects to valid state. A typical default initializer method in Mysidia takes no argument, this default initializer for Fraction Class is shown below:

```
class Fraction {

    fields: {
        numerator, denominator
    }

    methods: {
        init(){
            numerator = 0;
            denominator = 1;
        }

        ... other methods below ...
    }
```

When an object is created by passing the message *.new()* to the Fraction class object, the instance method *init()* is activated in response to this message. In fact, the *.new()* message is first sent to the class of *Fraction*(a metaclass) which responds to this message by activating a special method *new()*. Inside the method body of *new()*, it creates an empty instance of class Fraction, and then passes message *.init()* to the new instance. If an initializer that matches the message is found, it will activate and execute initialization operation for a given object. In the above example, the *init()* method sets the numerator to 0 and denominator to 1 as default value.

Initializers can optionally accepts arguments, in which case they may be activated by sending them different messages. For instance, the below initializer accepts an argument numerator and an argument denominator:

```
init(numerator, denominator){
    this.numerator = numerator;
    this.denominator = denominator;
}
```

To create an object with this initializer, one may choose one of the following two approaches below:

```
f = Fraction.new(numerator: 3, denominator: 5);
f = Fraction new: 3 denominator: 5;
```

The first line sends a standard primary message *.new(numerator:3, denominator:5)* to Fraction Class, the secod line sends a keyword (secondary) message to Fraction Class, though the name of the first argument is not part of the selector for keyword messages and are conveniently omitted(as will be explained in the next section). Behind the scene, the class of *Fraction*(the metaclass) automatically creates method *new(numerator, denominator)* when an equivalent *init(numerator, denominator)* method is defined, details about this behavior will be discussed in later chapters about Metaclasses. It makes Mysidia's initializers convenient to use, in contract to Smalltalk's initializers which can not accept arguments.

Methods and Explicit Keyword Parameters

Methods in Mysidia usually use the name of the parameters as keywords for the selectors, but it is possible to override this behavior by explicit keyword parameters. This keyword is specified as *keyword: * before the current parameter and after the comma separating the previous parameter. Note the first parameter of a method cannot have optionak keyword. With this in mind, the above initializer method can be rewritten as:

```
init(numerator, over: denominator){
    this.numerator = numerator;
    this.denominator = denominator;
}
```

Then we can instantiate a fraction object using this explicit keyword instead:

```
f = Fraction.new(3, over: 5);
f = Fraction new: 3 over: 5;
```

By using explicit keywords, Mysidia methods can respond to more expressive messages as the one above, which can shorter and more elegant than the previous section. They work exceptionally well when the argument names are unsuitable as keywords for method/message selectors, for any possible reasons. However, abusing them can make the methods and even the corresponding messages unnecessarily verbose. It is up to the developers to pick a good balance of using explicit keyword parameters, or adhering to parameter names as implicit keywords for selectors.

Methods and Messages

In Mysidia, methods are activated by the objects after they receive messages. At the beginning, the interpreter handles message precedence rules, then converts the tokens that make up a message into a string format, which matches a *selector*. The below 4 rules apply for converting messages into selectors in Mysidia.

1. For Secondary Unary Messages, the selector is the message identifier basename.
2. For Secondary or Tertiary Messages, the selector is the message identifier basename and a colon.
3. For Secondary Keyword Messages, the selector is a list of keywords(messate identifier basename as 1st keywords and other parameter keywords) separated by colon.
4. For Primary Messages, they are converted to equivalent form of Secondary Messages, and then the above 3 rules apply.

Consider the following examples:

```
factorial
+ 3
ifTrue: { a = 1 } else: { a = 0 }
.new()
.new(4)
.new(numerator: 3, denominator: 4)
.new(3, over: 4)
```

We can apply the 4 rules above to find out the selectors for each message. The first line is a standard unary message, which follows rule #1. The second line is a standard binary message, which follows rule #2. The third line is a standard keyword message, which follows rule #3. The last 3 lines are keyword messages, so we apply rule #4 first to convert them to equivalent secondary messages. After conversion, the fourth line is equivalent to a unary message, fifth line is equivalent to a binary message, while last two lines are equivalent to a keyword message. The results for selector interpretations are shown below in the same order as they appear:

```
factorial
+
ifTrue: else:
```

```
now
new:
new: denominator:
new: over:
```

There are a few implication of Mysidia's message interpretation rules. First of all, there is no difference between primary and secondary messages for the selector format, although it affects precedence rule as primary messages are sure to be sent before secondary messages. Second, the keyword messages in Mysidia works similarly to Smalltalk and Newspeak's keyword messages, their main use cases are mostly for handling multiple closures(ie. ifTrue: trueBlock else: falseBlock) as well as creating internal Domain Specific Languages(DSL).

Also, for primary messages with multiple arguments, the first argument name is not used as keyword(the selector basename is used instead), thus can be safely omitted. It is still recommended to use them when it is not obvious what the first argument is/means, as in the case of Fraction class initializer method *init(numerator, denominator)*. Although it does not make any semantic difference, it will help readers understand the code better. Alternatively, one may specificy explicit keywords to improve readability and expressiveness.

Once the selector format is determined, it is converted to a string, and the receiver object's class will attempt to perform **Method Lookup** to find the corresponding method associated with the method selector. A detailed description for Mysidia's Method Lookup rule will be given in next chapter, but in general it is a constant time search inside a method dictionary. If a method is found, it will be activated to respond to the message, passing arguments into parameters for the corresponding methods and execute the method body. Otherwise, it will activate a method called *messageNotUnderstood(message)*, which by default produces an error.

An interesting fact is that, the arity for each method is effectively the same as the number of colons in the selectors for messages. For this reason, Mysidia does not perform arity check like many programming languages do for method call. This behavior is similar to Smalltalk's, and it significantly improves the speed of dynamic method lookup in Mysidia.

Summary

Classes are core to the OO Model of Mysidia Programming Language, from which objects are created, initialized and manipulated. Mysidia supports a standard declarative way to create classes, which is a syntactic sugar for passing message *.new()* to the object called *Class*, and in the meantime assigning classes to be a global object that can be used anywhere in the program. Classes usually consist of instance fields and methods. The instance fields cannot be directly accessed by outside code, only instance methods of the class can read or write to them. Mysidia also does not have the concept of calling methods, the methods are objects with selectors and block closures, in which the latter are executed when a matching message is sent to an object.

In Mysidia everything is an object, and that every object is an instance of a class. This also apply to classes, as classes themselves are objects, and are instances of their own classes too. The class of a Class is called a Metaclass, details of Metaclasses will be explored in Chapter 12: Metaclasses and Anonymous Classes. The next chapter, on the other hand, will introduce Inheritance and how it works in Mysidia.

Chapter 5: Inheritance and Abstract Classes

Introduction

After declaring a class in Mysidia, we can use it as a factory to create many different objects. Classes usually have many methods which act as responders to the messages passed to the objects. It happens often that certain classes are conceptually similar, and they share the same set of methods. In order for similar categories of classes to reuse these methods, we can use **Inheritance**, which is a fundamental concept of Object Oriented Programming. This chapter provides a thorough explanation of how Inheritance works in Mysidia, and what can be achieved with them.

Superclasses and Subclasses:

We define a total of 10 rules for Mysidia's OO Model at the beginning of the language spec book, from which 3 of them have been described in details before. The next rule #4 specifies that *Every class has a superclass*, which refers to inheritance. Inheritance is a key OOP concept which allows new classes to be built upon other existing classes. The classes from which new classes are based from are called **Superclasses**, while these new classes themselves are called **Subclasses**. Inheritance is an important mechanism for code reuse, as subclasses acquire data and behaviors from superclasses, in addition to their own implementations. A Superclass may provide basic behaviors that are shared and used by many subclasses. The associations of superclasses and their subclasses is called **Class Hierarchy**.

Inheritance in Mysidia can be achieved with one of the three approaches as demonstrated in the below code snippet. The first way is the standard declarative approach to create classes that inherit another class. The second way is how the first one translates into, as behind the scene Mysidia creates a class by sending message *.new(name, superclass, fields, methods)* to the receiver object Class. The third way mirrors Smalltalk's way to create subclasses from a superclass directly, which sends message *.subclass(name, fields, methods)* to the superclass object.

```
class Vehicle { }
class Car extends Vehicle { }
Class.new(name: #Car, superclass: Vehicle, fields: [], methods: []);
Vehicle.subclass(name: #Car, fields: [], methods: []);
```

The class declarative/literal approach is preferred since it automatically assigns the global object Fraction that can be used anywhere in the Program. The other 2 approaches however, allows dynamic creation of classes for which names of the classes and superclasses may not be known ahead of time, and they have use cases sometimes(ie. create a class whose name comes from a variable, or evaluation of an expression). Note the class name uses Symbol *#Car* for the message passing syntax since the class object Car does not exist yet. It is not necessary after the creation of class Car, when this class object becomes available to use.

Mysidia supports single inheritance only, a class has one and only one superclass, which prevents the classic diamond problem. In the earlier example, the class Car's superclass is Vehicle. As every class must have one superclass, the class Vehicle should have a superclass too. If no such superclass is specified explicitly, it will default to the class *Object*, which is the superclass for Vehicle as its does not inherit from a class explicitly. For this reason, the below two lines are equivalent:

```
class Vehicle { }
class Vehicle extends Object { }
```

Object is a built-in class for the Mysidia standard library, which is at the root of Mysidia class hierarchy. Every class in Mysidia either extends Object directly, or are descendents of class Object. As Object is at the top of class hierarchy, it has no superclass of its own. Therefore the rule #4 is only partially correct, the more precise description is that every class(except the root class Object) has a superclass. It is possible to inspect Mysidia's class hierarchy by sending message *superclass* to a class. With this, We can inspect superclasses for the following built-in classes in Mysidia:

- Int.superclass(): *Number*
- Number.superclass(): *Magnitude*
- Magnitude.superclass(): *Object*
- Object.superclass(): *null*
- UndefinedObject.superclass(): *Object*

As can be seen from the result above, the message *.superclass()* passed to a class object will return the superclass of the given class. The superclass of Int is **Number**, which is also has other subclasses such as Float, BigInt and Complex. The superclass of Number is **Magnitude**, which also has other subclasses such as Character and Date. The superclass of Magnitude is **Object**, the root class of all Mysidia classes.

If we send message *.superclass()* to Object, it will return **null** as it does not have a superclass. Since null is an object(special instance of the class *UndefinedObject*), it has a superclass as well, which is Object. For this reason, we may consider that class Object and UndefinedObject forms a circular relationship that they are superclasses and subclasses of each other. Such a circular relationship at the top of class hierarchy is unique, although later on we will see that Mysidia's metaclass associations also has a circular relationship at the deepest level(*MetaClass* and *MetaClass class* are classes and instances of each other).

Inheritance

Once a class inherits from its superclass, it acquires all the instance fields and methods defined in the superclass. For instance, the vehicle class has instance fields *color* and *speed*. It also has instance methods *init(speed)*, *color()*, *speed()*, *isMoving()*, *drive(speed)*, *accelerate(dSpeed)*, *stop()*, and *toString()* as shown below:

```
class Vehicle {
    fields: {
        color, speed
    }

    methods: {
        init(color, speed){
            this.color = color;
            this.speed = speed;
        }

        color() => color;
        speed() => speed;
        isMoving() => speed > 0;
        drive(speed){
            this.speed = speed
        }

        accelerate(dSpeed){
            speed += dSpeed
        }

        stop(){
            speed = 0
        }

        toString(){
            return this.isMoving() ifTrue: {
                "${color} ${this.class()} is moving at ${speed} mph"
            } else: {
                "${color} ${this.class()} is stationary"
            }
        }
    }
}
```

And now we create 2 subclasses **Car** and **Truck**, both extend from the superclass Vehicle. The class Car has a instance field *model* which describes its manufacturer. The class Truck has an instance field *load* that describes how much weight it is carrying. Note a Truck can only add more load to its carriage if it is stationary, so we force the truck to stop moving incase it is not.

```
class Car extends Vehicle {
    fields: {
        model
    }

    methods: {
        init(model, color, speed){
            this.init(color, speed);
            this.model = model;
        }

        model() => model;
    }
}

class Truck extends Vehicle {
    fields: {
        load
    }

    methods: {
        init(load, color, speed){
            this.init(color, speed);
            this.load = load;
        }

        load() => load;

        carry(dLoad){
            this.isMoving() ifTrue: { this.stop() };
            load += dLoad;
        }
    }
}
```

It is evident that methods defined in subclasses can directly access instance fields *color* and *speed*, as well as instance methods such as *isMoving()* and *stop()* of their superclasses. This is what inheritance brings on the table, which allows subclasses to 'inherit' data and behaviors from their superclasses, so they can use for their own. Since the instance methods of superclasses are acquired by subclasses, it is also possible for objects of the subclasses to respond to messages that their superclasses can understand, as demonstrated in the code example below:

```
var car = Car.new(model: "BMW", color: "white", speed: 0);
Transcript.show("Car's current speed: " + car.speed());
car.drive(50);
Transcript.show("Now driving car at: ${car.speed()} mph");
car.accelerate(15);
Transcript.show("Then accelerating car to: ${car.speed()} mph");

var truck = Truck.new(load: 5, color: "brown", speed: 0);
Transcript.show("Truck's current load: ${truck.load()} ton");
truck.drive(40);
Transcript.show("Now driving truck at: ${truck.speed()} mph");
truck.carry(3);
Transcript.show("Truck's current load: ${truck.load()} ton");
Transcript.show(car);.show(truck);
```

This program declares two variables *car* and *truck*, which are assigned with **Car** and **Truck** objects. We send messages *.drive(50)*, *.accelerate(15)* to car object, and messages *.drive(40)*, *.carry(3)* to truck object. As each message is sent to its receiver object, we prints out the car and truck's corresponding information on the command line prompt. At the very last line we send two cascaded *.show(arg)* messages to Transcript, and it will print out the car and truck's string representation. If we run the script, the following results are displayed:

```
Car's current speed: 0 mph
Now driving car at: 50 mph
Then accelerating car to: 65 mph
Truck's current load: 5 ton
Now driving truck at: 40 mph
Truck's new load: 6 ton
white Vehicle is moving at 65 mph
brown Vehicle is stationary
```

We create a car object, and then pass the messages *.drive(50)* and *.accelerate(15)*, and it is able to respond to these two messages with the correct result as we expect. It is evident that methods from superclass Vehicle are properly 'inherited' by subclass Car. A truck object can understand messages a Vehicle can understand as well, so it will be able to respond to *.drive(40)* and *.carry(3)* by activating the corresponding methods. Note the method *carry(dLoad)* has a side effect that stops the Truck from moving, so the *toString()* method will inform us that it is now stationary. We can safely conclude that, subclasses can respond to messages that their superclasses understand, in addition to their own.

Method Overriding and Lookup:

Although it is usually the intended behavior for subclasses to inherit instance methods from their superclasses, it does happen occasionally that these subclasses will need to redefine these specific methods differently from the implementations of their superclasses. This is referred to as **Method Overriding**, through which subclasses can replace the behaviors of their superclasses with their own implementation. In the above example, we may wish to redefine *toString()* methods for Car and Vehicle classes as follows:

```
class Car extends Vehicle {
   methods: {
       toString(){
            return this.isMoving() ifTrue: {
                "model ${color} ${this.class()} is moving at ${speed} mph"
            } else: {
                "model ${color} ${this.class()} is stationary"
            }
       }
   }
}

class Truck extends Vehicle {
   methods: {
       toString(){
            return this.isMoving() ifTrue: {
                "${load} ton ${color} ${this.class()} is moving at ${speed} mph"
            } else: {
                "${load} ton ${color} ${this.class()} is stationary"
            }
       }
   }
}
```

Now if we use Transcript to print the string representation of car and truck objects as defined in the last section, we get a different result instead:

```
BMW white Car is moving at 65 mph
6 ton brown Truck is stationary
```

As we can see, when the message *.toString()* is passed to Car and Truck objects, they will instead activate the corresponding *toString()* defined in the subclasses Car and Truck. This means the superclass' method has been overriden by the subclass'. Method overriding is a powerful tool for developers to redefine behaviors of superclasses, but it should be used with caution. If a subclass appears to override too many methods already defined in the superclass, it is usually a sign that class hierarchy design is inappropriate. One may want to rethink about the class design, whether the subclass really should extend from its superclass at all.

In Mysidia, when a message is sent to its receiver object, it carries out a mechanism to find the appropriate method to activate in response to this message. This process is called **Method Lookup**, from which an object is able to locate the method based on the message it receives. The rule #5 of Mysidia's OO Model specifies that: *Method lookup follows the inheritance chain*. If a matching method is defined in the object's class, it will be activated to respond to the very message received. If it cannot be found, it goes 'one level above' and search at the object's superclass. If it still cannot identify such a method, it goes yet 'another level above' at the object's super-superclass.

Eventually, it will reach the root of class hierarchy, the class **Object**(ie. the method *class()* is defined in class Object). It will try one last time to find the appropriate method, and if this process fails once again, Method Lookup stops. Instead it will activate a method *messageNotUnderstood(message)* defined in class Object, whose default behavior is to produce an error. With method overriding, it is possible for subclasses to override this method and provide their own implementation. *messageNotUnderstood(message)* is no different from other methods in Mysidia. It can be overriden by subclasses, and it also follows the standard *Method Lookup principles*.

Sometimes it is preferable for the receiver object to activate an overriden method from its superclass. Mysidia allows instance methods of child class to send messages to an implicitly declared variable called **super**. *super* is very similar to *this*, in fact both super and this refers to the current receiver object responding to messages. On this regard, using this or super as argument or return value will make no difference, and there is no practical reason to pass or return *super*.

For *super* however, the *Method Lookup* begins at the object's superclass, rather than the object's class. These *super send* messages are powerful tools for code reuse as well. With *super* in our toolbox, the *toString()* method for Car and Truck classes can be rewritten as:

```
class Car extends Vehicle {
    methods: {
        toString(){
            return "${model} ${super.toString()}"
        }
    }
}

class Truck extends Vehicle {
    methods: {
        toString(){
            return "${load} ton ${super.toString()}"
        }
    }
}
```

As we can see, making use of *super send* can significantly simplify the code we write. It happens frequently that the subclass' version of a method is an extension of the superclass', in which case the subclass' method may send message to *super* and make use of its return value. Its worth noting that we send message to *this* instead of *super* inside the subclass' initializer, this is because methods with different arguments in Mysidia are considered different methods. For this reason, there is no need to perform *super send* as the subclass' initializer does not really override superclass' initializer unless they share the same arguments.

Both *this* and *super* are specially reserved words like *null*, *true* and *false*. We cannot declare local variable named this and super, nor can we re-assign these variables to different values(they are immutable values within methods of a given class). Another reserved word that cannot be used as local variable names is *thisContext*, which we've seen before and will explore more details in later chapters.

Abstract Classes and Methods

Sometimes a class exists for the sole purpose of being inherited. It usually applies to classes whose concept are rather abstract and hard to visualize, thus we should not create instances of these classes, only their subclasses. They are called **Abstract Classes**, and their implementation is incomplete, the responsibility of implementing the missing features/behaviors is on the subclasses that extend from the abstract superclasses.

In the Vehicle class hierarchy from previous sections, the class Vehicle is a good candidate for an Abstract Class. Unlike Smalltalk and many other dynamically typed programming languages, Mysidia has a special way to create such Abstract Classes. We can declare a class as abstract with the following three approaches:

```
abstract class Vehicle { }
Class.new(name: #Vehicle, superclass: Object, fields: [], methods: [], abstract: true);
Object.subclass(name: #Vehicle, fields:[], methods: [], abstract: true);
```

Again the first declarative approach(using class literal) is preferred for the same reason explained above. The other two ways follow standard message passing syntax, and they are good for creating classes dynamically from run-time. Now if we attempt to create an object of class Vehicle, we will get an error message:

```
var vehicle = Vehicle.new(color: "blue", speed: 30);
// Runtime Error: Cannot create instances for Abstract Class Vehicle
```

Since *Abstract Class* are themselves incomplete implementations, they may contain methods with incomplete implementations as well. These methods are referred to as **Abstract Methods**, they cannot be used to respond to messages, and instead concrete implementations needs to be provided by subclasses extending from the abstract superclasses. Mysidia also has a special way to define Abstract Methods, though only Abstract Classes may contain Abstract Methods. In the below example we create an abstract method *canDrive(speed)* and redefine method *drive(speed)* to check for it first:

```
abstract class Vehicle {
    methods: {
        abstract canDrive(speed);
        drive(speed){
            this.canDrive(speed) ifTrue: { this.speed = speed }
        }
    }
}
```

Now vehicles will need to check if it is possible to drive with the given speed, as well as other criteria that may be important. A car is drivable so long as the speed is not faster than the car's maximum allowed speed(80 mph, for instance). For a truck however, it may need to check if the load is too heavy, as an overloaded truck(heavier than 30 ton, for instance) may not be able to move. We shall implement the method *canDrive(speed)* in the two subclasses like the code snippet shown below:

```
class Car extends Vehicle {
    methods: {
        canDrive(speed) => speed <= 80;
    }
}

class Truck extends Vehicle {
    methods: {
        canDrive(speed) => (speed <= 60) && (load <= 30);
    }
}
```

With this change, the subclasses of Vehicle will do speed check(and load check too for truck) when their instances receive message *.drive(speed)*. Inside the method *drive(speed)*, a message *.canDrive(speed)* is sent to *this* and the corresponding methods canDrive(speed) is activated in response to this message. If a subclass does not override *canDrive(speed)*, it will attempt to use the abstract superclass method, which produces an error as the method is not implemented. It is necessary for subclasses to implement all abstract methods from their superclasses to be complete and useful.

The Mysidia Standard Library make uses of *Abstract Classes* and *Abstract Methods* heavily as part of the class diagram/API. The classes **Magnitude** and **Number** are both abstract classes, and they have a few abstract methods implemented by their subclasses. The abstract class *Magnitude* has abstract methods responding to comparison messages such as >, >=, ==, max(arg), min(arg), implemented by their concrete subclasses(ie. Character and DateTime). It is possible for an abstract class to inherit from another abstract class, as we see *Number* extends from *Magnitude*. In this case, the implementation of abstract methods is not necessary, although concrete subclasses will need to implement abstract methods defined in both the abstract class and its abstract superclasses.

Summary

Inheritance is an important OOP concept, which allows classes to extend from superclasses to reuse common data and behaviors. Inheritance usually establishes is-a relationship between superclasses and subclasses, as subclasses are concrete and specialized version of their superclasses. In this chapter we have introduced 2 more fundamental rules for Mysidia's OO Model: *Every class has a superclass*, and *Method lookup follows the inheritance chain*. With this, we have effectively covered half of the total 10 rules. The rest of them will require solid understanding of Metaclasses, which will be discussed in details in *Chapter 12: Metaclasses and Anonymous Classes*. Before we get down to the complex magic of Metaclasses, we will visit two other features Mysidia offers, such as Namespaces and Traits in the upcoming chapter.

Chapter 6: Namespaces and Traits

Introduction

The last few chapters have described the fundamental concepts of Mysidia Programming Language. Knowing how Objects, Messages and Classses work is sufficient for us to write a simple application in Mysidia. In order to use them though, it is necessary to learn about **Namespaces** and how it works. Mysidia has language support for **Traits**, which will be explained as well. A solid understanding of these more advanced OO concepts will be vital to write powerful Mysidia code.

Namespaces

Namespaces are declarative regions that provide scopes for a set of identifiers such as objects and classes. In Mysidia, classes are global objects accessible from anywhere in the program, it happens frequently that names need to be reused for different classes. For instance, *Event* can be a valid class name in many different context. Namespaces can help with organize code into logical groups and prevent name collisions, without which we will have to use prefixes and other suboptimal techniques to name and organize our classes/code.

As Mysidia programs are file based, it is necessary to specify a namespace on top of each file. We can achieve this by sending secondary keyword message *current: namespace-identifier declare: block* to the class object Namespace. It also has a syntactic sugar version that looks similar to C#'s namespace declaration. The two approaches below are both valid:

```
Namespace current: #Demo declare: {
    class Point { }
};
// or:
namespace Demo;
class Point { }
```

As everything is an object in Mysidia, this rule also applies to namespaces. Declaring namespaces is effectively sending messages to the *Namespace* class. There is a minor difference between the two variants. The first approach enables classes in a file to be declared in different namespaces, while the second syntactic sugar approach allows classes to be declared in one and only one namespace declared at the top of a file. The first variant may be good for testing and quick prototyping, it is in general not a good practice to declare more than one namespace in one file. For this reason, the second variant is preferred for developing full fledged applications, and hence what we will be using throughout the rest of the language specs.

Following namespace declaration, everything defined within the same file will be inside the scope of this namespace. At this point, identifiers such as classes are prefixed by *namespace-name.*, thus class *Point* becomes *Demo.Point*. Classes and other objects defined within the same namespace do not need to use this prefix. For instance, another class *Fraction* defined inside namespace *Demo* can refer this class as simply *Point* instead of *Demo.Point*. However, anything outside of the namespace will have to refer to this class as *Demo.Fraction*.

It is also possible to declare nested namespaces in Mysidia, it starts with a top namespace-name, followed by any number of dots(.) and subnamespace-names. Below is a code snippet that shows 1-3 levels of nested namespaces:

```
// nested 1 level
namespace Demo.Sub;
// nested 2 level
namespace Demo.Sub.Sub2;
// nested 3 level
namespace Demo.Sub.Sub2.Sub3;
```

If a class *Point* is defined in each of the 3 namespaces above, it can be identified as *Demo.Sub.Point*, *Demo.Sub.Sub2.Point* or *Demo.Sub.Sub2.Sub3.Point* respectively. This long version of class identifier is called a **Fully-Qualified Name**, while the class identifier without the namespace prefix is called a **Shortened Name**. If we attempt to print out an object's class by sending message *class* to this object, what we get is actually the *fully-qualified name* of the class. It is also possible to get the class' *shortened name* and *namespace* by sending the appropriate messages to the returned class object. Assuming class Fraction is declared in namespace Demo.Sub, and we run the following code snippet:

```
namespace Demo.Sub;
// we are in the same namespace as Fraction Class

var fraction = Fraction.new();
Transcript.show(fraction.class());
Transcript.show(fraction.class().shortName());
Transcript.show(fraction.class().fullName());
Transcript.show(fraction.class().namespace());
```

And we will see this on the command line prompt:

```
Demo.Sub.Fraction
Fraction
Demo.Sub.Fraction
Demo.Sub
```

Mysidia's core library classes are all organized in namespaces, usually 2-3 levels deep. The top namespace is *Mysidia*, classes defined in Mysidia namespaces are created and maintained by Mysidia's Core Team. For Mysidia's standard library classes, the first subnamespace is *Standard*, which comes bundled with the default installation/package of Mysidia. The second subnamespace is where we begin seeing classes inside, some widely used subnamespaces include *Lang*, *Utility*, *Collection*, etc. The third subnamespace may contain highly specialized classes used for more complex programs. The chapter 8-10 will introduce some of the standard library classes in details.

Importing and Aliasing

Sometimes it becomes tedious to write out the fully qualified names for classes, especially if the classes have deeply nested namespace. Also it happens frequently that we need to reference the a given class many times inside another class/method defined in a different namespace. Fortunately, Mysidia provides a way to *import namespaces* so we can type much less than we otherwise have to for a class identifier. This can be achieved by sending message *using:arg* to Namespace Object:

```
Namespace current: #Demo.Math.Algebra
          using: [Demo.Math.Calculus.Integral, Demo.Math.Geometry.Point]
          declare: {
    class Fraction { }
}
// or:
namespace Demo.Math.Algebra;
using Demo.Math.Calculus.Integral;
using Demo.Math.Geometry.Point;
class Fraction { }
```

Both approaches from the above example code will work, though again the second declarative variant is preferred for the same reason mentioned in last section, as well as the fact that it is more elegant and compact when importing multiple namespaces. Though not strictly required, it is generally recommended to declare only one class in a Mysidia file. Mysidia has a default class loader that will load class files according to its filenames and directories. Details of Mysidia's class loading mechanism will be explored in later chapters.

Note we can import a subnamespace instead of a specific class. If we simply write using *Demo.Math* or *using Demo.Math.Geometry*, we can refer to the class Point by *Math.Geometry.Point* or *Geometry.Point*. This is useful when we need to use a lot of classes inside one namespace and it becomes too tedious to import all of them one by one. Alternatively, we can use wildcard import *, which works similarly to Java and Python's. It import all classes and other global objects declared in the subnamespace preceding the wild card symbol *. Developers should use wildcard import sparingly, abusing it can result in unmaintainable code.

```
// import one level
using Demo.Math;
var point = Math.Geometry.Point.new();
// import 2 levels
using Demo.Math.Geometry;
var point = Geometry.Point.new();
// wildcard import
using Demo.Math.Geometry.*;
var point = Point.new();
```

Occasionally we need to import 2 or more classes with the same shortened name in different namespaces, or an imported class has name collision with the class created in a given file. To avoid name collision, we can use fully qualified name for a class, or import with their subnamespace as described above. Another way to resolve this issue is to use *alias*, which allows importing a class or subnamespace as a different identifier. In Mysidia, **namespace aliasing** can be achived with the following two ways:

```
Namespace current: Demo.Math.Algebra
          using: [#GPoint: Demo.Math.Geometry.Point]
          declare: {
    class Fraction { }
    class Point { }
}
// or:
namespace Demo.Math.Algebra;
```

```
using Demo.Math.Geometry.Point as GPoint;
class Fraction { }
class Point { }
```

The above example assigns symbol *#GPoint* as alias to class Demo.Math.Geometry.Point, and it will then be possible to refer to this class as its alias. As we can see, we will be able to create another Point class in the namespace Demo.Math.Algebra as the name collision is no longer an issue. The alias stays valid for code written inside the block closure in the first variant, or through the end of file for the second variant. Namespace importing and aliasing are helpful tools to simplify the code we write in Mysidia, which will both come in handy in large application development.

Traits

We have learned from the OO Model that Mysidia only supports single inheritance, a class can have one and only one superclass. This solves the traditional diamond problem that can arise from multiple inheritance, but also limits flexibility. There are times when the effect of multiple inheritance is desirable, in which classes can reuse methods defined in more than just their superclasses. Fortunately, Mysidia has **Traits** which are collections of methods and can be included/used by classes and objects for code reuse.

Similar to the declaration of Namespaces, Traits can be defined by both the standard and declarative(syntacic sugar for trait literal) approaches. Both are equivalent in general, though the declarative/literal approach is preferred, unless the trait name comes from a variable(in this case, only the first variant will work).

```
Trait.new(name: #TFlyable, methods: [
    fly(){
        Transcript.show("It is flying.")
    }
]);

// or:
trait TFlyable {
    fly(){
        Transcript.show("It is flying.");
    }
}

trait TSwimmable {
    swim(){
        Transcript.show("It is swimming.");
    }
}
```

As we can see from the example above, trait definition is in fact sending messages to the class Trait. Sending message *.new(traitname)* creates an empty trait with the specified name, it also declares the trait to be a global object. Mysidia's coding standard recommends that trait names are prefixed with letter T, but this is not a strict requirement. In order to write methods inside this trait, we send message *.new(name: traitname, methods: methodlist)* and the methods can be defined inside the trait block as they are defined in methods block of classes. Traits are first class objects in Mysidia, and they can be assigned to variables and passed as arguments in a message as well.

Traits are defined inside the enclosing namespace, and like classes they can be referenced by the fully qualified name. In the above example, assuming namespace *Demo.Animal*, we can reference the trait by *Demo.Animal.TFlyable*. Traits can be imported in the same way as classes by sending message *using traitname* to the implicit Namespace object, and developers can then refer to them by their shortened name(in this case TFlyable). Aliasing also works for traits, as it does for classes:

```
namespace Demo.Animal;
using Demo.Animal.TFlyable;
//or aliasing if needed:
using Demo.Animal.TFlyable as TAFlyable;
```

Applying Traits

Now that we are able to create Trait objects, the next step is to use them in our program. To use a Trait, we can send message *.new(name: classnamesymbol, superclass: superclassname, traits: traitnames)* to the class object, the traitnames argument is an array of trait objects. Alternatively, we can use the syntactic sugar version to append *with* expression to the class or trait literal, before the block that defines methods, as shown in the code snipper below:

```
Class.new(name: #Butterfly, superclass: Insect, traits: [TFlyable])
class Bat extends Mammal with TFlyable { }
trait TBird with TFlyable { }
```

As we see, the trait *TFlyable* is applied on two classes and one trait, allowing them to reuse the methods defined in Trait TFlyable. In this case, every instance of the classes that apply trait TFlyable will be able to respond to messages that Trait TFlyable can respond to. It is also possible to send it to each instance of a class to apply traits dynamically:

```
var flyingSquirrel, swimmingCat;
flyingSquirrel = Squirrel.with(TFlyable).new();
swimmingCat = Cat.with(TSwimmable).new();
```

This way we can create objects of the same class but with different traits similar to Scala's Trait implementation, it is flexible and can come in handy sometimes. Behind the scene Mysidia creates an *Anonymous Class* which inherits from the named class as well as the supplied trait objects. More details about Anonymous classes will be described in later chapters together with the concept of MetaClass.

The Receiver Object can apply more than one trait, this can be done by sending message *with traitname* multiple times, or separate the trait names by comma as syntactic sugar for the declarative application of traits:

```
Class.new(name: #Whale, superclass: Mammal, traits: [TSwimmable, TLungBreathable])
class Puffin extends Bird with TFlyable, TSwimmable, TDiving { }
var superHuman = Human.with([TFlyable, TSwimmable]).new();
```

Mysidia's traits are composable units which can contain only methods, this means that it is not possible to define instance fields in traits. For this reason, the methods can be declared directly inside the trait literal block instead of a sub-block for methods(as in class literal definition). This behavior is consistent with Traits in Smalltalks, but is very different from Scala's traits.

Method Lookup with Traits

Upon receiving a message, the object will look up the inheritance chain to find the appropriate method to activate. It follows the same pattern as most OO languages, that methods in the subclasses will be activated to respond to messages, overriding the behaviors of superclasses. The logic becomes more sophiscated as Traits are added to the mix, in general the pattern is subclass methods -> trait methods -> superclass methods. Method lookup starts at the subclass, then the traits it uses, and finally the superclasses it inherits. Consider the following example:

```
class Staff {
    methods: {
        greet() => "Hello, I am a staff.";
        work() => "I am going to work until 6pm.";
    }
}

trait TRecruiter {
    work() => "I am going to work until 9pm.";
    hire() => "I am hiring a new staff for anyone.";
}

class Manager extends Staff with TRecruiter {
    methods: {
        manage() => "I am managing our staff.";
        hire() => "I am hiring a new staff for my company.";
    }
}

var person = Manager.new();
Transcript.show(person.manage());
Transcript.show(person.greet());
Transcript.show(person.work());
Transcript.show(person.hire());
```

The above code snippet has a class Manager whose superclass is Staff and uses trait TRecruiter. At the code we display the results of sending 4 messages to the manager object, and it shows:

```
I am managing our staff.
Hello, I am a staff.
I am going to work until 9pm.
I am hiring a new staff for my company.
```

The manager object first receives message *manage* and activates the corresponding method in the subclass, and then receives message *greet* and responds with the method in superclass. These behaviors are consistent with the method lookup pattern in the chapter of inheritance. Then it receives message *work*, the corresponding method is defined in both superclass Staff and trait TRecruiter. The result shows that the method in trait TRecruiter is activated instead of the method in superclass, thus the method *work* in trait overrides the same method in superclass. At last, it receives message *hire* which is defined in both subclass Manager and trait TRecruiter. The result shows that the method in subclass Manager is activated instead of the method in TRecruiter, thus the method *hire()* in subclass overrides the same method in trait. Method Lookup starts at subclass, then trait, and superclass, which is to be expected.

In Mysidia it is possible to use multiple traits for a given class. Although this enhances the power of code reuse, it raises a concern when the traits share same methods, similar to diamond problem for multiple inheritance. In fact, a fatal error will happen when a class uses traits with same names, and the program will not compile. There is an exception to the rule when the method name collision happens between a class trait and instance trait(trait used by a non-class object).

```
trait TSeniorRecruiter {
    hire() => "I am hiring a lot of staff for my client.";
}

var manager = Manager.with(TSeniorRecruiter).new();
Transcript.show(manager.hire());
```

This will print the text *I am hiring a lot of staff for my client.* on the screen, thus the trait TSeniorRecruiter's method *hire()* is activated instead of the trait TRecruiter's. In fact, methods from the instance traits will be activated even before the methods in the class. With this in mind, we can update the method lookup order as: Instance Traits -> Classes -> Class Traits -> Superclasses. This behavior is a direct result of creating anonymous classes behind the scene, which will be discussed in more details at chapter 12 about MetaClasses and Anonymous Classes.

Traits and Abstract Methods

It is possible for traits to contain abstract methods(or a combination of abstract and concrete methods). In this case, classes using the traits must also implement all these abstract methods. Below is a trait object with an abstract method, and a class that implements this trait:

```
trait TDrawable {
    abstract draw();
}

class Rectangle with TDrawable {
    fields: {
        length, width
    }

    methods: {
        init(length, width){
            this.length = length;
            this.width = width;
        }

        draw() => "Drawing Rectangle...";
        toString() => "Rectangle with length: ${length}, width: ${width}";
    }
}
```

If a class(or object) uses the above trait *TDrawable*, it will have to implement the method *draw()*, unless it has already been implemented by its superclass. If violated, the compiler will complain with an fatal error that halts the program execution. Traits with abstract methods are common in writing large and complex object oriented software, and they are quite similar to interfaces with default methods in Java and C#. Both classes and traits are considered type objects that can be used for typehinting, which will be explained in detail in chapter 13: Type System and Type Hinting.

Summary

Namespaces and Traits are complementary to the classic OO model in Mysidia. Namespaces are designed for organizing classes and resolving name collision, they can be highly useful when working with large teams or third party libraries in which classes may easily share the same names. Traits on the other hand, provides a set of methods that can be reused in the classes. They are both powerful tools to enhance application development, making lives easier and more convenient for the programmers. Solid understanding of how to use namespaces and traits properly is vital to writing elegant and reusable Mysidia program. The next chapter will continue to explore Mysidia's core language features, namely closures and contexts.

Chapter 7: Closures and Contexts

Introduction

Closures are block of code with deferred execution, this concept is very similar to Smalltalk's BlockClosures. Although Mysidia does not have the concept of Function being a pure OO language, Closures can be considered as Anonymous Functions in other programming languages. They are used in writing complex and powerful code in Mysidia, as already seen in the previous chapters when the messages such as *ifTrue: trueblock else: falseblock*, *do: closure* are passed to the corresponding objects with Closures as arguments. Alongside with Closures are the concepts of **Contexts**, which represents the state of method activation and determines the visibilities for local variables. Both of these will be explained in details in the next few sections of this chapter.

Closures

Closures are essential in Mysidia Programming Language, as it does not have support for procedural control structures, and these tasks must be performed by sending a message to certain receiver objects(boolean objects for conditional, number or collection objects for iteration). The arguments passed with the messages are usually closure objects, as we have seen in previous chapters. Closures are instances of the class Closure, which contains portions of executable code that can be evaluated at later time. The body of a closure is enclosed in a pair of curly brackets, which can be a single statement or multiple statements. In the latter case, the statement termination message(;) needs to be send to separate statements(but can be omitted for the last statement).

```
var a = 0;
a == 0 ifTrue: { Transcript.show("Hello World") };
// closure with single line body.
5.times() do: (i){
    a = i + 1;
    Transcript.show("the value of a is: " + a);
    a += 1;
}
// closure with multiple lines body.
```

A closure body is similar to a method body that it can define own local variables inside. This is usually useful for a complex multi-statement closure, and local variables of the closure can be defined with the syntactic sugar var on the very top of the closure body(aka sending tertiary message *var varname* to *thisContext*). See the example below for declaring/using local variables in closures:

```
var x = 5;
var a = ArrayList.new(x);
x.times() do: (i){
    var y = i - 2;
    y > 0 ifTrue: { a.add(i) };
}
Transcript.show(a);
Transcript.show(y);
```

This will print 3 4 5 on the screen, as expected from the logic inside the closure body. It will not print y however, as the variable y is local to the closure but is nonexistent outside. This phenomenon shall be explained in the later section about Method/Closure Context.

Closure Evaluation and Assignment

As Closures themselves are objects, they can receive and respond to messages. To evaluate the block of code in closures, we can send message *.value() .value(arg)* or *.values(args)*(array of arguments) to a closure. Since this is a rather common operation, Mysidia provides a syntactic sugar to evaluate a closure by ommiting the *.value* or *.values* portion of the messages. The below code snippet shows an example of how block closures can be invoked:

```
{ Transcript.show("Hello World") }.value();
{ Transcript.show("How are you doing?") }();
(x){ Transcript.show(x * 2) }.value(5);
(x){ Transcript.show(x ^ 2) }(5);
(x, y){ Transcript.show(x * y) }.values([2, 3]);
(x, y, z){ Transcript.show(x * y - z) }(2, 3, 5);
```

If we run the program with command prompt, the transcript will print: Hello World, How are you doing?, 10, 25, 6 and 1 on the screen. Note the syntactic sugar () or (arg) is equivalent to message *.value()* or *.value(arg)*, while the syntactic sugar (arg1, arg2, ... , argn) is equivalent to message .values([arg1, arg2, ... , argn]). Since the syntactic sugar is concise and elegant, it will be used throughout the rest of the language specs.

Since closures are first class objects in Mysidia, it is possible to assign them to local variables of the enclosing method, as well as passing them as argument when sending a message to a given object. This enables deferred execution of the code inside the closure body, by various means. The below code snippet demonstrates more advanced usage of closures:

```
var a = { Transcript.show("Hello World") };
var b = { Transcript.show("My Pleasure!!") };
b();
5.times().do(a);
```

The above program will print *My Pleasure!!* first, and then *Hello World* five times. The two closures are assigned to variable a and b, the latter is evaluate first while the former is then passed as argument with message *.do(a)* to the number iterator object returned from 5.times(). This implies that closures are no different from any other objects discussed in the previous chapters, they can be assigned and passed at will.

Closures as Arguments and Return Values

The fact that closures are objects also implies that they can be used as argument for sending messages, or as return value from method body(or another closure body in nested closures). We've already seen how closures can be passed as argument in a message sent to a receiver object in the last few chapters about conditional(ifTrue: trueblock, ifFalse: falseblock, etc) and the iteration messages as we will see in next chapter. We can write nested closures by passing another closure as argument, the following code emulates a nested if statement from other languages.

```
var x = 0, y = 1;
x == 0 ifTrue: {
    Transcript.show("The value of x is zero");
    y == 0 ifTrue: {
        var z;
        z = x * y;
        Transcript.show("All values of y, z are also zero.");
    }
}
```

The above code will execute the outer block but will skip the inner block(x is 0 but y is not 0), as expected. It is also quite often for methods to return closures that can be used by the client code to use this value, consider the following example:

```
class RepeatablePrinter {
    methods: {
        print(num){
            var text = "Hello World";
            return (name){
                num.times() do: (i){ Transcript.show(text + name) }
            };
        }
    }
}

main{
    var printer, operation;
    printer = RepeatablePrinter.new();
    operation = printer.print(5);
    operation("Joe Doe");
}
```

This code above will print the text "Hello World Joe Doe" 5 times on the screen, the *print(num)* method from class RepeatablePrinter returns a closure which can accept argument to concatenate the string printed by the Transcript. Although Mysidia does not provide language support for functions being a pure object oriented language, the syntactic sugar for evaluating closures with parenthesis *()* feels like calling a function as how it works in other programming languages.

Local and Nonlocal Returns

Similar to methods, closures also have their return values. Returning a value from a closure is referred to as **Local Return**, Mysidia's closures will always return the value evaluated from the last line inside the closure body. Note it is necessary to make sure not to use a semicolon(;) at the end of evaluating the expression. This is because the return value of sending the end statement(;) message is always null, though this works well for closures whose return values are unimportant(ie. closures for message selector *ifTrue: else:*). See the below code snippet for how to achieve local returns from closures:

```
var a, closure, closure2;
closure = (x, y){ x + y };
closure2 = (x){
    x += 1;
    x.square() //must not have semicolon
}
a = closure(2, 3);
Transcript.show(closure2(a));
// prints 36 on the screen, since expression 6.square() is 36
```

When an explicit return message is sent in the block closure body however, the execution does not return from the closure itself, but from the defining method. This is called **Nonlocal Return**(or Block Return), as the method execution immediately halts regardless of how deeply nested the closure is. Nonlocal Returns can be used as early return point from a method body, as demonstrated from the following examples:

```
var a;
a ifNull: {
    a = 1;
    return a;
};
Transcript.show(a);
// it will not print anything on the screen, as the value a(1) is returned outside of
the method body.
```

It is common to use nonlocal returns when passing conditional messages *ifTrue: block*, *ifFalse: block* to boolean objects, which allows early return values and skip the rest of code execution in a given method. Proper use of both local and nonlocal returns enables a programmer to write more powerful and clean code in Mysidia Programming Language. The important take is that developers should avoid sending return messages inside a closure unless they mean to return from the enclosing method. Failure to realize how local and nonlocal return works in Mysidia closures will lead to bugs that may be extremely confusing to the coders.

An implication of Mysidia's implementation of Local/Nonlocal return mechanism is that early return is not possible inside a closure, since the return message will always lead to exit from the enclosing method. This may seem to be a limitation at first, but it is actually a good design decision since multiple return point is proven to be more problematic than useful for the majority of the software problems. Also closures are meant to be concise and lightweight, and it is recommended that a closure body should not exceed more than 10 lines. Otherwise, it may be better to just define a new method inside the class.

Blocks and Arrow Closures

Mysidia supports two alternative syntax for special cases of creating closures. The first special closure variant is to declare a closure without any parameters, having only the code block enclosed in curly brackets with the leading paranthesis omitted. Such closures are referred to as **Blocks**, as they simply execute the code inside upon receiving message *.value()* or *()*. We've already see blocks in actions when passing it as argument to a conditional message. More examples of blocks are shown below:

```
var block, block2;
block = { Transcript.show("Evaluating Block.") };
block();
```

```
block2 = { Transcript.show("Evaluating Block 2") };
1 < 0 ifTrue: block1 else: block2
```

The first block object is evaluated immediately after it is assigned, and the resulting text "Evaluating Block" is printed on the screen. Then both block and block2 are passed as arguments in the message selector *ifTrue: else:* to a boolean receiver object(which evaluates to false). Upon activating the corresponding method in the false object, the second block is evaluated, and the resulting text "Evaluating Block 2" is printed on the screen.

The other special closure variant can be used when the closure body contains only one line of code. This one single line usually computes a meaningful value from the closure. If the closure contains only one line of code, it is possible to use a special closure syntax known as **Arrow Closure**. An arrow closure has a parameter list enclosed in parenthesis, followed by an arrow sign =>, and then a single line of code that is not enclosed in curly brackets. It is very similar to single line methods, which also uses the arrow notation => without curly brackets for the method body.

The below code snippet demonstrates how arrow closures can be created and used, which creates three arrow closures which has 0, 1 and 2 parameters respectfully. They all return a computation result from the closure body, and the results are printed out by the Transcript object to the screen:

```
var arrow0, arrow1, arrow2;
arrow0 = () => 5 * 2 / 4;
arrow1 = x => x * 4;
arrow2 = (x, y) => x * y;
Transcript.show(arrow0());
Transcript.show(arrow1(2));
Transcript.show(arrow2(2, 3));
```

It will print the results *2.5*, *8* and *6* on the screen, as we expect. It is worth noting that arrow closure will require the leading parenthesis for parameters even if no parameters can be passed to the closure. However, in the case of exactly 1 parameter, the parenthesis can be omitted, leading to more elegant code that is frequently used in many other programming languages that support functional programming. Arrow closures are widely used as arguments passed in enumeration messages, as we will see them in action at chapter 10 about Enumeration.

Contexts

In the earlier section, we have observed that local variables declared inside closures cannot be accessed by outside of the closure body, while closures are able to capture local variables declare outside of its body. This is characterized by the concept of Context, which represents the point of program execution. A context object is used by the VM to access the runtime state of the executing method/closure. Context is effectively similar to to stack frames in other programming languages, but Mysidia's context objects are more powerful since they are effectively reified stack frames. We can inspect the context object, manipulate the outer context that activates it, which is the sender of a message that activates the corresponding method/closure.

A context object is created when a method or closure is activated upon receiving the corresponding message, and terminated when the execution finishes. Mysidia has a special implicitly declared variable called **thisContext**, which refers the current executing context. We can inspect *thisContext* by sending a series of messages to itself, consider the following example:

```
class CylinderCalculator {
    methods: {
        surfaceArea(radius, height){
            var pi = 3.14;
            Transcript.show("Inspecting surfaceArea Context: " +
thisContext.inspect());
            return (num){
                var a = 0, b;
                Transcript.show("Inspecting surfaceArea closure's Context: " +
thisContext.inspect());
                num * pi * radius.square() * height
            };
        }
    }
}

main{
    var calculator, area;
    calculator = CylinderCalculator.new();
    area = calculator.surfaceArea(radius: 2, height: 5);
    Transcript.show(area(3));
    Transcript.show("Inspecting Main Context: " + thisContext.inspect());
}
```

If we run the code a lot of details about the current executing context will be printed on the screen, most of them are out of the scope of this chapter. However, we can still extract some useful information such as the sender object, receiver object, local variables etc:

```
Inspecting Surface Area Context:
class -> Class: MethodContext
Method -> Method: CylinderCalcualtor >> surfaceArea(argc, argv)
sender -> MethodContext : Program >> main()
receiver -> Method Object
selector -> surfaceArea: height:
is closure -> false
instance fields -> pi
scoped variables -> radius: 2, height: 5, pi: 3.14
... other details omitted ...

Inspecting Surface Area Closure Context:
class -> Class: ClosureContext
method -> Method: CylinderCalculator >> surfaceArea(argc, argv)
sender -> MethodContext: CylinderCalculator >> surfaceArea(argc, argv)
receiver -> Closure Object
is closure -> false
instance fields -> a, b
scoped variables -> radius: 2, height: 5, pi: 3.14, a: 0, b: Null
... other details omitted ...
```

```
Inspecting Main Context:
class -> Class: MethodContext
method -> Method: CylinderCalculator >> surfaceArea(argc, argv)
sender -> VMContext
receiver -> Closure Object
selector -> main
is closure -> false
instance fields -> calculator: Object CylinderCalculator, area: Object Closure
scoped variables -> calculator: Object CylinderCalculator, area: Object Closure
```

As we can see from the text above, the magical *thisContext* holds a lot of information regarding the current executing method or closure. When a method is activated, the corresponding *MethodContext* is assigned to the *thisContext* implicit variable capable of inspecting the envirionment. When a closure is created however, the variable *thisContext* within the closure body changes to the environment of the closure. The closure has access to the enclosing method context(or in the case of nested closure, inner closures also has access to the outer closure context). For this reason, the local variables declared in method body(or outer closure body) are also accessible in the closure itself. Details about the context classes and what messages they can understand will be discussed in next chapter.

However, local variables declared in closure body cannot be accessed by the enclosing method, since the method context does not hold reference to its inner closure's context. This parallel's smalltalk's context, and is similar to how scope works in other programming languages. The variable *thisContext* is more powerful than what we've seen so far, recall from chapter 4 that return statement is effectively sending tertiary messge *return value* message to thisContext object as receiver.

It is worth noting that the local variables declared inside a method/closure are referred to as instance fields on their corresponding context object. This is an interesting feature Mysidia offers, as every variable is an instance variable on another object. We will explore the idea of Unified Variable Implementation in chapter 11: Immutability and Slots. Also more details about the magic of *thisContext* will be discussed in later chapters.

Summary

As Mysidia does not support control structures like many other programming languages do, it relies on closures to complete certain complex tasks. Closures are deferred block of code that can be executed at later time, they are objects that can be assigned to variables or passed as argument in a message to receiver objects. Due to the nature of nonlocal return, it is necessary to be cautious when sending the explicit return messages inside a closure. When a method or closure is activated, a context object is created and can be used to get inner information regarding program execution. Closures and Contexts are magical tools developers can use to write more powerful programs in Mysidia, the next chapter will introduce the basic classes in Mysidia's standard library.

Chapter 8: Standard Library and Basic Classes

Introduction

Mysidia comes with a rich **Standard Library**, which has a group of basic classes and most commonly used classes. It is necessary to understand how they work in order to write effective applications in Mysidia. This chapter covers Mysidia's standard library and a few very basic/primitive classes that every developer will find useful. A complete coverage is beyond the scope of this language specs, and one may look up class reference/documentations for advanced usage of the standard library.

Mysidia Standard Library

Mysidia's standard library consists of hundreds of classes and other resources, which provide very basic functionality that every developer needs to write programs in Mysidia. The standard library is located in namespace **Mysidia.Standard**, with the 5 following subnamespaces(or packages):

- Mysidia.Standard.Lang
- Mysidia.Standard.Utility
- Mysidia.Standard.Collection
- Mysidia.Standard.GUI
- Mysidia.Standard.IO

Among all these resources available, the most fundamental and widely used is the package **Mysidia.Standard.Lang**. The basic classes such as **Object**, **Class**, **Message** and **Transcript** all come from this subnamespace, as well as primitive objects such as **UndefinedObject**, **Boolean** and **Number**, etc. It is at the core of Mysidia Programming Language, all the classes are written in the native code(Java, C or Javascript depending on the implementation). For this reason, they are also referred to as Kernal **Classes**. When we write a simple variable declaration *var a* and assignment message *a = 2*, we are effectively using classes **Object**, **Int** and **Message**. It is safe to say that we have been using this package since the beginning of the introduction chapters, without realizing their presence.

The **Lang package** is so important that Mysidia automatically imports this entire package when a namespace is declared in any Mysidia files. For this reason, there is no need to explicitly send the *using* message to import *Mysidia.Standard.Lang*, it is already done behind the scene. This is why we can use shortened class names such as *Class*, *Namespace* and *Transcript* without referring to their fully qualified class names. It is recommended to avoid creating classes with the same names as used in this package, although doing so does not cause an error, but rather *shadows* the classes in the Lang package. Developers can still refer to classes in Lang package by their fully qualified class names(ie. Mysidia.Standard.Lang.Message, incase one creates another class named Message), even if they are *shadowed* by another class.

The other four packages(Utility, Collection, GUI, IO) are also bundled in Mysidia's standard library. The **Utility package** provides additional functionality that developers may find useful from time to time, such as *Date* and *Random* classes. The **Collection package** consists of a hierarchy of classes that represent groups of objects, among them the most commonly used are *Array*, *List*, *Map*, etc. The **GUI package** has all the UI classes such as *Button*, *Checkbox*, *DatePicker* which are needed for desktop/mobile graphical applications. The **IO package** contains a selection of classes for file system and data input/output streams. All these packages will come in handy for writing programs in Mysidia, however simple or complex. We will give an overview for the core primitive classes in the next few sections(in package *Mysidia.Standard.Lang*), while the classes that demonstrate collections will be discussed in next chapter.

Class Object

The class **Object**(Mysidia.Standard.Lang.Object) is at the root of class hierarchy, the superclass that every class in Mysidia derives from. It provides default behaviors common to all objects, such as accessing, copying, comparing, error handling and message sending. For instance, all objects understand the comparison messages == *obj* and *!= obj*, the corresponding methods are implemented in the root Object class, with a few specialized subclasses providing their own versions.

Every Mysidia object understands the messages *.memberOf(cls)* and *.instanceOf(cls)*, the former checks if this object belongs to a specific class, the latter checks if this object belongs to the class or its subclasses. It is also possible to send messages *.respondTo(msg)* to any objects to check if they can respond to a given message. Mysidia supports shallow and deep copying of objects, which can be achieved by sending messages *.clone()*(shallow copy) and *.copy()*(deep copy) to the given objects. We can also do null-check on every object to see if it is uninitialized/undefined using *.isNull()*, or we may send message *ifNull: block* to perform some actions if a given object is null.

The usage of basic operations on Mysidia objects are shown in the code snippet below:

```
a == b
a != b
a.memberOf(Number)
a.instanceOf(Number)
a.respondTo(.isNull())
c = a.clone()
c = a.copy()
a.isNull()
a ifNull: { Transcript.show("variable a is null.") }
```

The class object has a special method which is activated when any objects receives a message that it cannot find a matching method to respond with. In this case, the receiver object will activate method *messageNotUnderstood(message)*, whose default behavior is just to print the text that the object from a given class cannot understand the supplied message with its selector and argument list. For instance, sending message *.abc()* to a number object will result in the following error message on the screen:

```
2.abc();
// Error: class Int does not understand message with selector "abc".
```

The above error messages can be very helpful for debugging code and programming errors.If a subclass overrides this method, it will be able to redefine the behavior and even resume execution by activating a different method. This is called Message Interception, and will be explored in details at later chapters.

Class UndefinedObject

The class **UndefinedObject** (Mysidia.Standard.Lang.UndefinedObject) is a subclass of root class *Object*, which has a unique instance *null* defined as a specially reserved variable that can be used anywhere in the program. Null represents a value for uninitialized variables, as well as meaningless results returned from methods/closures. As everything is an object in Mysidia, it is understandable that the null value is also an object, which conforms to the philosophy of Mysidia programming language.

The purpose of UndefinedObject and its instance null is primarily for error/exception handling. If a message is sent to a null object, then it is highly likely that the programmer uses an uninitialized variable which should be assigned a value before its use. The text printed on the screen will tell that the message is sent to a null object, immediately implying the cause of the human error.

UndefinedObject overrides the five methods *isNull()*, *isNotNull()*, *ifNull(block)*, *ifNotNull(block)* and *ifNull(block, else: block2)* to provide the opposite implementation as its superclass. The below code snippet illustrate how sending messages to null works:

```
var a; //declare variable a, it will be null before assigning a value.
Transcript.show(a.isNull());
Transcript.show(a.isNotNull());
a ifNull: { Transcript.show("Evaluating a block for null value.") };
a ifNotNull: { Transcript.show("Evaluating a block for non-null value.") };
a ifNull: { Transcript.show("Evaluating a block for null value") }
  else: { Transcript.show("Evaluating a block for non-null value.") };
```

The result printed on the screen will be: *true*, *false*, and Evaluating a block for null value x 2. The messages such as *ifNull: block* are elegant way to handle the presence of null values in a program. It is also worth noting that Mysidia's UndefinedObject class is not a singleton class, thus it is possible for create multiple instances from this class which will be identical to the special variable null. However, it is not recommended since it achieves nothing and can make the program harder to reason and test.

Class Boolean and its Subclasses

Booleans are the most widely used classes/objects available in Mysidia. The class **Boolean** (Mysidia.Standard.Lang.Boolean) is an abstract class that extends directly from the root class **Object** (Mysidia.Standard.Lang.Object). It has two subclasses **True** and **False**, each of which have two unique instances *true* and *false* that are defined as specially reserved variables that can be used anywhere in the script. *true* and *false* are special that are constants and cannot be overriden, thus it is impossible to declare variables by the name true or false. The boolean class hierarchy mirrors smalltalk and ruby's, except that *True* and *False* are not singleton classes. It is possible to create other instances of these two classes, although there is no practical reason to create other instances of *True* and *False*.

The Boolean abstract class and its two subclasses all understand the messages *&& arg*(and: arg), || *arg*(or: arg) and *not*. These are important logic messages that will return a boolean true or false object depending on the receiver object and the argument passed, as demonstrated in the example below:

```
var a, b;
a = 1 < 2;
b = 3 > 4;
Transcript.show(a && b);
Transcript.show(a || b);
Transcript.show(a not);
```

As every developer knows, boolean-and evaluates to true only if both operands are true, while boolean-or evaluates to true so long as one operand is true. The boolean-not negates the logic, converting true to false, and false to true. The following results are printed on the screen, as we expect:

```
false
true
false
```

The logic messages are implemented using the VM's native language, and they are optimized to be fast and efficient. The evaluation is lazy, which means that the argument will not be evaluated if the result of return value from the message can be determined already from the receiver object itself. For instance, the message *&& arg* quicky returns false if the receiver object is a False object, while || *arg* immediately returns true if the receiver object is a True object. In fact, Mysidia's true/false rule specifies that everything is true except for: *null* object, boolean *false*, integer *0*, *empty string* and *empty array*. For this reason, one can safely pass these logic messages to anything even if they are not boolean true or false object.

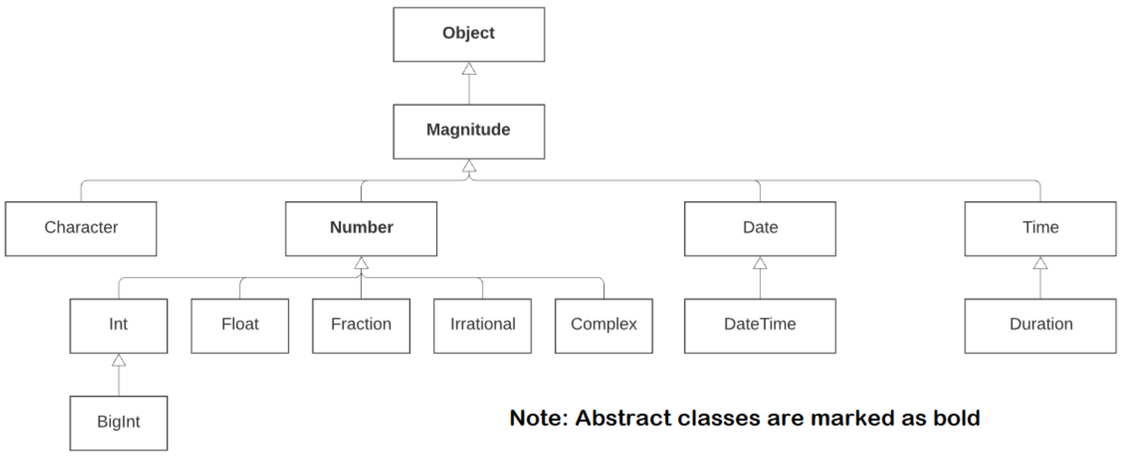
The Boolean classes also understand the conditional messages *ifTrue: block*, *ifFalse: block*, *ifTrue: trueblock else: trueblock*, *ifFalse: falseblock else: trueblock*. As Mysidia does not have language support for procedural conditional statements, these messages are the one and only way to achieve conditional branching. Consider the following code snippet:

```
10 > 3 factorial ifTrue: { Transcript.show("evaluate true") };
10 > 3 factorial ifFalse: { Transcript.show("evaluate false") };
10 > 3 factorial ifTrue: { Transcript.show("evaluate true") } else: {
Transcript.show("evaluate else") };
10 > 3 factorial ifFalse: { Transcript.show("evaluate false") } else: {
Transcript.show("evaluate else") };
```

The command line propmpt will display "evaluate true", "evaluate true" and "evaluate else" respectfully, since the receiver object 10 > 3 factorial is true. Internally, the two boolean subclasses implement ifTrue and ifFalse separately. The true object evaluates the block for *ifTrue: trueblock* message, and it skips the *ifFalse: falseblock* message, while the false object does the opposite.

Class Number and its Subclasses

As in Mysidia everything is an object, this also applies to numbers which are implemented as immutable objects in the virtual machine. At the top of the number class hierarchy is the abstract class **Number** (Mysidia.Standard.Lang.Number), which is a subclass of the abstract class **Magnitude**(this abstract class provides comparison methods like ==, !=, >, <). The class *Number* offers common arithmetic calculations(+, -, /, ^, etc) and mathmatic functionality(ie. *abs*, *power*).



The concrete subclasses **Int** and **Float** extend from class *Number* to provide specialized features for integers and floating point numbers. There are also a **BigInt** class that extends from *Int* for larger than 64-bit integer values. Mysidia also provides *Fraction*, *Irrational* and *Complex* classes, but they are from the package Mysidia.Standard.Utility and will need to be imported before they can be used(or refer to them as their fully qualified names). The below code snippet demonstrates some arithmetic and logic messages numbers can understand:

```
2 + 4.5
5 - 3.2
3.6 * 10
8 / 4
7 % 3
2 ^ 3
15 == 16
15 != 16
15 > 16
15 >= 16
15 < 16
15 <= 16
```

The results are *6.5*, *1.8*, *36*, *2*, *8*, *1*, *false*, *true*, *false*, *false*, *true* and *true*, respectfully. Mysidia does not have the concept of operators, arithmetic and logic operations are completed by sending messages to objects which understand them. In the above example, we send binary messages to the receiver object on the left of the symbolic messages. Other than the arithmetic and logic messages, numbers also understand other mathmetic messages such as *.abs()*, *.pow(arg)*, etc. Some messages can only be understood by certain subclasses, ie. *.factorial()* for Int and *.round(digit)* for Float. There are also messages *.toInt()* and *.toFloat()* for converting among number classes. The below code snippet shows a few more example messages can be sent to number objects:

```
5.abs();
2.pow(3);
3.1415926.round(2);
```

```
4.toFloat();
7.5.toInt();
```

Since numbers are immutable objects, every message that attempts to change/mutate a number will instead result in another number object returned from the corresponding method. It is also worth noting that Mysidia's integers use 64 bits system so there are minimum/maximum values for them(from - 2 ^ 63 - 1 to 2 ^ 63 -1). It is possible to get these two values by sending message *.min()* and *.max()* to Int class. In contrast with how Java/C# handles integer overflows, Mysidia will automatically convert it to *BigInt* if the number falls outside of the bound. For instance, Int.max() + 1 will make an instance of BigInt object, it will not become Integer.min() instead.

Numbers in Mysidia can also understand iteration messages, which create an instance of Range class with the receiver object as the starting value. To define iteration, we can send keyword message *do: block* to a Range object. This is the Mysidia way for handling iterative behaviors, as it does not support procedural while/for loops. Among the iteration messages the most commonly used are *.times()*, *.upto(upperlimit)*, *.downto(lowerlimit)* and *.step(size: size, limit: limit)*:

```
6.times() do: (i){ Transcript.show(i) };
1.upto(5) do: (i){ Transcript.show(i) };
4.downto(2) do: (i){ Transcript.show(i) };
2.5.step(size: 0.3, limit: 5.5) do: (i){ Transcript.show(i) };
```

All of the code above will create iterations in Mysidia, and prints out the counter *i* to the console. The first line prints the number from 1 to 6, and the second line does this from 1 to 5, with step size 1 for each iteration. The third line prints number from 4 to 2, with step size -1 for each iteration. The fourth line prints from 2.5 to 5.5 with custom step size(2.5, 2.8, 3.1, ..., 5.5). Note the message *.times()* message can only be sent to an integer object, it will cause an error if sent to a floating point number. The remaining three iteration messages work for any number objects. Since the class Range is in fact a collection class, its details will be examined in next chapter along with the other messages Range objects can understand.

Class Character

Characters are unicoded letters, numbers or symbolic entity that represents single grapheme units. In Mysidia, characters are defined by wrapping a unicoded entity in single quote (''), ie. *'a'*, *'0'*, *'$'*. Some special characters need to be prefixed with \u, such as unicoded entities: '\u00A0', '\u0071'. Each character is an immutable and unique instance of class **Character** (Mysidia.Standard.Lang.Character). The default encoding format for characters in Mysidia is UTF-8, although it is possible to configure different encoding such as ASCII, UTF-16, etc.

Characters understand messages *.value()* and *.asciiValue()*, which are integer representations of the characters in UTF-8 and ASCII encoding respectively. The class Character is a subclass of abstract class Magnitude, and it also understands comparison messages such as >, <, ==, etc. Consider the code snippet below:

```
var char = 'a';
Transcript.show(char.value());
Transcript.show(char.asciiValue());
Transcript.show(char > 'b');
Transcript.show(char < 'b');
Transcript.show(char == 'b');
```

The following result will be printed on the command line prompt. Note the messages *.value()* and *.asciiValue()* sent to the character 'a' will yield same result, but this is only true for ASCII characters. A UTF-8 char such as the greek letter 'Π' will only have a valid result from message *.value()*, while the other message *.asciiValue()* will return 0.

```
97
97
false
true
false
```

As arrays of characters are very common in composing software, Mysidia provides a specialized class called *String* which represents such an immutable collection of characters. We've used strings thoroughly in the last few chapters, and details about the class String(array of characters) will be explored in next chapter about basic/literal collections.

Class Closure

As closures themselves are objects from the class Closure (Mysidia.Standard.Lang.Closure), it is also possible to send messages to them. We've seen in last chapter that sending the message *.value()*, *.value(arg)* or *.values(args)* to a closure object will evaluate the code in its body, but thats not the only messages they can understand. We may also pass the message *whileTrue: block* and *whileFalse: block* to closures to repeat a block closure of code multiple times until the closure evaluates to True or False. Consider the below example:

```
var i = 0, a = ArrayList.new(5);
{ i < a.length() } whileTrue: {
    i += 1;
    a[i]= i ^ 2;
}
Transcript.show(a);
```

It will print the result 1 4 9 16 25 on the screen, as one would expect coming from other language background. Mysidia does not have language support for while loop control structures, but this feature can be achieved by sending message to closure objects. Both block or arrow closures can be passed as argument with selector whileTrue/whileFalse, though block closures are usually preferred for these operations.

Closures also comes in handy for Exception handling, as they understand message *ifError: block* and *on: exception catch: block*. The second variant is suitable when it is necessary to inspect the thrown exception object, otherwise the first variant may be used. The following code snippet demonstrates how exception handling works in Mysidia, further details on this topic will be discussed in the later chapters about Errors and Exceptions.

```
var i = 0, j;
{ j = 5 / i } ifError: {
    Transcript.show("Cannot divide a value by zero.")
};

{ j = 5 / i } on: exception catch: {
    Transcript.show(exception.getText())
    // this should contain the default text generated for ZeroDivideException.
};
```

It is worth noting that methods in Mysidia are simply a combination of selectors and closures, although they usually exist in the form of CompiledMethod and CompiledClosure for efficient program execution. More about Closures, Methods and their compiled forms will be discussed in the last chapter about VM implementation.

Class Context and its Subclasses

As we have seen from last chapter, the execution contexts of methods/closures are first class objects in Mysidia, which can be accessed and modified at any point of code execution. The abstract class **Context** (Mysidia.Standard.Lang.Context) inherits directly from the root object class, and it contains methods that can respond to messages common to any types of context objects such as *.inspect()*, *.var(name)*, *.var(name, value: value)* and *.return(value)*. The concrete subclass **MethodContext** represents the execution of a method, while the other subclass **ClosureContext** represents the execition of a closure.

As we've seen from last chapter, the specially reserved variable *thisContext* is automatically created inside each method or closure to represent the current executing context. The easiest way to access context objects is to directly send messages to *thisContext* object. Assuming the method of the *surfaceArea(radius, height)* or the returning closure, we can send these messages to the context object:

```
class CylinderCalculator {
    methods: {
        surfaceArea(radius, height){
            var pi = 3.14;
            Transcript.show(thisContext.hasInstanceField(#pi))
                      ;.show(thisContext.isClosure())
                      ;.show(thisContext.receiver())
                      ;.show(thisContext.currentLine())
                      ;.show(thisContext.selector());
            return (num){
                var a = 0, b;
                Transcript.show(thisContext.hasInstanceField(#a))
                          ;.show(thisContext.isClosure())
                          ;.show(thisContext.receiver())
                          ;.show(thisContext.currentLine())
                          ;show(thisContext.outerContext());
                num * pi * radius.square() * height
            };
        }
    }
}
```

We modified the code from last chapter to remove the *.inspect()* message which returns all the info, and instead we can send individual messages to get the information useful to us. Every context object understands message *.hasInstanceField(name)*, which checks if the context has a given instance field. Remember from last chapter, local variables in method/closure bodies are none other than instance field on their context objects. The method *hasInstanceField(name)* will return true if a local variable is declared inside the method/closure body.

Context objects also understand message *.isClosure()* which returns true for *ClosureContext* and false for *MethodContext*. The message *.receiver()* will fetch the receiver object of the executing method/closure, but not the context object itself. The message *.currentLine()* asks the line number inside the file where the current execution is, it is useful for advanced metaprogramming. The message *.selector()* can only be sent to *MethodContext* objects as Closures do not have selectors. On the other hand, the message *.outerContext()* is understood only by *ClosureContext* objects, it will return the enclosing method or closure(if inside a nested closure).

The Context objects understand a few tertiary messages such as variable declaration message *var a*, variable declaration and assignment message *var a = 5*, return value message *return x*, as well as throw exception message *throw Exception.new("Error occurred")*. These messages are special and are optimized by the VM internally. The method *return(value)* is handled differently in *MethodContext* and *ClosureContext*, as the former returns from its own body while the latter returns from the enclosing method body.

Note Mysidia does not have reserved keywords for these tertiary message selectors, though it is recommended to avoid naming variables by *var*, *return* and *throw* so it will not trick the parser into unprecictable behaviors. Advanced usage of Context objects will be explored in later chapters together with the concept of Continuation.

Summary

This chapter gives a very minimalistic introduction on Mysidia's **Standard Library**, specifically the core package **Mysidia.Standard.Lang**. We have seen the very basic classes in action, understanding how they work and how to use them is a good starting point to write simple Mysidia applications. As everything is an object in mysidia, we can send some common messages to any objects and they will be able to respond to them properly. The primitive classes such as Boolean, Number and Character provide specialized features that developers may find useful. All these basic classes from the package *Lang* are written in the native language through the underlying VM, and they can be used without importing the namespace for convenience reasons. We will further explore collection classes in the next two chapters, which will enable developers to write more complex and elegant programs.

Chapter 9: Arrays and Strings

Introduction

The last chapter makes a detailed description about Mysidia's standard library as well as the basic classes in the **Mysidia.Standard.Lang** package like Object, UndefinedObject, Boolean, Number, etc. There are more classes in the Lang package that deserves mentioning, especially the basic collection classes. Mysidia's collection classes all inherit from the abstract Collection class from the package **Mysidia.Standard.Collection**, although the basic collection classes are defined in the Lang package for convenient usage. This chapter will introduce these most common collection classes such as Array, String and Dictionary, etc.

Literal Collections

Among the available collection classes, there are a few special classes that Mysidia define them in the Lang package and provide literal forms for these important collections. These classes are Array, String, Symbol, Dictionary and Range. To create instances of these classes, we can use the literal forms as below:

```
Array Literals: [1, 2, 3, 4], ["US", "EU", "CA", "RU", "JP"]
String Literals: "Hello World", "Welcome to Mysidia"
Symbol Literals: #arg, #MyClass
Dictionary Literals: ["US": "USA", "EU": "Europe", "CA": "Canada"]
Range Literals: 1...4, 10...100
```

The above literal collections are all immutable, which means that they cannot be mutated once created and must be instantiated with all their elements set. Messages attempting to mutate the elements inside these collections will either result in a new collection object returned from the responding methods, or simply an exception thrown to indicate failure to update collection elements.

As these classes are all subclasses of Collection, they also understand the basic collection such as *.get(element)*, *.length()*, as well as enumeration messages such as *each: block* and *.collect(closure)*. Details of the basic collection methods will be introduced in depth at next chapter when we describe the entire Mysidia's Collections Framework. The next few sections will explain the five most common collection classes that we refer to as Literal Collections.

Class Array

Arrays are among the most widely collection objects used in a programming language. In Mysidia, the class **Array** (Mysidia.Standard.Lang.Array) inherits from the abstract *List* class, which is a fixed sized collection whose index begins at 0. This behavior is consistent with most C-family languages. There are usually two ways to create an array, by using array literals(inside square brackets) or sending message *.new(arg)* as demonstrated in the example below:

```
var a = [1, 3, 5, 7];
var a2 = [true, 1.5, "Hello World", DateTime.now(), a];
var a3 = Array.new("abc");
```

The first array object a is created using the literal version, which has elements enclosed inside a pair of square brackets delimited by comma(,). The second array object a2 is similar to a, except that it contains objects from different classes instead of the same integers as first array. The last array object a3 creates a new Array from a string "abc", which returns an array of characters ['a', 'b', 'c']. It is possible to pass any kinds of collection object to the message *.new(arg)*, and it will simply convert the argument object into an equivalent Array.

As Mysidia supports dynamic typing by default, an array does not need to hold objects of the same type as they do in most statically typed languages. For this reason, we can safely add any element to an array in Mysidia and there is no compile time error generated. Still, it is recommended to avoid putting objects of completely unrelated classes in an array, unless there is a very good reason. Also Mysidia's array literals can contain complex expressions/message compositions as well as variables as element, unlike Smalltalk's array literals that everything must be a constant literal itself.

To access an element in a Mysidia array, we can send message *.get(index)*. The index must be an integer and the usualy range is between 0 to array size - 1. If index is equal or greater than the size of array, an IndexOutofBoundException will be thrown. Negative indexes work too, in which the index -1 represents the last element, and the index -2 represents the second last element, etc. It is possible to send the square bracket message [index] to the Array object, which is equivalent to *.get(index)*. The below code snippet shows a few examples of array accessing:

```
var a = [1, 4, 9, 16, 25];
Transcript.show(a.get(0));
Transcript.show(a.get(4));
Transcript.show(a.get(-2));
Transcript.show(a[1]);
Transcript.show(a[1].isEven());
```

Run it from command prompt and we will see the follwing 5 numbers: 1, 25, 16, 4 and true printed on the screen. The square bracket syntactic sugar is converted into *.get(index)* automatically by the parser, which is considered as a primary message that follows the precedence order from left to right when composed with other messages. The last line is to send message *.isEven()* to element *a[1]*, as expected.

As a subclass of the Collection and List class, Array in Mysidia understands every message that Collection and List class understand. It is possible to send messages *.isEmpty()*, *each: closure*, etc which are understood by Array's superclasses. Since Arrays in Mysidia are immutable, changing the elements inside an array is disallowed. In fact, attempting to mutate arrays will simply result in a new array object returned from the corresponding methods. If mutable version of array is desired however, we may use the class *ArrayList* which will be introduced in next chapter along with other collection classes from the Collection package.

Class String

Strings are a special collection objects that represent arrays of characters, they are instances of class **String** (Mysidia.Standard.Lang.String). String literals are created by wrapping a sequence of characters inside a pair of double quotes(""). "Hello World" is a textbook example for a typical string, though we can also create strings with a single character like "a", "1" or even empty string "". It is also possible to create strings by sending message *.new(arg)* to class String, which will still give the same string objects as creating them using the double quote literals. The below 3 approaches will create the same string object:

```
var a;
a = "Hello World";
a = String.new("Hello World");
a = String.new(['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']);
```

As strings are objects just like anything else in Mysidia, we can send messages to them as well. If the receiver string object understands a given message, it will execute the corresponding matching method. Unlike Smalltalk and Ruby, strings are immutable in Mysidia, hence every method that appears to manipulate a string will not modify the original string. Instead, a new string object will be created and returned. For instance, the message *.capitalize()* sent to string object "america" will return capitalized string object "America", but it is a completely new string object instead of the original one. The code snippet demonstrates a few messages that every instance of String understands:

```
var str = "america";
Transcript.show("The string: ${str}'s length is: " + str.length());
Transcript.show("The capitalized version is: " + str.capitalize());
Transcript.show("It contains substring \"er\": " + str.contains("er"));
Transcript.show("The first index of 'a' is: ${str.indexOf("a")}, the last index of 'a'
is: ${str.lastIndexOf("a")}");
Transcript.show(str.get(1));
Transcript.show(str[3]);
```

String objects understand the message *.length()*, which returns the string's character length. The message *.capitalize()* will attempt to convert the first letter of the string to be uppercase, similar messages *.toUpper()* and *.toLower()* will convert the entire string to upper/lower case. The message *.contains(substr)* will check if a substring(or character) exists as part of the string. The messages *.indexOf(substr)* and *.lastIndexOf(substr)* will find the first and last index of a substring(or character) inside the receiver string. The last two messages attempt to acquire the character stored at the index 1 and 3 positions of the string object, note the square bracket message [3] works the same ways as *.get(3)*.

There are far more messages we can send to string objects, a complete coverage can be found in class references. If we run the above code, the following result will show on the screen, note we use \" to excape the double quote character *"* so it can be used inside a string object:

```
The string: america's length is: 7
The capitalized version is: America
It contains substring "er": true
The first index of 'a' is: 0, the last index of 'a' is 6.
m
r
```

Mysidia supports **string concatenation**, which is effectively sending message "+ str" to the receiver string object. String concatenation is widely used when we need to append a string in a variable to a string literal, or vice-versa. Another way to achieve this is to use **string interpolation**, which allows embedding variables inside a string literal. This implementation is handled in the VM, in which the embedded variable is enclosed inside the a placeholder delimited with *${}*. The below string concatenation and interpolation give the same result:

```
"The string: " + str + "'s length is: " + len;
"The string: ${str}'s length is ${length}";
```

Usually string concatenation is preferred for appending 1-2 simple strings together, while string interpolation is better at dealing with multiple strings, especially complex strings with lots of symbols that make concatenated strings hard to read/understand. As both tools are equivalent, it is ultimately up to the developers to decide what to use for their applications.

The root class *Object* has a method *toString()*, which allows itself to be converted to the string representation of itself. As every class inherits from the root object class, we can safely send the message *.toString()* to any objects in Mysidia and it will converts them to strings properly. Some subclasses of Object override method *toString()* and provide their own implementations. Sending message *.toString()* to the basic classes in Mysidia will usually result in their literal value returned enclosed in double quote.

Class Symbol

Symbols are unique identifiers that may be used for names of variables, classes, namespaces, etc. The class **Symbol** (Mysidia.Standard.Lang.Symbol) is a subclass of class String and inherit most of the String methods. However, a symbol object may contain only alphanumeric characters as well as the underscore(_) and dollar sign($) special characters, and it cannot start with a digital character. The below example shows what are valid and invalid symbol objects:

```
#helloWorld //Valid
#MyClass //Valid
#my_method //Valid
#$ //Valid
#2n //Invalid, 2 cannot be used as starting char
#+5 //Invalid, + is not an allowed special char
```

Symbols are frequently used when declaring variables as well as creating a class, and trait. As symbols are guaranteed to be valid for the names of variables, they are highly suitable for this purpose. The VM also optimizes declaration/creation of these special objects with Symbols, which makes them efficient and more convenient to use. It is possible to convert a Symbol into a String and vice versa, note the latter may fail if the String cannot be converted into a valid Symbol:

```
#argName.toString() => "argName"
"HelloWorld".toSymbol() => #HelloWorld
"My String".toSymbol() => Error: String "Hello World" cannot be converted into Symbol.
```

As Mysidia's Strings are already immutable objects, and various syntactic sugar exists for variable/class declaration, there are not many use cases for Symbols. Usually the only reason to use them is to refer to a variable/class that has yet to be declared. However, Symbols are still very important, as internally Mysidia makes use of them heavily in the VM for sending primitive messages as well as speed optimization.

Class Dictionary

Dictionary (Mysidia.Standard.Lang.Dictionary) is a special literal collection class, which stores a collection of immutable key-value pairs known as entries. It is a subclass of the abstract class Map(Mysidia.Standard.Collection.Map), which defines basic behaviors for map type collections. A Dictionary object can be conveniently created by using the literal form, which is similar to creating an array, with key:value pairs(aka entries) enclosed in a pair of square brackets delimited by comma(,). Alternatively, it is possible to create a Dictionary from another collection object. Consider the following code:

```
var dict, dict2;
dict = ["US": "United States", "RU": "Russia", "EU": "Europe", "JP": "Japan"];
// Create a Dictionary with literals.
dict2 = Dictionary.new(keys: ["US", "EU", "RU", "JP"], values: ["United States",
"Europe", "Russia", "Japan"]);
// Create another Dictionary from an array of Keys and Values.
Transcript.show(dict == dict2);
// prints: true
```

The first variable dict uses the literal notation to create a Dictionary, which has US, RU, EU and JP as keys, and their corresponding values are United States, Russia, Europe and Japan. The second variable dict2 is created from two arrays that represent keys and values. If we test equality of these two, we find that the two dictionaries are identical, despite the fact that the elements are defined in different orders. This is because Dictionaries do not preserve the order the entries are inserted, it simply 'sorts' by the key's hash code value. For this reason, dictionaries with same set of entries are always identical.

The following code snippet demonstrate how dictionary objects respond to common Map-related messages *.contains(value)*, *.containsKey(key)*, *.keys()*, *.values()*. Dictionary objects also understand message *.get(key)* to obtain the values stored inside with a given string key. Note the square bracket syntactic sugar can be used for dictionaries as well, thus *[key]* is equivalent to *.get(key)*.

```
var dict = ["US": "United States", "RU": "Russia", "EU": "Europe", "JP": "Japan"];
Transcript.show(dict.contains("United States"));
Transcript.show(dict.containsKey("RU"));
Transcript.show(dict.get("EU"));
Transcript.show(dict["EU"]);
Transcript.show(dict.keys());
Transcript.show(dict.values());
```

The first line declares a variable dict and assign its value to be a dictionary object identical to last code example. The 2nd/3rd line will both print *true* as the dictionary contains "United States" as value and "RU" as key. The 4th/5th both dislay "Europe" on the screen, suggesting that they are identical in semantics. The last two lines will print out a set of Keys and Values, more about Set will be explained in next chapter.

Class Range

Another literal collection class immensely useful for Mysidia programs is **Range** (Mysidia.Standard.Lang.Range), which has the literal form *a...b*(triple dots). Ranges are immutable sequences of numbers that can respond to a good number of iteration and other collection-specific messages. In fact, ranges are automatically created when we send iteration messages to number objects, which returns the corresponding range objects that can understand message *do: block* to iterate over elements inside the range with a given *step*. The step value is 1 for literal ranges, although it is possible to specify a different value by sending message *.new(start: startNumber, end: endNumber, step: stepValue)*. The below code snippet demonstrates how ranges work in action:

```
var r = 1...5;
var r2 = 3.5...6.6;
var r3 = Range.new(start: 2, end: 11, step: 2);
Transcript.show(r);.show(r2);.show(r3);
```

If we inspect the elements by passing them to the Transcript object to display what are inside, the first range r1 contains: 1, 2, 3, 4, 5; the second range r2 contains 3.5, 4.5, 5.5, 6.5; and the last range r3 contains 2, 4, 6, 8, 10. As we can see, ranges are inclusive from the start to the end number. Ranges may contain numbers as well as any objects that can be represented as numeric values, ie. Character and DateTime.

Since Ranges are collections themselves, they can understand the same set of messages that access the elements inside. It is also possible to use the square bracket notation to send message *[index]* to a range object to obtain the element located at the given index. Ranges are also immutable collection objects like Arrays and Dictionaries, attempting to alter the elements in a range object will lead to creation of a new range or a runtime error. As evident from last chapter, ranges can respond to iteration message *do: closure* to loop over the elements inside. The below code example demonstrates what messages can be sent to range objects:

```
var r = Range.new(start: 1, end: 5, step: 2);
Transcript.show(r.get(0)); //prints: 1
Transcript.show(r[1]); //prints: 2
Transcript.show(r.contains(4.5)); //prints: false
r do: (i){ Transcript.show(i * i) }; //prints: 1, 4, 9, 16, 25
```

A unique feature about range objects is that they can be used as argument in message *.slice(range)* to return a sub-collection from the supplied start to end index. Slicing is a very helpful feature existing in many programming languages, Mysidia makes this elegant with square bracket notation as well as range objects. The below code snippet demonstrates how it works for Strings and Arrays:

```
var s = "Hello World", a = [2, 4, 6, 8, 10];
Transcript.show(s.slice(0...3)) prints: "Hell"
Transcript.show(s[0...4]); //prints: "Hello"
Transcript.show(a[1...3]); //prints Array: (4, 6, 8)
```

Summary

This chapter introduces the very basic collection classes from the core package **Mysidia.Standard.Lang**. They are referred to as literal collections as Mysidia defines a set of unique literal notations to represent these collection objects. The square bracket messages for accessing elements is an elegant syntactic sugar to make these collection classes convenient to use. As the literal collections all understand common messages from the abstract Collection classes, we will need to explore Mysidia's Collection class hierachy to learn more about what messages we may send to these objects. The next chapter will introduce the rest of Mysidia's collection classes from the package Mysidia.Standard.Collection, as well as the feature called Enumeration that makes Mysidia's collection classes great to use in writing real world applications.

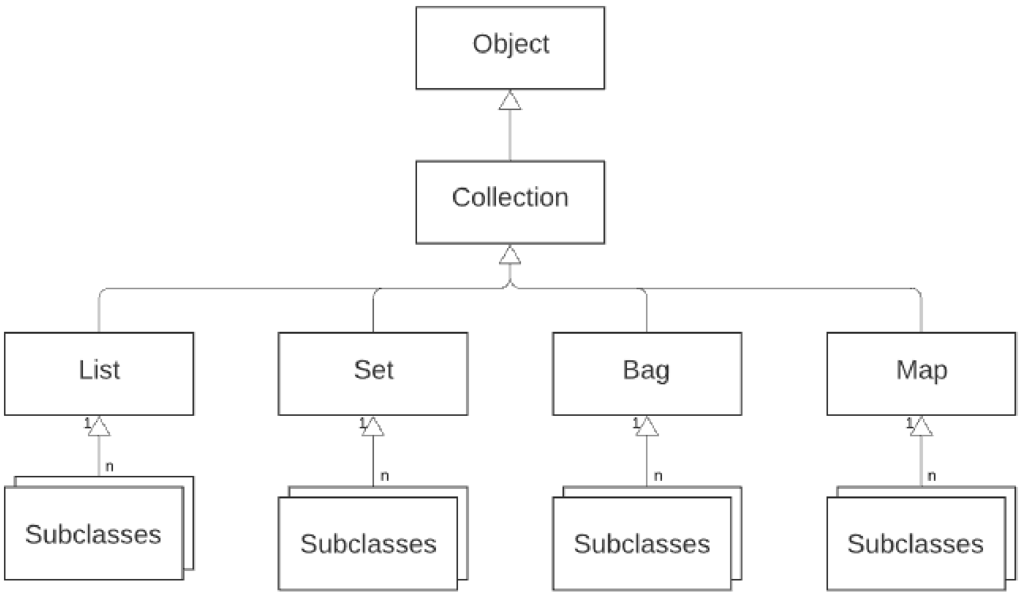
Chapter 10: Collections and Enumeration

Introduction

Following a brief description of the basic and literal collection classes in Mysidia, we are now well equiped to write small programs in Mysidia. However, in order to code medium and large sized applications, we still need solid understanding of more complex classes and concepts available in Mysidia Programming Language. The package **Mysidia.Standard.Collection** provides a rich library of such classes whose main purpose is to be containers of other objects. This chapter will explore the entire **Collection** package that every developer is likely to find them useful, as well as the concept of Enumeration and how it works with collection classes.

The Collection Class Hierarchy

There are a few tens of classes in Mysidia's *Collection* package, each of which provides various functionalities suitable for different use cases. At the top of the class hierarchy is the abstract base class called **Collection** (Mysidia.Standard.Collection.Collection), which shares the same name as the package itself. This class inherits directly from the root Object class, and provides common features applicable for all of its subclasses. Some of the methods are abstract and are implemented different across the subclasses of *Collection*, while others have default implementation.



Directly below the base *Collection* class are the 4 additional abstract classes: **List**, **Set**, **Bag** and **Map**. These subclasses of *Collection* are considered the base classes of the 4 families of corresponding collection types. The *List* sub-hierarchy contains a group of sequential collections whose elements are ordered by specific approaches, among them the most widely used subclasses are **String**, **Array**, **ArrayList**, **LinkedList** and **SortedList**. Remember from last chapter that *String* and *Array* actually belong to the *Lang* package, but they are subclasses of *Collection* and behave similar to other collection classes.

The *Set* sub-hierarchy is a family of container classes whose elements are unique and can only appear once in the collection, ie. **HashSet**, **LinkedSet**, and **SortedSet**. The *Bag* sub-hierarchy is exactly the opposite as *Set*, as it may contain duplicate appearances of the same object. The classes in the Set and Bag sub-hierarchies mirror each other. The *Map* sub-hierarchy has a special group of classes whose elements are stored and accessed using keys, good examples are **Dictionary**, **HashMap**, **LinkedMap** and **SortedMap**, note Dictionary belongs to the *Lang* package. Unlike the basic classes and literal collection classes, these classes from the *Collection* package are written in Mysidia itself, though some methods are optimized in the VM for better performance.

Instances of Collection and its subclasses understand the following types of messages: Accessing, Checking, Converting, and Enumeration as shown in the code examples below(assuming an instance of any arbitrary mutable collection class). Note mutable collection classes also understand messages of mutating its elements.

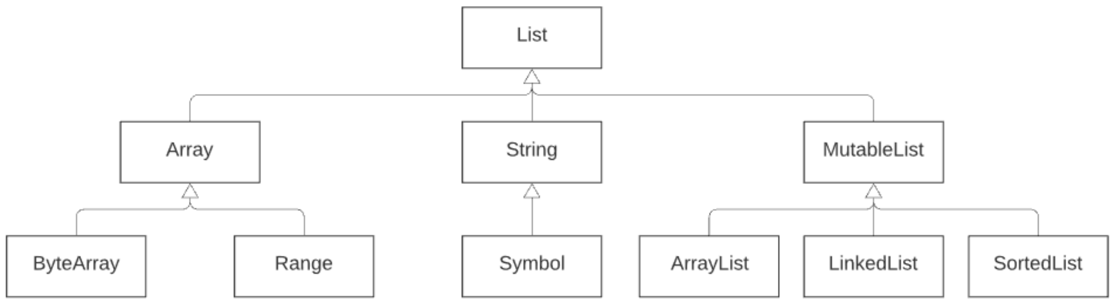
```
a.get(2);
a.isEmpty();
a.contains("Hello");
a.add("Friend");
a.remove("World");
a.toArray();
a each: (element){ Transcript.show(element.toString()) };
a.select(element => element.contains('e'));
```

The first line accesses the element at index 2 for collection a and returns it. The second code verifies if collection a is empty, while the third line tests if the string element "Hello" is found inside collection a. The fourth and fifth lines are mutating messages, adding/removing an element from collection a, though only instances of mutable collection classes may respond to such messages. The next code converts collection a to an array object, which is understood by every collection object. The last two lines are enumerating messages that iterate over collection a to do something with its elements. *Enumeration* is a rather important concept and feature with Mysidia collection classes, and it will be discussed in details in the later sections.

This is just an overview for Mysidia's collection class hierarchy, in the next few sections we will explore the most useful classes in detail. We will not be able to cover every subclass of Collection in this chapter, but the most commonly used ones will be discussed thoroughly to provide a comprehensive overview of how collection classes work in Mysidia, and how developers can use them in convenient and elegant ways.

Class List and its Subclasses

Lists are ordered sequential collections whose sizes usually grow with the addition of elements. Usually the order is the sequence for which each element is added to the lists, but it is also possible to define customized ordering/sorting mechanisms. At the top of the sub-hierarchy is the abstract class **List** (Mysidia.Standard.Collection.List), which itself is a subclass of **Collection** (Mysidia.Standard.Collection.Collection). Below the abstract List class are a group of subclasses such as **Array**, **String** and another abstract class **MutableList**. The mostly common subclasses of *MutableList* are **ArrayList**, **LinkedList**, **SortedList**, which will be explored in this chapter.



The abstract List class contains a few methods that every subclass can use to respond to the accessing messages available in collection objects such as *.isEmpty()*, *.contains(element)*, as well as enumeration and selection/filtering with block closures. The subclasses of *MutableList* also understand mutating messages(ie. *.add(element)*, *.remove(element)*). As Lists are sequential collections with indexes for each item inside, every instance of List and its subclasses can understand messages such as *.indexOf(element)*, *.lastIndexOf(element)*, etc. It is also possible to send message *.subList(start: element, end: element2)* to get SubLists from the Lists with given starting and ending index.

The most commonly used subclass among the List hierarchy is **ArrayList**, which extends from MutableList to inherit both accessing and mutating methods. It is very similar to a Mysidia Array except that it is mutable, thus can have elements inserted/removed at anytime. The elements are ordered by the sequence they are added. Though it is very easy to append an element to the end of the ArrayList, it is also possible to insert an element at any indexes. The square bracket notation from last chapter works for ArrayList too. Below are the example messages that can be sent to an instance of ArrayList:

```
var al = ArrayList.new();
al.add("Hello");
al.get(0);
al[1];
al.put(2, "World");
al.contains("My Image");
al.indexOf("BBQ Steak");
al.size();
al.toArray(); //convert to an Array, which will be immutable.
```

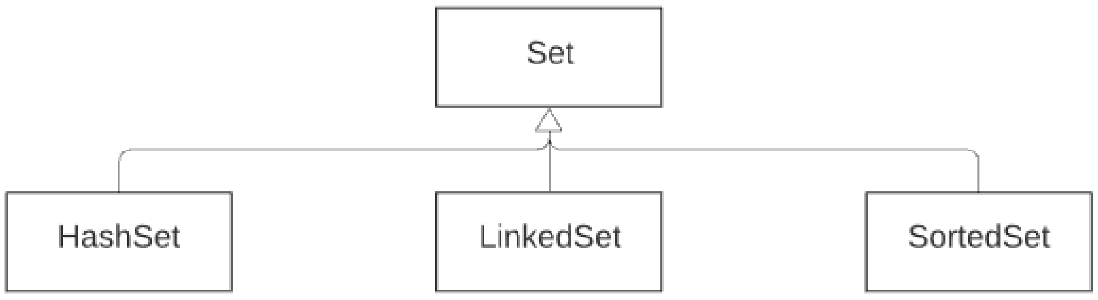
Another commonly used List subclass is LinkedList, which represents an ordered list whose elements are linked by the current element holding a reference to the next. Compared to ArrayList, it is easy to modify elements at the beginning of the LinkedList, and every instance of LinkedList understands message *.getFirst()*, *.addFirst()* and *.removeFirst()*. However, manipulating the last element with messages *.getLast()*, *.addLast()* and *.removeLast()* is rather inefficient, as it has O(n) time complexity for traversing the entire LinkedList. For efficient handling of last element, Mysidia provides **DoublyLinkedList** which has O(1) time complexity for accessing/modifying both first and last elements. Unlike ArrayList, neither LinkedList nor DoublyLinkedList understands the square bracket messages. The common messages that can be sent to LinkedList(and DoublyLinkedList) are shown below:

```
var ll = LinkedList.new();
ll.add("Hello");
ll.addFirst("World");
ll.getFirst();
ll.contains("Touchdown");
ll.removeLast();
ll.toArray();
```

The other subclass **SortedList** allows elements of the corresponding List to be sorted with arbitrary comparison logic. By default it uses a tree-like structure to sort elements by their the hash code values, but it can be changed by adding Comparator objects with message *.addComparator(comparator)*. The compator objects can either be instances of **Comparator** (Mysidia.Standard.Collection.Comparator), or a closure object that describes the comparison logic.

Class Set and Class Bag

Sets are unordered collections whose elements must be unique(aka no duplicate elements), which mirrors the mathematic concept for sets. At the very top of the Set class sub-hierarchy is the abstract class **Set** (Mysidia.Standard.Collection.Set), which provides common methods such as *union(set)*, *intersect(set)* and *except(set)*. Subclasses of Set can all understand these messages corresponding to mathematic Set operations, and returns a new Set with some elements.



The most common subclass for abstract class is the **HashSet** Class, which is a set whose elements are unique but does not preserve the order of addition. This class can understand all messages available to the superclasses Collection and Set, although it has its own implementation for methods *add(element)* and *remove(element)*. The subclass **LinkedSet** is similar but does preserve the order of addition, it also has a method *subSet(from, to)*, which understands the message to extract a sub set from itself. Another subclass **SortedSet** can use a comparator object to sort the elements in a specific order for each addition/removal operation. The code snipper demonstrate some examples on how to use Set classes in action:

```
var hs = HashSet.new();
hs.add("Hello World");
hs.add(true);
hs.add(2);
hs.add(true);
Transcript.show(hs);
```

```
var ls = LinkedSet.new();
ls.add("Hello World");
ls.add(false);
ls.add(2);
ls.add(3.14);
Transcript.show(ls);
Transcript.show(hs.union(ls));
Transcript.show(hs.intersect(ls));
Transcript.show(hs.except(ls));
```

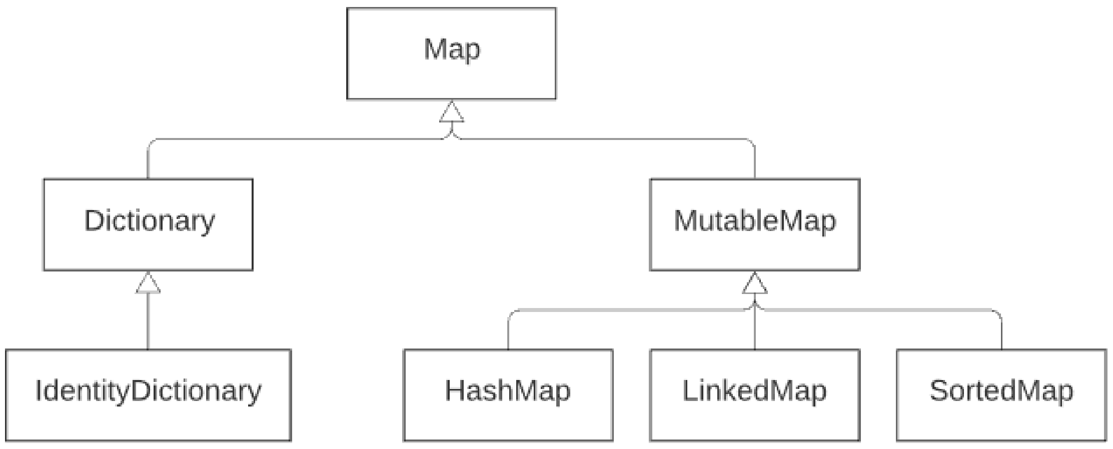
The Transcript will print the below result on the screen, which is as expected. The HashSet does not preserve order of addition, but LinkedSet does. Also note that adding the element *true* twice to a Set will lead to only one of them appearing in the Set, the duplicate is removed. The Union/Intersect/Except operations are exactly the same as what we learn from Mathematics classes, while the return values belong to the same classes as the receiver objects:

```
True, 2, Hello World
Hello World, False, 2, 3.14
False, True, 2, 3.14, Hello World
2, Hello World
True
```

The **Bag** class is similar to a **HashSet** except that it allows duplicated elements to exist, in a way a Bag can be considered as an unordered list. It has methods to support Union, Intersect and Except mathematic operations, and the return value is another Bag. It is also possible to send these messages with Bag as argument, while the return value is corresponding to the class of the receiver object. For instance, sending message *.union(set)* to a Bag object will return a Bag, while sending message *.union(bag)* to a set will return a Set. The message *.toSet()* and *.toBag()* can be sent to the receiver objects to convert them into Sets or Bags.

Class Map and its Subclasses

The Maps are special type of collection classes whose elements are key-value pairs. At the top of the class hierarchy is the abstract **Map** (Mysidia.Standard.Collection.Map) class, which defines some methods such as *containsKey(key)*, *get(key)*, *keys()*, *removeKey()*, *values()* for basic map operations. The keys are a unique set, which are used to access the elements(values) in a Map. The keys can be any objects, though integers and strings are most common. In order to use an object as key, the object must be able to understand the equality message == *obj*. Failing to satisfy this criterion will lead to compile time error which will halt the program from running. *Map* has a concrete subclass **Dictionary** as well as an abstract subclass **MutableMap**. The former has been described in last chapter, while the latter will be discussed next.



The class **HashMap** is the most widely used subclass of *MutableMap*, which is an unordered hash table that parallels to a HashSet with key-value pairs. The keys in Dictionary class are compared by their hashcode values, which will not guarantee order of insertion. A HashMap object can be instantiated by sending message *.new()* to its class for an empty HashMap, as well as message *.new(map)* which creates a HashMap object from another map object. The method *put(key, value)* is used to insert elements into the HashMap, and in fact all subclasses of MutableMap understands the square bracket message [key]= value. Consider the following code:

```
var hm = HashMap.new();
//create an empty HashMap
hm.put(key: "US", value: "United States");
hm.put(key: "JP", value: "Japan");
hm["RU"]= "Russia"; //sending square bracket assignment message [key]= value
var hm2 = HashMap.new(["US": "United States", "RU": "Russia", "JP": "Japan"]);
//create a HashMap from a Dictionary.
Transcript.show(hm == hm2);
```

The transcript will print *true* on the screen, the two dictionaries are identical, implying that the order of entries are not preserved for HashMap(similar to Dictionary). The following code snippet demonstrate how dictionary objects respond to common Map-related messages *.contains(value)*, *.containsKey(key)*, *.keys()*, *.values()* *remove(value)*, and *removeKey(key)*:

```
var hm = HashMap.new(["US": "United States", "RU": "Russia", "JP": "Japan"]);
Transcript.show(hm.contains("United States"));
Transcript.show(hm.containsKey("CN"));
Transcript.show(hm.keys());
Transcript.show(hm.values());
Transcript.show(hm.remove("Japan"));
Transcript.show(hm.removeKey("RU"));
```

The following results will be printed on the screen:

```
True
False
```

```
JP, RU, US
Japan, Russia, United States
RU: Russia, US: United States
US: United States
```

The class **LinkedMap** parallels with the LinkedSet class whose elements are sorted with the order of addition. If a LinkedMap is created with the order from last example(US -> RU -> JP), the elements will preserve this order for printing and iteration(will be explained in next section). The other class **SortedMap** on the other hand, is similar to SortedList and SortedSet that it is possible to pass a custom comparator object who will sort the elements in arbitrary order, though it is necessary to specify whether the sort mechanism is applied to keys, values or both(the entries).

Enumeration

Since everything in Mysidia happens by objects sending messages to each other, it does not have language support for Control Flows. The if conditional statement is instead handled by sending messages to boolean objects, and for loop structure is replaced by sending messages to number objects. Similarly, there is no foreach loop structure in Mysidia, and **Enumeration** is here for the rescue.

The most fundamental enumeration messages collection objects can understand are the iteration messages. Iterating over collections can be achieved by sending message *each: closure* to these collection objects. Subclasses of *Map* also understands two more iteration messages *eachKey: closure* and *eachValue: closure*. The following code snippet describes how this can be done in Mysidia, note the dictionary iteration shows order of addition is not preserved.

```
var array = [1, 2, 3, 4];
array each: (value){ Transcript.show(2 * value) };
// prints: 2 4 6 8
var dict = ["US": "United States", "RU": "Russia", "JP": "Japan"];
dict each: (key, value){
    Transcript.show("${key} => ${value.toLowerCase()}; ");
};
// prints: JP => japan; RU => russia; US => united states;
dict eachKey: (key){ Transcript.show(dict[key]) }
// prints: Japan Russia United States
```

There are other variants of iteration messages that collection objects can understand, such as *eachReverse: closure*, *each: closure without: compareClosure* and *each: closure separatedBy: separatorValue*. Note the last two messages are sent using secondary keyword selectors similar to *ifTrue: else:*, which are recommended to use for most iteration messages as they usually do not return a meaningful value, and may span multiple lines. Examples are given below:

```
var array = [1, 2, 3, 4];
array eachReverse: (value){ Transcript.show(value) };
// prints: 4 3 2 1
array each: (value){ Transcript.show(value) } without: value => value.isOdd();
// prints: 2 4
arrya each: (value){ Transcript.show(value) } separatedBy: " -> ";
// prints 1 -> 2 -> 3 -> 4
```

Other than iterations, Mysidia's collections also has methods that allow selective enumerations for the elements inside. The method *collect(closure)* evaluates the passed closure for each element, resulting in a new collection object with the returned object from the closure. The methods *select(closure)* and *reject(closure)* will result in new collection objects that matches the selection or rejection criteria, which are usually boolean objects returned from the closures. The *detect(closure)* method is similar to *select(closure)* but will return the first element found in the collection, instead of all elements that match selection criteria. There are also methods *inject(closure)* and *inject: base with: closure* methods used to combine elements in the collection objects. Examples of usage of these methods are shown below:

```
var list = ArrayList.new([1, 2, 3, 4]);
list.add(5);
Transcript.show(list.collect(i => i * 2));
Transcript.show(list.select(i => i.isEven()));
Transcript.show(list.reject(i => i.isEven()));
Transcript.show(list.detect(i => i.isEven()));
Transcript.show(list.inject((sum, i) => sum + i));
Transcript.show(list inject: 10 with: (sum, i) => sum + i));
```

Run the command prompt with this program, and the following results will be printed on the screen. Note these collection enumeration messages will result in new collection objects returned, they will not modify the original collection even if they are mutable. The returned collection objects share the same classes as the receiver object classes, which make their behaviors consistent and logical(ie. in the above example, all objects returned from these methods are also ArrayList objects).

```
2 4 6 8 10
2 4
1 3 5
2
15
25
```

There are a few more enumeration messages that collection objects can understand. For instance, the message *.count(closure)* can be sent to collection objects to find the number of elements that match the criteria specified in the closure. The message *.satisfiesAny(closure)* and *.satisfiesAll(closure)* can be sent to collection objects to check if the at least one element, or all elements matches the criteria specified in the closure. See the following code snippet for more coverage:

```
var countries = ["Argentina", "Brazil", "Canada", China", "France", "Germany",
"Italy", "Japan", "Mexico", "Netherlands", "Portugal", "Russia", "Spain", "United
Kingdom", "United States"];
Transcript.show(countries.count(country => country.startWith("C")));
// prints 2
Transcript.show(countries.satisfiesAny(country => country.startWith("United")));
// prints true
Transcript.show(countries.satisfiesAll(country => country.startWith("United")));
// prints false
```

Summary

Collection Classes are an integral part of the Mysidia programming language. They usually represent commonly used data structures for groups of elements, and proper usage of these classes are vital to write elegant and powerful code in Mysidia. All of these Classes are subclasses of the abstract class Collection, and there are different subclasses of Collection Classes that can be chosen for various use cases. Other than basic access and mutation(for mutable collections) operations, it is also possible to enumerate elements with supplied criteria. Note all literal collections from last chapter are immutable collections, and the other collections described in this chapter are mutable collections. It is recommended to favor immutable collections over mutable collections, and the reason will be explained shortly in the next chapter about Immutability.

Chapter 11: Immutability and Slots

Introduction

As we have learned from earlier chapters that Mysidia offers immutable collections that can be described in literal notation and for convenient use. There is a very good reason for this design of our standard library, as immutable collections have the merit of being easier to use, test and maintain. The immutable-first principle has greatly influenced the modern software development world, which originated from functional programming languages and has made it way to most mainstream languages. To embrace this trend, Mysidia provides language support for immutability. This chapter will describe how to declare/use immutable variables, as well as a how variables/slots work in general.

Immutability

Immutability is the quality that an object stays unchanged once created and initialized, throughout its lifecycle. We have already seen in the earlier chapter that several objects in Mysidia are immutable, ie. instances of Boolean, Number, String, Array, Dictionary, etc. For immutable objects, messages that attempt to mutate its value will either result in an error, or it may return a copy of itself instead. Immutable objects are easier to understand and reason, they are also inherently thread-safe. It is very beneficial to work with immutable objects, which helps with debugging and maintenance.

Since an object has instance fields, these instance fields may in term have their own sets of instance fields, which are all objects. If an object as well as all of its direct and indirect instance fields are all immutable, the object is regarded as **Strongly Immutable**. The basic immutable objects such as booleans, numbers and characters are all strongly immutable. String objects are also strongly immutable, as it contains character elements which are immutable.

However, if the object itself is immutable, but at least one direct or indirect instance field is mutable, the object is considered **Weakly Immutable**. Good examples of Weakly Immutable objects are immutable collection objects such as Arrays and Dictionaries. We cannot modify/reassign the element inside an array/dictionary, but these elements may be mutable elements. For instance, an array may contain elements such as Vehicle and Animal objects which are mutable objects.

Both strongly and weakly immutable objects have their usefulness, the former is especially suitable for small and simple classes, while the latter works better for big objects that are memory-expensive. But no matter whether we wish to achieve strong or weak immutability, the very first step is to make sure that the variable referencing the value of an object cannot be re-assigned. Unlike Smalltalk, Mysidia provides language support for Immutable Variables, as we shall see in next section.

Immutable Variables

So far, all the variables declared in the previous chapters are mutable. When we send the tertiary message *var a* to the current executing context, it will declare a mutable variable named a. The value of a can be reassigned anytime, as we have seen in the code examples. Albeit flexible, mutable variables are vulernable to accidental assignment bugs, also they are harder to reason about their correctness as the values may be changed anytime inside a method. Testing a method can be frustrating if it uses mutable variables and actively reassign them to unpredictable values.

Fortunately, it is possible to declare variables as immutable in Mysidia. To achieve this, we will send a different tertiary message *.val(name)* or *.val(name, value: object)* to the implicit variable *thisContext*, which declares an immutable variable with the supplied name(and value in the latter variant). Mysidia also offers a syntactic sugar *val name* and *val name = value* which is equivalent to the above tertiary messages. Once initialized, this immutable variable cannot be reassigned, which will result in a compile time error as shown in the below sample program:

```
var a = 5;
a = 6 //ok
val b = 3;
b = 2 //Error: Immutable variable b cannot be reassigned.
val c;
c = 10; //No Error
c = 20; //Error: Immutable variable c cannot be reassigned.
```

As we see, reassigning the mutable variable a works as we've seen before, while the immutable variable b cannot be reassigned. It is also interesting that declaring immutable variables without assignment will work, unlike many other programming languages. This variable will have a null value before initialization, and can be assigned to any objects(literal or another variable, as well as null isself). However, this same variable can never be re-assigned again once initialized. It is still recommended to initialize an immutable variable immediately as it is declared, which is optimized by the VM for better performance, and also more maintainable and testable.

Behind the scene variable objects are are all refied as instances of Slots, and the class *NamedSlot* defines behaviors for all immutable variables. Immutable variables in Mysidia are weakly immutable, since their internal states may be mutable despite the fact that they cannot be reassigned to other values. This is evident from the example of mutable ArrayList object declared with *val*, which prevents reassigning this variable al to a different object, but will not forbid inserting/replacing elements inside the ArrayList object:

```
val al = ArrayList.new(5);
al[0] = 4; //works fine
al.add("Welcome"); //works fine
al.remove(4); //works fine too
al = HashMap.new(); //error: cannot reassign immutable variable al
```

In order to create strongly immutable object, it is necessary to make sure that all the direct and indirect instance fields of this object are also immutable. Fortunately, Mysidia has a special syntax to declare immutable instance fields with the syntactic sugar *val*. It is possible to declare both immutable and mutable fields, and they will need to be defined separately with the latter using *var*. The below program uses the *Car* class example from previous chapters:

```
class Car {
    fields: {
        val model, color;
        var speed;
    }

    methods: {
        init(model, color, speed){
            this.model = model;
            this.color = color;
            this.speed = speed;
        }

        model() => model;
        color() => color;
        speed() => speed;
        setSpeed(speed) => this.speed = speed;
        accelerate(dSpeed) => this.speed += dSpeed;
        isMoving() => this.speed > 0;
        stop() => this.speed = 0;
    }
}
```

The above class Car is a simplified version without inheritance(all instance fields/methods are directly defined in Car class instead as there is no Vehicle superclass for this example). The instance fields *model* and *color* are immutable as they are declared with *val*, while the other field *speed* is mutable which is declared with *var*. The immutable instance fields have only getters, while the mutable fields may have setters or other related methods that change its value. Instances of class Car assigned to immutable variables are actually weakly immutable due to the presence of mutable field *speed*. If this mutable field is removed however, instances of Car objects can be strongly immutable. It is a design decision for each developer to evaluate the benefits and costs of creating/using strongly immutable variables.

Similar to immutable local variables, immutable instance fields may be initialized later, either in the initialize or a setter method. However, once assigned a value, this immutable instance field can never be reassigned again. For this reason, immutable variables in Mysidia are effectively 'write-once' variables. A good design principle is to initialize all immutable instance fields inside the initializer methods, failure to follow this practice may lead to fragile and unmaintainable programs.

Unified Variable System

We have seen that both instance fields and local variables can be declared as immutable, and they share a great deal of similarity. In fact, Mysidia has a **Unified Variable System** such that all variables are one of the same kind, namely *Instance Variables* of certain objects. Instance fields are instance variables of an ordinary object created with classes, method/closure parameters are instance variables of the method/closure argument binding objects, while local variables are instance variables of the method/closure context objects where they are declared. This unified variable system ensures that variables are all similar to each other, and it is very similar to Self and Newspeak's variables system. Note we use the term *Instance Field* to describe instance variables declared inside an ordinary class, while the term *Instance Variable* represents all kinds of variables in Mysidia. It's necessary to understand this distinction in terminonogy to avoid confusion of the text in this language specs.

Developers familiar with dynamically typed languages such as Smalltalk, Python and Ruby may have noticed that Mysidia does not have the concept of class variables or static variables. As everything is an object, classes themselves are also objects and instances of their own classes(aka Metaclass). For this reason, it is unnecessary to implement language support for class variable, which are just instance variables of a class object, declared in a *metaclass*. The next chapter will explore the concept of Metaclass thoroughly, and several differences in the implementation between Mysidia and Smalltalk's metaclass systems will also be discussed.

Similar to Smalltalk, Mysidia code also has global state, which is different from Newspeak which eliminates global state in a proper and innovative way. However, Mysidia has heavy restriction on the usage of global state, as only classes, namespaces, traits and objects representing specially reserved keywords(ie. null, true and false) may be created as global variables. This implies that creation of ordinary object instances as global variables is not possible in Mysidia without reflection and advanced metaprogramming techniques. As these global objects can only be declared by immutable variables with *val*, we know that the names representing global objects cannot be reassigned to a different objects. This prevents accidental reassignment of global objects, but will not prevent the instance fields of classes and other global objects from being changed(the specially reserved objects null, true and false are strongly immutable, others are mostly weakly immutable). It is nonetheless highly recommended to avoid mutating global objects once they are created and properly initialized.

Slots and Instance Variables

Behind the scene, all variables in Mysidia are reified as instances of the **Slot** class. Slots are metaobjects that define how variables may be accessed and manipulated. Once Slot object is instantiated, it associates a symbol with a variable name, and enables access to the newly declared variables by its name. There are a few subclasses of Slot which deal with some unique characteristics of special variables, and will be discussed in details at the next few sections.

The details of slot objects are usually abstracted away from the developers, but sometimes it is convenient to use them explicitly. The class *Slot* (Mysidia.Standard.Lang.Slot) is the abstract superclass that defines all behaviors for different types of slots. The class **NamedSlot** is a concret subclass of *Slot* that represents immutable instance variables, while the subclass **MutabledNamedSlot** represents mutable instance variables. We can get such a slot object simply by sending message *.namedSlot(name)* to a class object. The below code example demonstrates how to obtain a named slot and what messages it understands, which reuses the previous example for Car class:

```
val car = Car.new(model: "BMW", color: "white", speed: 65);
val slot = Car.namedSlot(#model);
Transcript.show(slot.class());
Transcript.show(slot.name());
Transcript.show(slot.read(car));

val slot2 = Car.namedSlot(#speed);
Transcript.show(slot2.class());
Transcript.show(slot2.name());
Transcript.show(slot2.read(car));
```

```
slot2.write(80, to: car);
Transcript.show(slot2.read(car));
```

The above code creates an instance of Car with instance fields initialized. We then create two slot objects by sending message *.namedSlot(name)* to the class Car, which returns the corresponding slot objects represending its instance fields. Instances of NamedSlot understand message *.name()* which returns the name of the instance field, and message *.read(object)* which fetches the value of the instance field from a given object. Instances of MutableNamedSlot also understands message *.write(value, to: object)*, which mutates the value of an instance field. If we run the program, we will get the below text on the Transcript:

```
Class: NamedSlot
#model
BMW
Class: MutableNamedSlot
#speed
65
80
```

As we can see, it is possible to acquire slot objects for instance fields defined inside a class, and use them to access/modify the values of these instance fields for objects of this class. As Mysidia's variables(instance fields, local variables, method parameters, etc) are all instance variables, it is possible to use slot objects for other types of variables too. The below program makes uses of slots for declared local variables inside and outside of a closure:

```
val a = 2;
val slot = thisContext.namedSlot(#a); //obtain a named slot from thisContext(local
variable a).
Transcript.show(slot.name()); //prints #a
Transcript.show(slot.read(thisContext)) //prints 2
val closure = (p){
    val a = p + 3;
    val argumentBinding = thisContext.argumentBinding();
    //obtain an argument binding object from the closure context.
    val slot2 = thisContext.argumentBinding().namedSlot(#p);
    Transcript.show(slot.read(thisContext))
              ;.show(slot2.read(argumentBinding))
              ;.show(slot.read(thisContext.outerContext()));
};
closure(1); //prints 3 1 2
```

The above code snippet clearly shows that every variable in Mysidia is an instance of the class Slot(more precisely, one of Slot's subclasses). We can also instantiate slot objects by sending messages *.new(name)* and *.new(name: slotName, owner: ownerObject)*. The former creates a slot ready to be attached to any owner object(class, metaclass, context, argument binding, etc), while the latter will create a slot and attach it to the owner object immediately. This is a form of metaprogramming and advanced usage of slots and variables will be explored in later chapters.

Mysidia provides language support for explicitly specifying the type of slots. The syntax *#name -> SlotClass* inside the instance fields declaration block will create a slot for the very instance field, as demonstrated in the below example of Car class:

```
class Car {
    fields: {
        model -> NamedSlot, color -> NamedSlot, speed -> MutableNamedSlot
    }
}
```

Here declaring an instance field as NamedSlot is no different from using the *val* syntactic sugar which creates an immutable instance field, while using MutableNamedSlot is equivalent to *var* syntactic sugar that declares a mutable instance field. This seems rather bizarre and redudant, but it can come in handy for advanced metaprogramming in which developers may create user-defined subclasses for Slot.

Other than *NamedSlot*(as well as its subclass *MutableNamedSlot*), there is another subclass *IndexedSlot* that represents special case of indexed instance variables, which are common in collection objects. The next section will introduce indexed slots(indexed instance variables) and how they work in action.

Indexed Instance Variables

The instance variables that we have seen before are all considered *named instance variables*, as they are instances of class **NamedSlot** associated with a specific name/symbol declared in its lexical scope(a class, method, context or argument binding). As evident from the last two chapters, the collection objects in Mysidia have their own unique way of storing elements. Internally, these elements are stored inside one or more *indexed instance variables*(or more precisely, instances of **IndexedSlot** class). This detail is implemented by the VM native language, and abstracted away from client coders. The below code example demonstrates how to use an indexed slot from a collection object:

```
val al = ArrayList.new(5);
al[0] = 1;
al[1] = true;
al[2] = "Mysidia";
val slot = ArrayList.indexedSlot(#elements);
Transcript.show(slot.class()); //prints Class IndexedSlot
Transcript.show(slot.name()); //prints #elements
Transcript.show(slot.read(car)); //prints 1, true, Mysidia, null, null
```

As Mysidia does not provide syntactic sugar for creating indexed instance field like immutable instance field, the only way to define them is to use the Slot declaration syntax sugar from the last chapter, this time with IndexedSlot. For instance, a Car with an indexed instance field *passenger* can be declared with the code below:

```
class Car {
    fields: {
        model -> NamedSlot, color -> NamedSlot,
        speed -> MutableNamedSlot, passengers -> IndexedSlot;
    }
}

val slot = Car.indexedSlot(#passengers); //obtain the indexed slot in car class.
```

It will be possible to use the class Car in similar way as the collection classes, which can achieve more efficient collection operations. However, unless performance is a real concern, it is recommended to simply store such data in a named instance field with collection objects such as Array, ArrayList, Dictionary, etc.

Implicit Variables

Mysidia has a total of 6 **Implicit Variables**, namely *null*, *true*, *false*, *this*, *super* and *thisContext*, which mirrors Smalltalk's implicit variables. They are also referred to as specially reserved keywords, as no other variables(instance field, local variables, method parameters etc) may be named after them. Declaring a variable with these 6 names will be a compile time error in Mysidia. They are also immutable variables that cannot be reassigned to other values.

The 3 simplest implicit variables *null*, *true* and *false* are special constant values representing unique instances from the class UndefinedObject, True and False. Their existence makes comparing objects to the three common values easy and convenient. The other 3 implicit variables *this*, *super* and *thisContext* are not strictly constants as their values vary depending on the enclosing class(for this and super) and method/closure(for thisContext).

The 6 implicit variables are all unique instances of ImplicitSlot, which is a special subclass of Slot and has all the instantiation messages disabled so the methods such as *new(name)* will not be able to create more implicit variables. However, we can still use the slot for the existing implicit variables to inspect them:

```
val thisSlot = Object.implicitSlot(#this);
Transcript.show(thisSlot.class()); //prints Class ImplicitSlot
Transcript.show(thisSlot.name()); //prints #this
Transcript.show(thisSlot.read(2)); //prints 2
Transcript.show(thisSlot.read("Hello World")); //prints "Hello World"

val closure = x => x * 2;
val thisContextSlot = closure.implicitSlot(#thisContext);
Transcript.show(thisContextSlot.class()); //prints Class ImplicitSlot
Transcript.show(thisContextSlot.name()); //prints #thisContext
Transcript.show(thisContextSlot.read(closure)); //prints Object ClosureContext
```

Apparently it is not very useful to send messages to these implicit variable slots, we can easily acquire the information we need in a much easier way. Nonetheless, these slots exist for the sake of consistency of language implementation. Since every variable is an instance of class Slot, this rule applies to even implicit variables. This is Mysidia's Unified Variable System in action, and we can be sure that the claim that everything is an object holds true even for variables.

Summary

Immutability is a concept that has received widespread native support in many different programming languages. In order to embrace this trend and the good rationales behind the immutable-first design approach, Mysidia offers language support for creating immutable variables, making the lives of developers easier and more convenient. Behind the scene, all immutable variables are instance of NamedSlot. In fact, every variable in Mysidia is an instance of Slot(aka its subclasses), and this unified variable system ensures consistency in language design and makes directions of future work clear. Slots are a powerful tool to manipulate variables, and as we shall see in later chapters how Mysidia's gradual typing works with typed slot objects. In the next chapter, we will examine Metaclasses and Anonymous Classes, which will help with understanding the more advanced concepts in Mysidia.

Chapter 12: Metaclasses and Anonymous Classes

Introduction

In the beginning of this language specs, the 10 rules of Mysidia's OO Models are introduced to the reader. At this point however, we have only explored half of these uniformly applied concepts, since the rest can be confusing for newcomers or programmers without background in Smalltalk, Python or Ruby. After delving into the simpler concepts in the past few chapters, it is time to move on to the more advanced OO concepts. A solid understanding of Metaclasses and how to use them in writing code, is crucial for expanding our knowledge about Mysidia's OO Model. We will also discuss the similarities and differences between Mysidia's Metaclass implementation and other languages such as Smalltalk and Python, followed with a detailed description of Anonymous Classes.

Class as Objects

The 1st rule of Mysidia's OO Model states that **everything is an object**. Booleans are objects, numbers are objects, strings are objects, arrays are objects, closures are objects, and even the contexts of methods/closures are objects. Classes are no exception to this, they are objects as well. The implication is that classes can be assigned as variables and passed as arguments in a message. For this reason we can say that classes are First-Class objects in Mysidia, as the below code snippet demonstrates:

```
val d = DateTime;
Transcript.show(d);
// prints Class: DateTime on the screen
```

We can send messages to class objects, which will activate methods to respond to such messages. For instance, the instantiation of an object is effectively sending a message *.new()*(or other messages that contains *new* as first selector keyword) to its corresponding class objects. There are more such messages that class objects can understand, as shown in the examples below:

```
String.new();
DateTime.now();
Color.blue();
Int.max();
Transcript.show("Hello World");
```

The first statement sends message *.new()* to the class String, which returns a new empty string. The second statement sends message *.now()* to the class DateTime, it fetches a DateTime object represending the current time. The third statement sends message *.blue()* to class Color, and in turn acquires a Color object with blue attribute. The fourth statement send message *.max()* to class Int, which returns the maximum integer value. The last statement send message *.show(text)* to class Transcript, it will print the text "Hello World" on the screen. All these examples demonstrate that class objects are no different from other objects when it comes to message passing, they can receive and respond to messages. Thus the 2nd rule **Everything happens by sending messages** is valid for classes as well.

Metaclasses

The 3rd rule for Mysidia's OO Model states that **Every object is an instance of a class**. Since classes themselves are objects, every class must be an instance of another class as well. A class whose instance is another class, is referred to as **Metaclass**. This brings us to the 6th rule of Mysidia's OO Model: **Every class is an instance of a metaclass**. The metaclass of an object, is the class of the object's own class.

In Mysidia, metaclasses are implicit, and automatically created when sending a class creation message, which is similar to Smalltalk's metaclass model. Classes are effectively global variables that can be accessed everywhere in the code, metaclasses are anonymous since it is not possible to reference them directly by a name like we do with classes. However, it is possible to indirectly access them by sending primary message *.class()* or just unary secondary message *class*. The former is preferred to access a metaclass since the precendence rule is easier to follow, while the latter is used in language specs to refer to a metaclass. The following code snippet demonstrates how to gain access to metaclasses.

```
val i = Int.class();
val d = DateTime class;
val s = "Hello World".class().class();
val t = false.metaclass();
Transcript.show(i);.show(d);.show(s);.show(t);
Transcript.show(String.instanceOf(s));
```

The transcript will print *Metaclass: Int class*, *Metaclass: DateTime class*, *Metaclass: String class*, *Metaclass: False class* and true on the screen. Sending primary message *.class()*(or secondary unary message *class*) to a class object will return the class of the class object, which is the corresponding metaclass object. Sending the message *.class()* twice to a non-class object, will also return the corresponding metaclass object(first message returns a class, second message returns metaclass). It is also possible to send message *.metaclass()* to a non-class object, which is effectively the same as sending message *.class()* twice. The last line evaluates to true, implying that classes are indeed instances of their metaclasses(*String* is an instance of *String class*).

Metaclass Hierarchy

The 4th rule for Mysidia's OO Model states that **Every class has a superclass**. The class Int's superclass is Number, the class Number's superclass is Magnitude, and the class magnitude's superclass is Object. Since Metaclasses are also classes, they need to have their corresponding superclasses as well, and of course they do. We can inspect the superclass of a metaclass by sending the message *.superclass()* to a metaclass object, as we do with classes. This can be illustrated by examining the class hierarchy for metaclasses:

```
val meta = String.class();
Transcript.show(meta.superclass());
// prints: Metaclass: List class
Transcript.show(meta.superclass().superclass());
// prints: Metaclass: Collection class
```

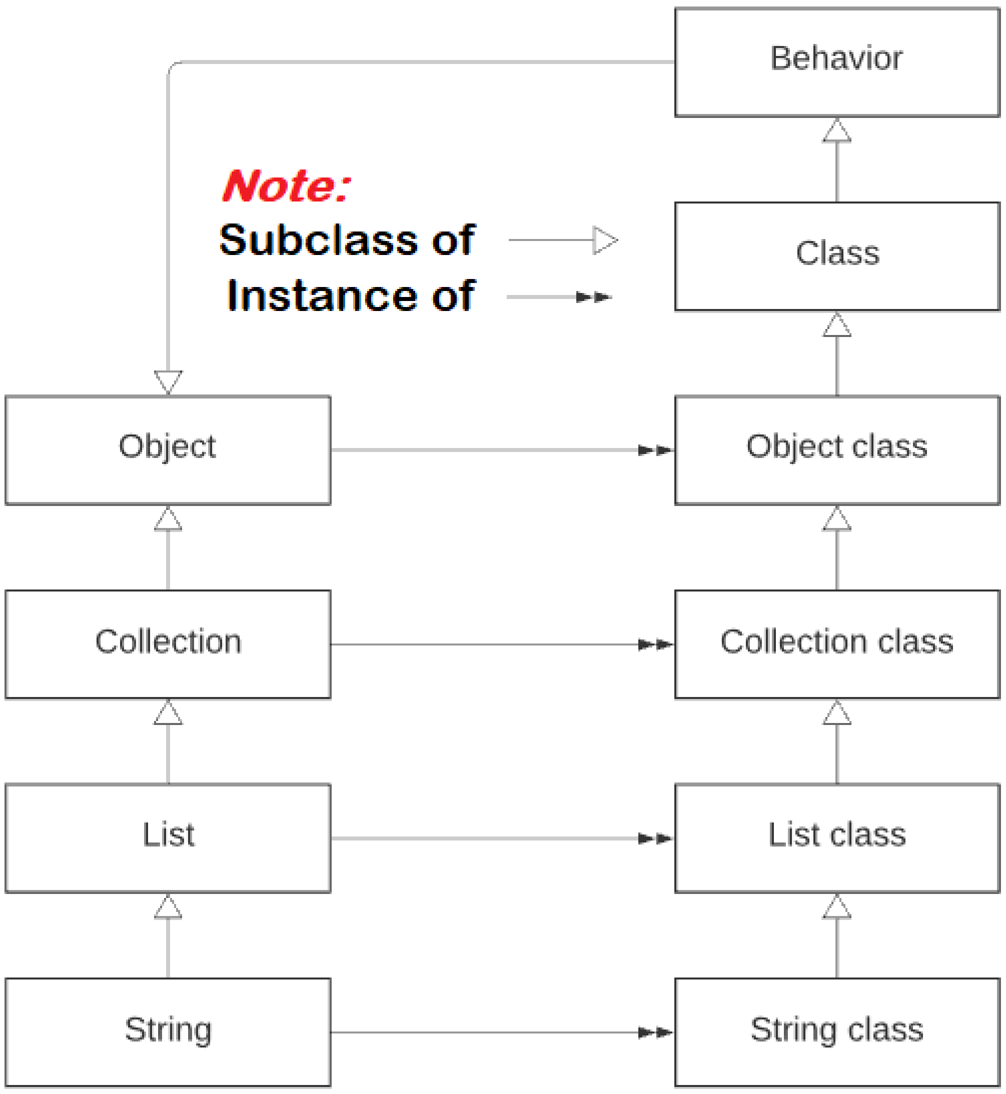
```
Transcript.show(meta.superclass().superclass().superclass());
// prints: Metaclass: Object class
```

This simple and yet informative program tells that the superclass of *String class* is *List class*, the superclass of *List class* is *Collection class*, and the superclass of *Collection class* is *Object class*. This not only confirms that metaclasses have superclasses, but also describes the hierarchy of metaclasses which mirrors their corresponding classes. We can now introduce the 7th rule of Mysidia's OO Model: **The metaclass hierarchy parallels the class hierarchy**. If we know String is a subclass of List, then the metaclass String class must also be a subclass of List class. It is not possible to substitute a metaclass by another for a given class, as metaclasses are produced automatically upon class creation, and it does not provide a method to set the value of a metaclass.

Since every class has a superclass(except for the root Object class), it gives rises to the question of what is the superclass for metaclass *Object class*. We can inspect this information by sending message *.superclass()* again to this metaclass like this:

```
var meta = Object.class();
Transcript.show(meta.superclass());
// prints: Class: Class
Transcript.show(meta.superclass().superclass());
// prints: Class: Behavior
Transcript.show(meta.superclass().superclass().superclass());
// prints: Class: Object
```

Now the interesting thing is, the superclass of the metaclass *Object class* is *Class*, which has a superclass called *Behavior*, and finally *Behavior* inherits from the root *Object* class, forming a complete class Hierarchy. This confirms the 8th rule of Mysidia's OO Model: **Every metaclass inherits from Class or Behavior**. The abstract class *Behavior* is important that it defines an instance method *new()*, which is used to instantiate objects. This method is implemented in the native VM to create the empty object, and then send message *.init()* to this object before returning it. It is possible to override this method in subclasses, and this detail will be discussed in later sections. The below diagram summarizes the metaclass hierarchy in Mysidia:



The class *Behavior* also provides minimum state necessary for objects that have instances, which includes a link to superclass, a method dictionary, a description for the instances, as well as useful methods for creating instances, manipulating class hierarchy, accessing methods, etc. The class *Class* represents common behaviors of all classes, it defines methods to access class names, instance fields, etc. *Behavior* and *Class* are not metaclasses themselves, although their methods are inherited by metaclasses. In contrast with Smalltalk's class hierarchy, there is no *ClassDescription* class in Mysidia, while *Class* inherits directy from *Behavior*.

Instance Fields/Methods with Metaclasses

As explained in the chapter 4 about Classes and Methods, Mysidia objects only have instance fields/methods, but not class fields/methods. However, since classes are objects themselves, they can also have instance fields and methods. We have already seen this in action, when messages like *.new()* is sent to class String, or *.show(text)* is sent to class Transcript. The instance fields/methods on classes are similar to class fields/methods in other programming languages. As instance fields/methods for an object are defined on its class, the instance fields/methods for a class are defined on the metaclass.

One way to achieve this is to declare a metaclass block in the class definition, another way is to define a metaclass outside of its corresponding class but in the same file. The first approach is preferred when the metaclass has only a few fields/methods, while the second approach is more desirable if metaclass has a lot of fields/methods. The syntax for defining metaclass instance fields/methods is shown below:

```
class Fraction {
    metaclass: {
        ... fields and methods for Fraction class inside ...
    }
}

// or:
metaclass Fraction class {
   ... fields and methods for Fraction class inside ...
}
```

To illustrate how to define fields and methods for class objects using metaclass, we will reuse the Fraction class example from Chapter 4. As everyone has learned in school, a Fraction can reduce itself to the simplest form without common factor between the numerator and demonimator. Also if the numerator is multiple of denominator, it may simply become an integer. The current implementation of class Fraction offers no such way to achieve this, but the new Fraction class below provides options to reduce Fraction by using metaclass *Fraction class*:

```
class Fraction {
    fields: {
        numerator, denominator
    }

    methods: {
        ... methods from chapter 4...

        reduce(){
            val gcd = numerator.gcd(denominator);
            numerator /= gcd;
            denominator /= gcd;
        }

        toString(){
            this.class().isReduceable() ifTrue: { this.reduce() }
            return "${numerator.toString()} / ${denominator.toString()}";
        }
    }

    metaclass: {
        fields: {
            reduceable
        }

        methods: {
            init(){
                reduceable = false;
            }

            isReduceable() => reduceable;

            setReduceable(reduceable){
                this.reduceable = reduceable;
            }
        }
    }
}
```

This modified *Fraction* class has a new method *reduce()* which can simplify a fraction to lowest term when converted to string, it is only performed if the metaclass explicitly specifies a flag *reduceable* to be true. Inside the metaclass block, we specify the flag *reduceable* as its instance field. There are getter and setter methods to access/modify the flag value at will. The *init()* method in metaclass *Fraction class* is a special initializer method for the metaclass to initialize instance fields on the class side. To test our new Fraction class, we can run the following program:

```
val frac = Fraction.new(numerator: 8, denominator: 12);
Transcript.show(frac);
Fraction.setReduceable(true);
Transcript.show(frac);
```

This will print 8/12 and then 2/3 on the screen, which is exactly as we expect. Mysidia classes have no access to the fields defined in their metaclasses, they have to use an accessor methods to obtain these values, assuming their metaclasses provides such methods. Also note methods defined in metaclass will be activated from the outside if a message is sent to the class object, as passing message *.setReduceable(true)* to Fraction class. They are similar to class methods in other languages, except that they are instance methods on the class object in Mysidia.

The metaclass called Metaclass

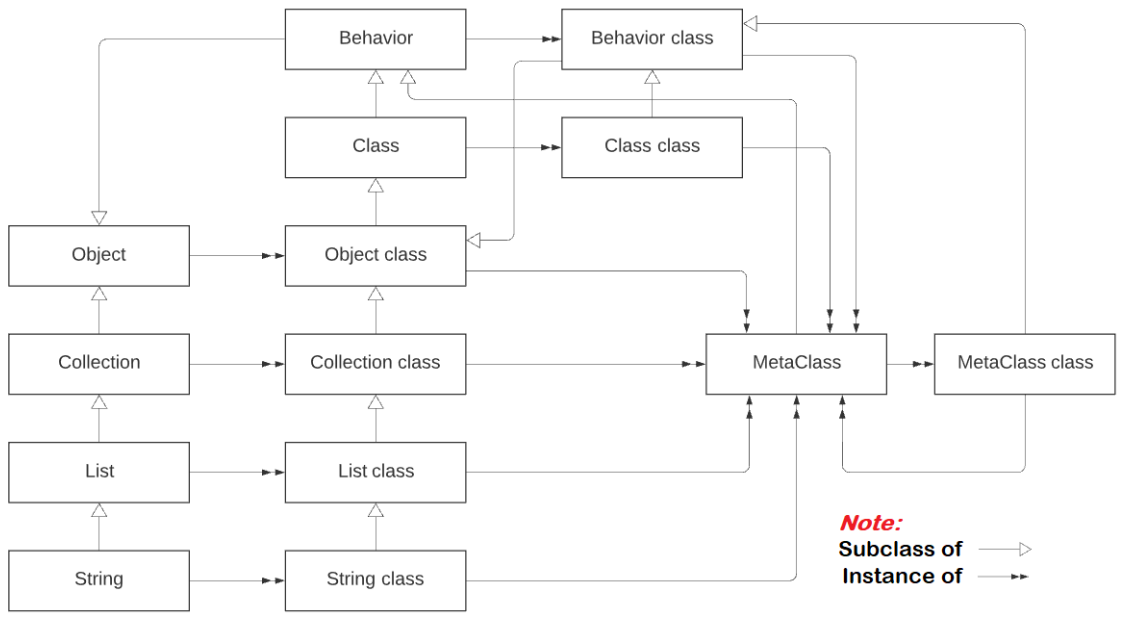
Metaclasses are classes themselves, and classes are objects. It implies that a metaclass should have a class too, which is the metaclass of the class object. In fact, every metaclass in Mysidia shares one common class called **Metaclass** (Mysidia.Standard.Lang.Metaclass), this brings us to the 9th rule of Mysidia's OO Model: **Every metaclass is an instance of Metaclass**. To see more about Metaclass in action, we can use the code snippet below:

```
Transcript.show(Fraction.class())
          ;.show(Fraction.class().class())
          ;.show(Fraction.class().class().class())
          ;.show(Fraction.class().class().class().class())
          ;.show(Fraction.class().class().superclass())
          ;.show(Fraction.class().class().class().superclass());
```

The first line will print *Metaclass: Fraction class* as what already seen in earlier sections. The second line will print *Metaclass: Metaclass*, and we can see clearly that Mysidia metaclasses are indeed instances of the class **Metaclass**. The third line will print *Metaclass: Metaclass class*, this is understandable too since every class has a class, including the special class *Metaclass*. The class *Metaclass* is the metaclass of class objects, which can be accessed by sending message *.metaclass()* to class objects. Similarly, the class *Metaclass class* is also the metaclass of metaclasses, and can be accessed by sending message *.metaclass()* to metaclass objects.

The fourth line is rather interesting, it will print *Metaclass: Metaclass* again like second line. The class of *Metaclass class* is class *Metaclass* itself, *Metaclass class* is both the class and an instance of **Metaclass**, indicating a circular relationship between these two classes. It may seem confusing at first, but it is a logical language implementation decision similar to Smalltalk's metaclass system. Now we can complete the entire Mysidia's OO Model with the 10th rule: **The metaclass of Metaclass is an instance of Metaclass**.

The fifth line on the other hand, will print *Class: Behavior* on the screen, implying that the class **Metaclass** inherits directly from class *Behavior*, rather than class *Class*. This further validates the 8th rule of Mysidia's OO Model, that every metaclass is a subclass of *Class* or *Behavior*. Similarly, the last line prints "Metaclass: Behavior class", as metaclass hierarchy parallels class hierarchy. The use cases for *Metaclass* and *Metaclass class* are rare, nonetheless their existence is essential for the consistency of language implementation. The below diagram depicts a complete metaclass hierarchy in Mysidia:



Anonymous Classes

In Mysidia, an object can have only one class, but a class can usually have many instances, forming a one to many relationship between classes and their instances. It becomes tricky when adding metaclasses into the mix, the other OO languages handle this rather differently. In Smalltalk, classes are singleton objects, the one and only instance of their metaclasses(except when it comes to the special metaclass called

MetaClass). In Python however, classes are not singleton objects, the metaclass Type and its subclasses can have multiple instances.

Although Mysidia's OO Model is most similar to Smalltalk's, it takes a diverging route when it comes to class/metaclass relationship. The metaclasses in Mysidia, can have multiple instances of the corresponding class objects. For instance, the class *Fraction* has only one class *Fraction class*, but the metaclass *Fraction class* has multiple instances including the one called *Fraction*.

Now the question is what are the other instances of metaclasses? The answer lies in the concept of **Anonymous Class**, which does not have a declared class name compared to the named classes we've seen in the spec book such as Object, Int, String, Array, DateTime, etc. An anonymous class can be created by sending the message *.new(fields: fields, methods: methods)* to the metaclass object. The fields and methods are collection objects with keys representing the field names or method selectors. There is also a declarative syntactic sugar for instantiating anonymous class objects similar to object instantiation:

```
Fraction.class().new(fields: [], methods: []);
// or:
Fraction class new{
    fields: { }
    methods: { }
};
```

If we pass fields and methods arguments in message *.new(fields: fields, methods: methods), the anonymous class may end up with additional fields and methods. It may look similar to Java's anonymous class, but Mysidia's anonymous classes are objects and can be assigned as variable, or passed as arguments in messages. If an instantiation message is sent to the anonymous class, an instance of this anonymous class will be created:

```
var anonyFraction = Fraction class new{
    methods: {
        inverse() => this.class().new(numerator: 5, denominator: 6);
    }
};

val frac = anonyFraction.new(numerator: 5, denominator: 6);
Transcript.show(frac);
val frac2 = frac.inverse();
Transcript.show(frac2);
Transcript.show(anonyFraction);
Transcript.show(anonyFraction.superclass());
Transcript.show(anonyFraction.class());
Transcript.show(frac.instanceof(Fraction));
```

Upon running the above program, the following lines will be printed by the Transcript. The first two lines are as what we expect, note the method *inverse()* was not defined in the named class Fraction. The anonymous class is effectively used to extend the functionality of the original class by adding its own instance fields and methods, while retaining all the fields and methods available from the original class.

```
5/6
6/5
Class: Fraction$1
Class: Fraction
Metaclass: Fraction class
true
```

The third line displays the 'name' of the anonymous class, which is randomly assigned at runtime and cannot be referenced. The fourth line implies that the superclass of an anonymous class, is effectively the original named class from which it extends. This actually makes perfect sense, as the anonymous classes in Java behave this way too. When we create an anonymous class, we creates a subclass of the named class it is based on. For this reason, anonymous class inherits all fields and methods from the original class. Anonymous classes can therefore be used in mock object test about abstract classes.

The fifth line, on the other hand, shows that anonymous classes share the same metaclass as the named class, instead of having its own metaclass(the class for both Fraction and the anonymous Fraction is metaclass *Fraction class*). This is a deviation from Smalltalk's metaclass model, as Mysidia's classes are not singleton objects created by the metaclasses. The metaclasses can have multiple instances, with the named class as its unique instance(also called companion instance), and anonymous classes that inherit from this named instance of metaclass.

Since anonymous classes are subclasses of the original named class, sending message *.instanceof(namedClass)* will return true. Note we are using *this.class()* in the inverse() to access the appropriate anonymous subclass instead of the original class where the method is defined. If it uses *Fraction.new()* instead of *this.class().new()*, it will return a Fraction object, rather than the anonymous Fraction object. This is necessary as anonymous classes do not have names and can only be referenced by sending message *.class()* to instances of the anonymous class. This is similar to how we can gain access to metaclasses, as metaclasses are all anonymous and can only be referenced if we have its named instance class.

As we've learned from chapter 6 about traits, it is possible to create objects with dynamic traits. Behind the scenes, Mysidia will create anonymous classes for objects instantiated with dynamic traits. It can be argued that we have been using anonynous classes without realizing they exist back then. This also explains why methods defined in instance traits are activated before methods in the original class, as anonymous classes are subclasses of the named class. Adding anonymous classes into the mix, our completed method lookup mechanism follows the chain of Anonymous (Sub)Classes -> Instance Traits -> Classes -> Class Traits -> SuperClasses.

Initializers and Message Interception

We can send message *.new()* to this class object to create its instances. The only explicitly defined instantiation method *new()* is found at the abstract class Behavior, which instantiates an empty object and send message *.init()* to this instance before returning it. No other *new* methods that take arguments are defined anywhere, but they do exist. Behind the scene Mysidia automatically creates the corresponding *new* methods based on the selector of these *init* methods found in the class. For instance, as the method *init(numerator, denominator)* is found in the class *Fraction*, a method *new(numerator, denominator)* is automatically created in metaclass *Fraction class*. This is why we can safely send message *.new(numerator: 5, denominator: 6)* to the class

Fraction to instantiate objects, without having to define this *new* method on its metaclass.

Nonetheless, it is possible to explicitly define such a new method on the metaclass in Mysidia. In this case, this explicitly defined new method will replace the implicitly created new method, and it will be activated upon receive the matching new message. For instance, we can add instance counter for class *Fraction*, by implementing such a new method explictly on metaclass *Fraction class*:

```
class Fraction {
    metaclass: {
        fields: {
            reduce, counter
        }

        methods: {
            init(){
                reduce = false;
                counter = 0;
            }

            new(numerator, denominator){
                var instance = this.new();
                instance.init(numerator: numerator, denominator: denominator);
                counter += 1;
                Transcript.show("Creating instance: " + counter.toString());
                return instance;
            }
        }
        ... other methods for Fraction class below ...
    }
}

val frac = Fraction.new(numerator: 8, denominator: 12);
// prints Creating instance: 1
val frac2 = Fraction.new(numerator: 5, denominator: 6);
// prints Creating instance: 2
```

As can be seen from the program above, an explicit new method *new(numerator, denominator)* is defined inside the metaclass *Fraction class*, which matches the selector of init method in class *Fraction*. The creation of instance does not only initialize a Fraction object, but also increment the counter instance field, and prints this value from Transcript. This is the simplest form of **Message Interception**, in which a method that is supposed to be activated in response to a message is intercepted, and decorated/replaced by another method. Another example of Message Interception is to implement method *messageNotUnderstood*. Message interception is a powerful tool for metaprogramming in Mysidia, and more about this black magic will be explored in later chapters.

Summary

Although metaclasses may appear as some kind of deep black magic, they are actually not that complex and enjoy wide applicability in Mysidia programs. They are anonymous and automatically created when a class is defined. The hierarchy of metaclasses mirrors that of classes, and every metaclass is an instance of a special metaclass called *Metaclass*. The introduction of metaclasses complete the other half of Mysidia's OO Model, and the developers can use them to implement instance fields/methods on class objects, which can be compared to class/static methods in other programming languages. Advanced use cases of metaclasses include creating anonymous classes and message interception, the latter will be explained in more detail in later chapters.

Chapter 13: Type System and Type Hinting

Introduction

So far we have introduced the Dynamic Typing side of Mysidia Programming Language. There is no type annotation appearing anywhere in the instance fields and methods, just like what Smalltalk, Python and Ruby are. This is not to say that it is a full dynamically typed language. In fact, Mysidia offers **Gradual Typing** as one of the most essential features of the language, which means that it supports both dynamic and static typing. Every programmer is free to choose between the two ways when writing their programs, achieving great flexibility for the use cases where either typing system finds its strength in. The implementation about Mysidia's type system will be discussed throughout this chapter, along with a detailed coverage of type hinting with classes, metaclasses and traits.

Type System

Ever since the emergence of type systems decades ago, there has been ongoing debate of static typing vs dynamic typing. The former is found in most of the compiled lanuguages such as C++, Java, C#, etc, while the latter is more widely used in scripting languages such as Smalltalk, Python, Ruby, etc. Both statically and dynamically typed systems have advantages and disadvantages, and there is usually a tradeoff when choosing one over the other. In recent years a new category of programming languages have emerged to support both static and dynamic typing, which is referred to as gradual typing, the Mysidia Programming Language is one of such languages.

The code presented in the previous chapters are all dynamically typed, which means that the variables(instance fields, method parameters and local variables) may be assigned as objects of any classes. The program is more flexible, at the cost of type safety. For statically typed code however, the compiler will enforce type safety for fields, methods and local variables so that only objects of specific classes can be used. This is usually achieved with type annotation, which places a restriction on the type of objects that can pass the static type checker. In Mysidia, we refer to the type of an object to be the class it belongs to, as well as traits it may use.

As Mysidia supports both static and dynamic typing, developers can easily convert a dynamically typed program into statially typed, and vice versa. This can be done at file level, class level and even method level, consider the following code snippet:

```
// statically typed program
class Program {
    methods: {
        main(Int argc, String argv){
            Transcript.show("Hello World");
        }
    }
}

// dynamically typed program
class Program {
    methods: {
        main(argc, argv){
            Transcript.show("Hello World");
        }
    }
}
```

The dynamically typed program is the same code from chapter 1, while statically typed program introduces type annotations to the method main. In this example, the argument *argc* must be an instance of Int class, while the argument *argv* must be an instance of class String. Failing to comply to this requirement will result in Type Error that halts program execution. For this reason, statically typed code has better type safety than dynamically typed code. There are circumstances when this type safety is essential, and in other occasions when it is unimportant and can get in the way of flexibility of dynamic typing. It is up to the developers to choose when to use static typing over dynamic typing, and vice versa.

Other than static and dynamic typing, another group of type system is strong and weak typing. In a strongly typed language, a variable is assigned by an object with a strict type/class that cannot be changed upon creation. Although in dynamically typed languages it is possible to change this object an object with a different type, this must be done explicit conversion or reassignment. In a weakly typed language however, a variable is assigned by an object with ambiguous type that may be implicitly converted to another type depending on context.

For instance, the expression *2 == "2"* is false in strongly typed languages, but true in weakly typed languages as it implicitly converts "2" to 2 when doing comparison. This type juggling behavior is convenient for beginners/noncoders to get started without worrying about type details, but usually is a hindrance for writing maintainable and bug free code. For this reason, most programming languages are strongly typed, including Java, C#, Smalltalk, Python, etc. The beginner-friendly languages like Javascript and PHP are weakly typed, which is suitable for HTML designers to embed scripts into a dynamic page.

Mysidia is a strongly typed language, it does not allow implicit conversion of types, which makes it a little harder to get start with compared with weakly typed languages such as PHP and Javascript, but the improvement in maintenability is worth this minor cost. One common misconception about type system is to associate static typing with strong typing, and dynamic typing with weak typing. A language can be statically and weakly typed, ie. C, while a language can be dynamically typed and strongly typed, ie. Smalltalk, Python and Ruby. To summarize, Mysidia is a gradually and strongly typed programming language.

Type Hinting

As Mysidia is a gradually typed programming language that supports static typing, we can achieve this with type annotations. The program in last section provides a basic example for typing method parameters, though it is possible to add type annotations to both instance fields and methods definition. This is termed **Type Hinting**, as the compiler uses type annotations to perform static type check and ensure type safety. Consider the same Point class from chapter 1 rewritten with type hinting:

```
class Point{
    fields: {
        Number x, Number y
    }

    methods: {
        init(){
            x = 0;
            y = 0;
        }

        init(Number x, Number y){
            this.x = x;
            this.y = y;
        }

        Number x() => x;
        Number y() => y;

        move(Number dx, Number dy){
            x += dx;
            y += dy;
        }

        Number distance(Point point){
            val dx = x - point.x();
            val dy = y - point.y();
            return (dx.square() + dy.square()).sqrt();
        }

        String toString() => "(${x}, ${y})";
    }
}
```

In the above code snippet, type hinting is present for both instance fields and methods. For instance field type hinting, the type annotation is positioned to the left of a field name. The syntax follows the format *field-type field-name*, which declares the field to be an object that match the declared type. In this case, both instance fields x and y are typed as class *Number*, so their values must be Number objects. An object matches a type if it belongs to the same class or its subclasses. For instance, the fields x and y can be Int, Float or whatever subclasses that inherit from the abstract Number class.

It is a common practice to use abstract classes(and traits as we will see later) for type hinting, as it makes a perfect balance of type safety and flexibility. If an object does not return a value, it implicitly returns this, thus eliminating the need for void return type popular in some statically typed languages. In such circumstances, it is a common practice to not specify the return type, as evident in the initializer and move methods from the statically typed Point class.

For instance methods type hinting, it is possible to specify both the return types and the argument types. The type annotation for return type is position to the left of method selector, while for argument type is placed to the left of each argument name. The syntax follows the format *return-type methodname(arg-type arg, arg-type2 arg2, ...)*, and developers can choose to omit type annotations for return values as well as any arguments. For instance, the method definition *Number distance(Point point)* has selector distance(point) that accepts only objects of *Point* class as argument, and shall only return an object of Number class. With type hinting on methods, the following code are invalid:

```
Point.new(x: true, y: 10)
// Type Error: Cannot pass object of True class as argument x, it expects object
Number.

Point.new(x: 2, y: 3).move(4.5);
// Type Error: Cannot pass object of Float class as argument, it expects object Point.

class InvalidPoint extends Point {
    methods: {
        String toString() => [x, y];
    }
}
// Type Error: Cannot return object Array in method toString(), it expects object
String.
```

It is not uncommon for only 1-2 arguments to be typehinted while the remaining arguments are not. This is the power of Mysidia's *Gradual Typing*, as developers can choose between static and dynamic typing at the anywhere in a Mysidia program. We can use type hinting for everything(fully static program), something(mix of static and dynamic program), or not at all(fully dynamic program). These type errors can be caught either at compile time by the static type checker when the compiler can safely determine the types of return values and arguments, or at run time dynamically when the violation is not obvious.

For immutable variables, Mysidia can infer their types from the line where they are assigned. For instance, in the method *distance(point)*, the type for local variables dx is safely inferred to be Number, as the compiler is able to tell that x - point.x() should return a Number object. If the return type for method *x()* is unspecified however, the type will not be inferred and static type checker will completely ignore it. Mysidia performs special optimizations for immutable variables, as their values cannot be reassigned and thus type inference is easy. This is yet one more reason to use immutable local variables in methods/closures.

Typed Slot

As we have learned from Chapter 11, that every variable in Mysidia is in fact an instance variable, and they are instances of the class Slot. The same holds true for typed instance fields and method parameters. In fact, the class Slot has an instance field called *type* which may contain information about the required type of the variable held inside the slot object. Such a slot is referred to as **TypedSlot**, and it is used heavily in Mysidia's standard library. For instance, the class String has a typed instance field called *length*, and we can inspect this by sending message *.type()* to a slot object:

```
val lengthSlot = String.namedSlot(#length);
Transcript.show(lengthSlot.class()) //prints: NamedSlot
Transcript.show(lengthSlot.name()) //prints: length
Transcript.show(lengthSlot.type()) //prints: Class Int
```

As we can see, the required type for the instance field *length* of class String is Int. This information may seem irrelevant for us since length is an immutable instance field, but it helps with compiler optimization as immutable variables can be optimized for better performance. For mutable instance field, this required type will prevent accidental assignment of objects from invalid types.

We can create a typed slot object simply by sending message *.new(name, owner: ownerObject, type: typeObject)* to any concret subclasses of Slot, which can be used to dynamically add typed variables to objects. Note the argument typeObject for keyword *type:* will be ignored if the owner object is a context object. For this reason, Mysidia does not support type hinting for local variables. This language design decision does not only prevent verbose explicit local variable type declaration, but also promote using immutable variables whose type will be inferred by compiler and makes type declaration completely unnecessary(immutable variables are inherently type-safe).

Type Objects

In Mysidia, **Type Objects** refer to any objects that may be used as type annotation/type hinting parameter. A type object must use the a special trait named **TType**, which is a unique trait object that defines the basic behaviors that qualifies an object for type hinting. Classes, Metaclasses as well as Traits all use the trait TType, and thus can be used as type annotation for instance fields, method parameters and return values.

The below sample program uses the metaclass for the Number class from last chapter's example for type hinting:

```
class NumberLogger {
    methods: {
        (Number class) generateNumberClass((Number class) numberClass){
            Transcript.show("Creating number from class: " + numberClass);
            return numberClass.new();
        }
    }
}
```

Note the type annotation for the metaclass *Fraction class* needs to be wrapped in parenthesis to avoid ambiguity. Any instance of metaclass Number class(the class Number's concrete subclasses, named or anonymous) may be passed as argument or returned from the method *generateNumberClass(numberClass)*.

Developers can also create user defined classes that use the trait *TType*, which enables their instances to be used in type hinting. In order for such an instance to be useful in type hinting, it must be able to be referenced anywhere in the program. Fortunately, Mysidia does allow any objects that uses TType trait(as well as Namespace objects) to be declared as global immutable variables, similar to classes, namespaces and traits. More info about this advanced metaprogramming technique will be discussed in later chapters.

It is also worth noting that Mysidia's type system is nominal, which is based on the name of type objects only. Some programming languages like Racket supports structural type system, which can be more expressive than the nominal type system in Mysidia. However, structural type system does not work well with gradual typing, which can be extremely slow and resource-heavy. For this reason, Mysidia's nominal type system is justified to be the right choice.

Similar to class objects and metaclass objects, trait objects also use the special trait called TType, and thus can be used as type annotation. The next section will explore type hinting with traits in details.

Subtyping with Traits

In the previous sections we've seen that typehinting works for not only the given classes, but also for the subclasses of the specified classes. This is a common approach to restrict the classes that can be used for type hinting using inheritance. However, inheritance based type hinting has a drawback of tight-coupling between the classes in the hierarchy. This can be an issue when more flexibility is needed, as it can happen frequently that a method should accept an object whose class cannot inherit from a superclass by some reason. For instance, an Aquarium should be able to accept animals who can swim. If it only accepts fish, it will be able to take trout, salmon, shark, etc, but miss out other marine animals such as turtle, penguin and whale. Consider the following example:

```
namespace Demo.Animal;
using Mysidia.Standard.Collection.ArrayList;

abstract class Fish extends Animal {
    methods: {
        swim(){
            Transcript.show("The fish ${this.class().shortName()} is swimming.");
        }
    }
}

class Trout extends Fish { }
class Salmon extends Fish { }
class Shark extends Fish { }
class Turtle extends Reptile { }

class Aquarium {

    fields: {
        val List species
    }

    methods: {
        init() {
            species = ArrayList.new();
        }

        accept(Fish fish){
            fish.swim();
            species.add(fish);
        }

        main(argc, argv){
            val aquarium = Aquarium.new();
            aquarium.accept(Trout.new());
            // prints -> The fish Trout is swimming.
            aquarium.accept(Salmon.new());
            // prints -> The fish Salmon is swimming.
            aquarium.accept(Shark.new());
            // prints -> The fish Shark is swimming.
            aquarium.accept(Turtle.new());
            // Type Error: Cannot pass object of Turtle class as argument, it expects
Fish.
        }
    }
}
```

Apparently the aquarium cannot accept non-fish marine animals, which imposes a strange constraint on the diversity of its collection. Fortunately, traits can leverage this problem and allow loose coupling. As explained in last section, traits are considered *Type objects* similar to Classes and MetaClasses(which all use the special trait *TType*). For this reason, traits can be used in type hinting as well. If a trait is used for type annotation, the corresponding variable(instance field, method parameter or return value) may be any class that uses this very trait, regardless of what superclasses it may inherit. With this in mind, we can modify the previous aquarium program to accept any marine animals(animal objects that uses trait TMarine):

```
namespace Demo.Animal;
using Mysidia.Standard.Collection.ArrayList;

trait TMarine {
    methods: {
        swim(){
            Transcript.show("The marine animal ${this.class().shortName()} is
swimming");
        }
    }
}

abstract class Fish with TMarine { }
class Shark extends Fish { }
class Turtle extends Reptile with TMarine { }
class Penguin extends Bird with TMarine { }
class Whale extends Mammal with TMarine { }

class Aquarium {
    fields: {
        val List species
    }

    methods: {
        init() {
            species = ArrayList.new();
        }
        accept(TMarine marine){
            marine.swim();
            species.add(marine);
        }
    }
}

main{
    val aquarium = Aquarium.new();
    aquarium.accept(Shark.new());
    // prints -> The marine animal Shark is swimming.
    aquarium.accept(Turtle.new());
    // prints -> The marine animal Turtle is swimming.
    aquarium.accept(Penguin.new());
    // prints -> The marine animal Penguin is swimming.
    aquarium.accept(Whale.new());
    // prints: The marine animal Whale is swimming.
}
```

As we can see, the new aquarium is able to accept any species if it uses trait TMarine. This is referred to as **Subtyping**, and the classes that uses a trait are considered *subtypes* of this very trait. A class is a *subtype* of its superclass, as well as the traits it uses. Subtyping with traits is far more flexible than subclassing with classes, as it allows type hinting to work with not only objects that belong to the same class hierarchy, but also unrelated objects that share similar behaviors. Note traits can have both abstract and concrete methods, while traits with only abstract methods may be referred to as 'interfaces' or 'protocols'.

Summary

Although Mysidia may seem as a dynamically typed language at first, it is in fact a gradually typed language with powerful optional support for static typing. Static typing comes in handy when it comes type hinting for instance fields, method parameters and return values, it helps ensure type safety as well as performance optimization. Mysidia is also a strongly typed language, which does not allow implicit type conversion like some weakly typed languages do. These typed variables are simply typed slot objects behind the scene. Using trait objects for subtyping is especially helpful, as it achieves type safety as well as flexibility and loose-coupling as long as they all share the specified trait. More about Mysidia's type system will be explored in the next chapter about Polymorphism and Generic Types.

Chapter 14: Polymorphism and Generics

Introduction

Object Oriented Programming has 4 fundamental features: Encapsulation, Inheritance, Polymorphism and Composition. The first and second concepts have been covered in chapter 4 and 5 respectfully, and the next in line is **Polymorphism**. As Mysidia is a gradually typed language, it supports different types of polymorphism. In fact, the concept of subtyping is one category of polymorphism, as already introduced alongside with Mysidia's type system earlier. Another category of polymorphism however, requires language support for **Generics**. This chapter will explain the concept of Polymorphism, and how Generics works in Mysidia Programming Language.

Polymorphism

The most widely accepted definition for **Polymorphism** in OOP is the ability for a programming language to process objects different depending on its classes/types. Though debatable, there are usually 4 categories of polymorphism that a language may support:

1. Row Polymorphism
2. Inclusion Polymorphism
3. Ad hoc Polymorphism
4. Parametric Polymorphism

Row Polymorphism is most common in dynamically typed languages, also called duck typing. It does not care about the actual type of an object, and methods may accept any arbitrary types of objects as arguments. Instead, it focuses on the behavior of an object, and the object's suitability to pass as an argument to a method is determined by whether it can respond to certain messages(possessing the corresponding methods). The famous duck test quote states that: If it walks like a duck and it quacks like a duck, then it must be a duck(hence the name duck typing). Since Mysidia supports Dynamic Typing by default, Row Polymorphism is commonly used when writing dynamic code. It offers the most flexibility, but the reliance on behaviors to check types may not be valid sometimes.

Inclusion Polymorphism is usually a feature for statically typed language. It is exactly the same as *Subtyping*, and this topic has been covered in last chapter about Mysidia's type system. Loosely-coupled interfaces make Inclusion Polymorphism more flexible than just using tighly-coupled superclasses. **Ad hoc Polymorphism** is also mostly present in statically typed language, it refers to methods behaving differently according to the types of arguments provided. This is commonly tied to the concept of *Method Overloading*, in which method with same arguments but different types are considered different methods. Unfortunately, Mysidia does not support Ad hoc Polymorphism, as the method selector does not store argument type information.

The last and most intriguing category of Polymorphism is **Parametric Polymorphism**, which allows classes and methods to be written 'generically'. In this way it can handle objects uniformly without depending on their types. Programming languages that support parametric polymorphism usually has special syntax for declaring type parameter, which can be substituted later by a specific class when used. This way the languages become more expressive and still manage to maintain type safety. Parametric Polymorphism is also called **Generics**, and Mysidia has language support for this feature which will be discussed in the later sections.

Generics

Generics is a widely supported feature in a number of maintstream programming languages, ie. C++, Java and C#. A Type parameter is declared alongside with the Type names(Classes, Metaclasses and Trait Objects), the types becomes *Generic types*, and can be used in type hinting. It is also possible for non-generic types to have methods with this type parameter next to their names. These methods become *Generic Methods* and developers can send *Generic Messages* to the receiver objects and activate the corresponding methods. The letter/symbol T is usually used to denote a type parameter, enclosed by a pair of angle brackets <>. The below example demonstrates a simple *Generic Class* for a tree node:

```
class Node<T> {
    fields: {
        T element, Node<T> parent
    }

    methods: {
        init(T element){
            this.element = element;
        }

        init(T element, Node<T> parent){
            this.element = element;
            this.parent = parent;
        }

        T element() => element;
        setElement(T element) => this.element = element;
        Node<T> parent() => parent;
        setParent(Node<T> parent) => this.parent = parent;
    }
}
```

The class Node is a *Generic Class*, and instantiation of this class will require the type parameter T to be substitute by a real type(class, metaclass or trait) such as Int, String, Array, TMarine. Once such an instance is created, the generic information is stored and will replace the type hinting parameter T by the generic type. For instance, a generic Node object will substitute T annotated on its fields and methods by the class String upon creation, and it will not be able to accept Integer or other unrelated classes in the place of T:

```
val node = Node<String>.new("Hello");
Transcript.show(node.element());
// prints: Hello
node.setElement("World");
Transcript.show(node.element());
// prints: World
node.setElement(3);
// Type Error: Cannot pass object of Integer as Argument, String expected.
```

A Type Error will be thrown from the activated method if the message contains arguments whose types do not match the information for generic type parameters. This ensures type safety and enables more flexibility as type T can be specified for each instantiation of the class differently. It is possible to specify Generic Methods in a Non-generic Class, although this practice is not common in Mysidia. The below code snippet shows an example of defining a Generic Method:

```
class NodeFactory {
    methods: {
        Node<T> createNode<T>(T element) => Node<T>.new(element);
        Node<T> createNodes<T>(T element, Node<T> parent) => Node<T>.new(element, 
parent);
    }
}

val node = NodeFactory.new().createNode<String>();
```

Sometimes it may be necessary for a Generic Class/Methods to specify multiple type parameters. This can be achieved by listing the type parameters(ie. T1, T2, ... Tn) in the angle bracket <> separated by comma. A good example is a key-value pair entity called Entry, as shown in the example below:

```
class Entry<K, V> {
    fields: {
        val K key, V value
    }

    methods: {
        init(K key, V value){
            this.key = key;
            this.value = value;
        }

        K key() => key;
        V value() => value;
    }
}
```

The generic type information is stored internally in a map on the VM, and not part of the class name. It implies that Generic and Non-generic Classes cannot share the same namespace, which will cause name collision. Similarly, Generic and Non-generic Methods cannot exist in the same class. In general, the type annotations are not part of the method signature in Mysidia, even for methods arguments and return types. This is why Mysidia does not support ad hoc polymorphism(method overloading) like fully static typed languages such as Java and C#. However, Mysidia's generics is actually reified, as will be discussed in later sections.

Constraints with Generics

The type parameters can be any type objects(which uses trait TType) by default, but it is quite often that we only want to allow certain types to be specified as the type parameters. Mysidia supports constraints on the generic type parameters, which act like rules or instructions to define how the type parameters can interact with the specific generic type objects. To specify a constraint for a type parameter, we can use the syntax assuming T is a class/trait that can inherit another type as in the below example:

```
class Aquarium<T extends Animal>
class Aquarium<T extends Animal with TMarine>
```

This way the type parameter T in class Aquarium<T> can only be substituted by a subclass of Animal(and uses trait TMarine as shown in the second line). If an unrelated type is used when instantiating an object from class Aquarium<T>, it will fail to compile as shown in the following code snippet.

```
abstract class Animal { }
abstract class Fish extends Animal with TMarine { }
class Shark extends Fish { }
class Turtle extends Animal with TMarine { }
class Dog extends Animal { }
class Aquarium<T extends Animal with TMarine> { }

val aquarium = Aquarium<Shark>.new(Shark.new());
val aquarium2 = Aquarium<Turtle>.new(Turtle.new());
val aquarium3 = Aquarium<Dog>.new(Dog.new());
// Type Error: the type parameter T cannot be class Dog, expected type IMarine.
```

The program will complain with a Type Error, since the type parameter T must be a subtype of class Animal, and trait TMarine. Dog is an animal, but not a marine animal, thus cannot be used for generic parameter T. Likewise it will not be possible to use a string in this case, since String is neither a subclass of class Animal, nor a subtype of trait TMarine. As is evident from this example, proper usage of Constraints in Generics will further improve type safety of the code.

Variances with Generics

Since Mysidia supports subtyping with classes and traits, it gives rises to an interesting question of how subtyping works with Generics. **Variance** is the concept that refers to how subtyping between parameterized types(ie. Generic Types) relates to subtyping between their components. Depending on the variances of occasions, the subtyping relationship may be preserved, reversed or completely ignored. In general, there are three categories of Variances:

1. Covariance: The relationship of subtyping is preserved.
2. Contravariance: The relationship of subtyping is reversed.
3. Invariance: The relationship of subtyping is ignored.

Consider the following example, a generic class Basket<T> is a container class that can take any food, which include Fruit and its subclasses Apple, Banana, Grape, etc. The relationship is considered covariance if Basket<Fruit> is a supertype of Basket<Apple>, covariance if Basket<Fruit> is a subtype of Basket<Fruit> is a subtype of Basket<Apple>, and invariance if Basket<Fruit> and Basket<Apple> are unrelated types:

```
abstract class Food { }
abstract class Fruit extends Food{ }
class Apple extends Fruit { }
class Banana extends Fruit { }
class Grape extends Fruit { }
class Basket<T extends Food> {
    fields: {
        val List items
    }

    methods: {
        init() => items = LinkedList.new();
        T get() => items.removeLast();
        add(T item) => items.add(item);
    }
}
```

By default, Mysidia's generic type parameters are invariant, which means that it cannot be substituted by a subtype or supertype of the declared type. If we declare a Basket<Fruit> and deal with an object of Apple class, it will throw a Type Error, and vice versa:

```
val basket = Basket<Fruit>.new(Apple.new());
// Type Error: Cannot pass object of class Apple as argument for Basket<T> with type
parameter Fruit.
```

However this behavior can be overriden with Declaration-site Covariance and Contravariance for Generics. Covariant type parameters can be declared with the syntax *out T*, a generic class with covariant type parameters is considered a Producer class since it can return T and its subtypes. Contravariant type parameters can be declared with syntax *in T*, a generic class with contravariant type parameters is considered a Consumer class since it can accept T and its subtypes as arguments when responding to messages. Consider the specialized WhiteBasket<T>(can return T and its subclasses) and BlackBasket<T> classes(can accept T as methods arguments):

```
class WhiteBasket<out T extends Food> {
    fields: {
        val List items;
    }

    methods: {
        init(List foodList) => items = LinkedList.new(foodList);
        T get() => items.removeLast();
    }
}

class BlackBasket<in T extends Food> {
    fieleds: {
        val List items;
    }

    methods: {
        init() => items = LinkedList.new();
        add(T item) => items.add(items);
    }
}
```

```
val wBasket = WhiteBasket<Fruit>.new([Apple.new(), Apple.new(), Banana.new(),
Grape.new(), Banana.new()]);
Transcript.show(wBasket.get());
// It works, will print the string representation of a Banana object.
val bBasket = BlackBasket<Fruit>.new();
bBasket.add(Apple.new());
// It works, will append a Apple object to the Basket.
```

As we can see from the above code snippet, using covariance and contravariance can alter the subtyping relationship of a generic class with parameters. Covariance is read-only(Producer) and contravariance is write-only(Consumer), thus the two basket classes above can support either add or remove operation, but not both. This behavior is desirable sometimes, but can be hindrance in certain cases. Fortunately the limitation can be easily overcome by using multiple inheritance with both a Producer and Consumer traits, making a flexible generic class capable of both read/write functionality:

```
trait TProducer<out T> {
    abstract T get();
}

trait TConsumer<in T> {
    abstract add(T item);
}

class RealBasket<T extends Food> with TProducer<T>, TConsumer<T> {
    fields: {
        val List items
    }

    methods: {
        init() => items = LinkedList.new();
        T get() => items.removeLast();
        add(T item) => items.add(item);
    }
}

val basket = Basket<Fruit>.new();
basket.add(Apple.new());
basket.get();
```

Now both messages *.add(apple)* and *.get()* can be understood by the generic Basket class to return or add Apple objects to Basket<Fruit>, it is a combination of Producer and Consumer thanks to its implementation of two such *interfaces*(traits with only abstract methods). Mysidia's standard library provides trait objects that are covariant, contravariant, or both, they can be used by classes to provide more functionality and better type safety.

Reification and Runtime Generics

For different programming languages, the type information may be available(ie. C#) or unavailable(ie. Java) at runtime. The former is referred to as reification, in which the generic type parameters can be inspected and used to create objects of their types. The latter is called type erasure, in which the type information is completely lost and using them at runtime will lead to errors. Reification is usually considered the better and more complete approach for generics, though type erasures are sometimes chosen for the ease of migration.

Since Mysidia is a pure object oriented programming language with every operation being message passing to receiver objects, the type parameters themselves are not only reified at runtime, but also first class objects. The type parameters are effectively variables that store a type object, they can be used in similar way as instance fields and implicit variables such as *thisContext* when accessed in methods. It is perfectly valid to assign type parameters to variables, pass them as arguments in messages to a receiver object, and even to send messages directly to them:

```
class BasketTest<T> {
    methods: {
        test(){
            val type = T;
            Transcript.show(type);
            Transcript.show(type.class());
            val item = T.new();
            Transcript.show(item);
            Transcript.show(T);
            Transcript.show(item.instanceof(T));
        }
    }
}

val basketTest = BasketTest<DateTime>.new();
basketTest.test();
```

The type parameter T is assigned to local variable *type*, and depending on the substituted type at runtime it will represent the corresponding type object(a class, metaclass or trait). Sending message *.new()* to T will create an instance of T provided that T is a non-abstract class(or metaclass), it will not work for traits. Sending message *.class()* to T will return the class of type T, which is a metaclass if T is a class itself, or the class *Trait* if T is a trait object. Sending T as an argument in a message works no differently from any other variables. The above code will print the following lines on the transcript, which is exactly as expected:

```
Class DateTime
Metaclass DateTime class
01-01-2020
Class DateTime
true
```

Generic Collection Classes

The Mysidia Standard Library comes with many generic classes, good examples are the Collection classes. They can be found at the namespace Mysidia.Standard.Collection.Generic, and unlike the dynamically typed collection classes introduced at chapter 9 and 10, they make good use of type hinting for fields and methods. Most of these collection classes have one type parameter E that represents type of the elements, while map classes have two type parameters K and V which are substituted for the types of keys and values. The most commonly used Generic Collections are:

```
Collection<E>
List<E>, Array<E>, ArrayList<E>, LinkedList<E>, SortedList<E>
Set<E>, HashSet<E>, Bag<E>
Map<K, V>, Dictionary<K, V>, HashMap<K, V>, LinkedMap<K, V>, SortedMap<K, V>
```

Proper use of these generic collection classes can enforce type safety for operations with their elements. For instance, we cannot add an Int object to a generic Array<String> object, it will trigger a compile time error:

```
val al = ArrayList<String>.new(5);
al[0] = "Hello World";
al[1] = 3;
// Type Error: Cannot add Int 3 to ArrayList, it expects String object.
val hm = HashMap<String, DateTime>.new();
hm.put(key: "today", DateTime.now());
hm["tomorrow"]= Date.tomorrow();
// Type Error: Cannot add Date object as value to Dictionary, it expects DateTime
object.
```

The collection classes use trait TEnumerable<T> to provide iterator methods such as *each: closure*. TEnumerable<T> is a covariant trait, whose return values can be any subtype of T. We can safely acquire an Int or Float object from ArrayList<Number>, since it satisfies the covariance(since Int or Float is subtype of Number). However, collection classes are not contravariant with method arguments, and thus cannot accept supertype of T as elements. This means that we cannot add a supertype to a collection of subtypes, ie. adding a Date object to HashSet<DateTime>(DateTime is subclass of Date) is illegal and will result in Type Error.

Generics and Closure Type Hinting

Type hinting can be tricky with methods that accept closure as argument or return value, and this issue can be safely resolved with Generics. Similar to generic collections, Mysidia also support generics for closures. Closures type hinting can be achieved with generic parameters for the arguments and return types. It may accept up to 6 generic parameters, namely 5 argument parameter types(T1, T2, T3, T4 and T5) and 1 return type R. The return type parameter is always specified at the very last, after all arguments, as shown in the example below:

```
Closure<R> //Closure with no argument, and return type R.
Closure<T, R> //Closure with 1 argument type T, and return type R.
Closure<T1, T2, R> //Closure with 2 argument types T1, T2, and return type R.
Closure<T1, T2, T3, T4, T5, R> //Closure with 5 argument types T1...T5, and return
type R.
```

With this in mind, we can easily write a typed hinted method with closures as argument(s) and return type. Consider the below class which has methods that make use of type hinting with closures:

```
class IntParser {
    methods: {
        String parseInput(Int input) => this.defaultParse()(input);
        String parseInput(Int input, Closure<Int, String> closure) => closure(input);
        Closure<Int, String> defaultParse(Int input) => input => "Integer: ${input}";
    }
}
```

The IntParser class may accept parammetric Closure<Int, String> as argument as well as return value, which provides type safe approach to handle closures as well as compiler optimization for future works. Note Mysidia's closure type hinting wtih generics only supports up to 5 arguments, it is not possible to type hint closures with more arguments. We believe that closures should be concise and light weight, and most use cases of closures will not need more than 2 arguments. Developers should consider creating a class instead of using closure objects if it needs to accept more than 5 arguments.

Summary

Polymorphism is vital to the concept of Object Oriented Programming, and Mysidia supports 3 of the 4 common types of Polymorphism. Among them the Parametric Polymorphism is unique that it allows statically typed programs to be far more flexible, it is implemented as Generics in modern programming languages. In Mysidia, Generics can be achieved by declare type parameters such as T enclosed in angle bracket next to the name of a class, trait or method. It is also possible to specify constraints and variances for type parameters. Generics information is reified at runtime and the type parameters are special variables that can be any type objects(which uses trait TType), which allows more powerful code to be written making use of them. Advanced usage of type parameters requires the knowledge of Reflection, and this feature will be explored in later chapters.

Chapter 15: Composition and Delegation

Introduction

The very last feature of object oriented programming yet to be introduced formally is **Composition**, which is a way to combine small and simple objects into more complex ones. Composition is typically acccomplished by defining an owner class with its instance variables being other objects. Usage of composition usually implies an aggregation relationship between the owner object and its dependent objects. One way to implement composition is to use delegation, and Mysidia has language support for the commonly used delegation pattern which will be discussed comprehensively in this chapter.

Composition versus Inheritance

In object oriented programming, both composition and inheritance are commonly used to describe relationship between objects/classes. Composition emphasizes the *'has-a' relationship* between an owner object and its dependent objects, ie. a car has an engine, a tree has branches and leaves. On the other hand, inheritance tells about the *'is-a' relationship* between a superclass and its subclasses, ie. an integer is a number, a car or truck is a vehicle. These are the textbook examples that we have been taught at universities and online-courses, and should be enough familiar to developers who have been exposed to at least elementary level of object oriented programming.

It may seem straightforward, but unfortunately real life applications tend to be more complex than that. Almost nobody works with vehicles and animals in practical software applications, while the 'has-a' and 'is-a' relationships between classes usually aint that obvious. Sometimes we run into 'gray-areas' that both 'has-a' and 'is-a' may sound logical. At this point, it will require developers to make informed decisions on whether to use composition or inheritance to describe object relations.

One good such example is a CMS system as shown in the below code snippet, in which there can be different types of users such as Member, Publisher, Admin, Banned etc. If we think in terms of people, then it seems natural to assume the *is-a* relationship as publishers, admins and banned users are all members, hence inheritance appears to be the proper choice. However, if we are talking about roles, then the *has-a* relationship makes more sense, as a publisher has user role as well as additional roles unique to publisher itself, similarly for admins and banned users, this leads to choice of composition.

```
class Member {
    fields: {
        val String id, String username;
        var Int totalLikes;
    }

    methods: {
        init(String id, String username, Int totalLikes){
            this.id = id;
            this.username = username;
            this.totalLikes = totalLikes;
        }
        String id() => id;
        String username() => username;
        Int like() => totalLikes += 1;
        Int dislike() => totalLikes -= 1;
        String comment() => "User ${username} is commenting on an article.";
    }
}

// Using inheritance
class Publisher extends Members {
    methods: {
        String publish() => "Publisher ${username} is publishing an article";
    }
}

// Using composition
class Publisher {
    fields: {
        val Member member
    }

    methods: {
        String publish() => "Publisher ${member.username()} is publishing an
article.";
    }
}
```

The GOF book suggests this following principle: **Favor composition over inheritance**. It has been proven in the industry that composition has many advantages over inheritance, as the latter introduces tight coupling between the class and its superclass. Most traditional OO languages do not allow substitution of superclass at runtime, while the lack of support for multiple inheritance makes it hard to describe multiple relations. This does not mean that inheritance has no place in object oriented programming, its simply that developers should use it with care. Use inheritance only when the *is-a* relationship is obvious, and when the tight coupling between classes is actually desirable.

Nonetheless, composition has one major drawback in traditional OO languages, especially when the owner class needs to implement behaviors from the dependent class, frequently found for decorator and adapter classes. If we need to make an instance of Publisher class understand messages available to a User object, we will have to re-implement every method from the User class to the Publisher class:

```
// Using composition
class Publisher {
    fields: {
        val Member member
    }

    methods: {
        init(Member member){
            this.member = member;
        }
        String id() => member.id();
        String username() => member.username();
        Int like() => member.like();
        Int dislike() => member.dislike();
        String comment() => member.comment();
        String publish() => "Publisher ${this.username()} is publishing an article.";
    }
}
```

Now we can see clearly that the traditional way of composition may introduce a nontrivial amount of boilerplate code. The need for *method forwarding* can make composition less favorable than inheritance, especially when the number of methods to be forwarded is significant compared to the methods to be added/overriden. Fortunately, Mysidia has language support for delegation to mitigate this problem with composition, which will be discussed in detail at the following section.

Delegation Fields

The delegation pattern is commonly used for implementing composition relationship between the owner and the dependent classes. This is usually done by passing the dependent objects as arguments to the object initializer(or class constructor for traditional OO languages), and assign them as instance fields. Developers familiar with dependency injection should have seen plenty of this pattern in action, though again the need for explicit method forwarding can be a serious drawback.

In order to make delegation and dependency injection more enjoyable and convenient to use, Mysidia provides language support for delegation to remove the boilerplate code for method forwarding. This can be achieved by appending *delegate* before the variable declaration as shown in the code snippet below:

```
class Publisher {
    fields: {
        delegate val Member member
    }

    methods: {
        init(Member member){
            this.member = member;
        }

        String comment() => "Publisher ${this.username()} is commenting on an
article.";
        String publish() => "Publisher ${this.username()} is publishing an article.";
    }
}
```

The above example uses the syntax sugar **delegate val field-name** to create a delegation field(also called delegate) member. This allows automatic method forwarding from field *member* to its owning class *Publisher*, as Mysidia compiler automatically implements every method from Member to Publisher if it is missing from Publisher's method dictionary. This includes Member and its superclasses, but excluding the root Object class. Detailed explanation of how delegation works behind the scene will be explored in the later sections about *delegation slots*.

Note delegates must have type information specified(Member in this case), or a compile time error will occur. On the other hand, only instance methods from the delegate will be implemented in the owner class, but not the instance fields. For this reason, it is still necessary to send getter messages if the owner class needs to directly access the instance fields on the dependent object for any reasons.

To see delegation fields in action, we can run the following program:

```
val publisher = Publisher.new(Member.new(1, username: "Hall of Famer", totalLikes: 5));
Transcript.show(publisher.username()) //prints: Hall of Famer
Transcript.show(publisher.comment()) //prints: Publisher Hall of Famer is commenting on an article.
Transcript.show(publisher.publish()); //prints: Publisher Hall of Famer is publishing an article.
```

The result shows that the Publisher class has successfully 'inherited' the method *username()* from the delegate member, while the method *comment()* is implemented differently and behaves different from the original implementation as if it has been overriden. The other method *publish()* is unique to the Publisher class and also behaves as expected.

We can see that delegation is actually quite similar to inheritance, except that the owner class and the dependent class are loosely coupled. It is possible to swap the instance of Member by an instance of its subclass, and the publisher object will still work properly, even exhibiting polymorphic behaviors. With delegation, developers enjoy greater flexibility as they can substitute the dependent object by a different object at runtime. Developers are advised to consider delegation when tight coupling between a class and its superclass does not make sense, while composable behavior is desirable.

Delegation with Traits

The code from last section uses delegation field with a class, which is suitable for simple owner classes that behave as decorator or adapter for the dependent class. Although it sharply reduces the amount of boilerplate code needed for composition, it is not without flaws. For instance, we may have a similar class *Admin* which also has a delegate *member*:

```
class Admin {
    fields: {
        delegate val Member member
    }

    methods: {
        init(Member member){
            this.member = member;
        }

        String comment() => "Admin ${this.username()} is commenting on an article.";
        String approve() => "Admin ${this.username()} has approved an article.";
        String edit() => "Admin ${this.username()} is editing an article.";
    }
}
```

It looks alright, but it appears that its still not flexible enough. What if the member can have multiple roles? An publisher can also be an admin, and vice-versa, they need not be be mutually exclusive. When an admin approves/edits an article, it does not matter if it is a publisher or an ordinary member. This multi-role system cannot be achieved with the current design, so we will need an alternative solution. To solve this problem, we can make use of traits as types for delegation fields. The classes Member, Publisher and Admin can be written as below:

```
trait TMember {
    methods: {
        String id() => id;
        String username() => username;
        Int like() => totalLikes += 1;
        Int dislike() => totalLikes -= 1;
        abstract String comment();
    }
}

class Member with TMember {
    fields: {
        val String id, String username;
        var Int totalLikes;
    }

    methods: {
        init(String id, String username, Int totalLikes){
            this.id = id;
            this.username = username;
            this.totalLikes = totalLikes;
        }

        String comment() => "User ${username} is commenting on an article.";
    }
}

abstract class DelegateMember with TMember {
    fields: {
        delegate val TMember member
    }

    methods: {
        init(TMember member){
            this.member = member;
        }
    }
}

class Publisher extends DelegateMember {
    methods: {
        String comment() => "Publisher ${this.username()} is commenting on an
article.";
        String publish() => "Publisher ${this.username()} is publishing an article.";
    }
}

class Admin extends DelegateMember {
    methods: {
        String comment() => "Admin ${this.username()} is commenting on an article.";
        String approve() => "Admin ${this.username()} has approved an article.";
        String edit() => "Admin ${this.username()} is editing an article.";
    }
}
```

A trait TMember is defined as the supertype that every class implements, and the delegate member is typed as trait TMember instead of class Member. Also to avoid code duplication, an abstract class DelegateMember is created which contains common field and methods for both Publisher and Admin. Now when sending message *.new(member)* to Publisher or Admin class, the argument member simply needs to be an object with trait TMember. With this change, it is now possible to even pass an instance of Publisher to Admin, and vicer versa. To see how it works in action, we can run the following code snippet:

```
val publisher = Publisher.new(Member.new(1, username: "Hall of Famer", totalLikes:
5));
Transcript.show(publisher.username()) //prints: Hall of Famer
Transcript.show(publisher.publish()); //prints: Publisher Hall of Famer is publishing
an article.
val admin = Admin.new(publisher);
Transcript.show(admin.approve()); //prints Admin Hall of Famer has approved an
article.
```

It is evident that traits can make the system more flexible, by achieving composable behavuors with objects/classes. There is another benefit for using traits as delegate type, as Mysidia allows only one delegation field to be typed as class, but there is no such restriction for traits. This design decision for delegation parallels with the single superclass/multiple traits policy for inheritance. Consider another example below:

```
trait TReadStream<T> {
    methods: {
        T next();
        T peek();
    }
}

trait TWriteStream<T> {
    methods: {
        next(Int num, Collection collection, Int startingPosition);
        nextPut(T object);
    }
}
```

```
class ReadWriteStream<T> {
    fields: {
        delegate val TReadStream reader, TWriteStream writer
    }

    methods: {
        init(TReadStream<T> reader, TWriteStream<T> writer){
            this.reader = reader;
            this.writer = writer;
        }
    }
}
```

It is valid to send messages *.next()* and *.nextPut(object)* to an instance of ReadWriteStream, and it will be able to respond to both messages though they are implemented from two different traits(TReadStream and TWriteStream). We can see clearly that traits allow for not only more elegant class design, but also reduction of boilerplate if used as delegate types. Note this is just a simplified version of Mysidia's IO/Stream, which will be discussed thoroughly in the next chapter.

Delegation slots

Delegates are a powerful language feature for creating compositional relationship among classes/objects, and developers may be intrigued at how Mysidia implement delegation pattern in the VM. Since Mysidia does not have the concept of keyword/operater, while everything happens by sending messages, the declaration of delegation fields is no different. The notation *delegate variable-literal* is actually an implicitly sent tertiary message to thisContext object, which happens to be the MainContext object in this case(assuming the class is defined directly in a namespace rather than inside a method/closure). The below two lines are equivalent:

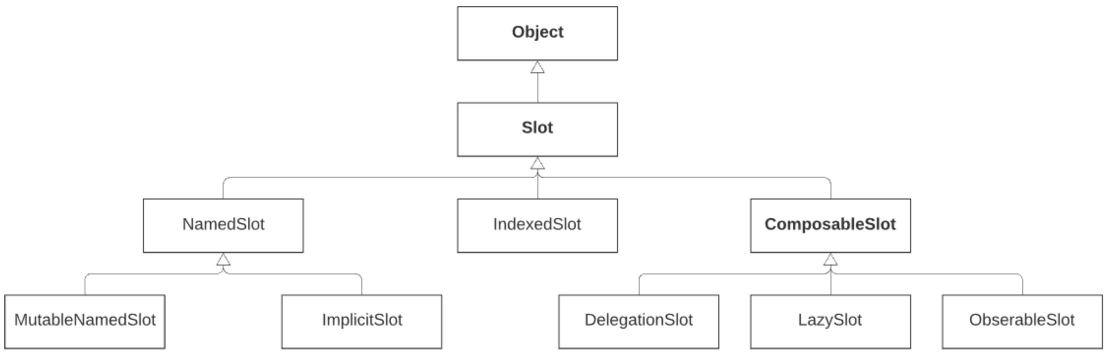
```
//Implicit send and variable literal declaration
delegate val TMember member

//Explicit send and explicit variable declaration
thisContext.delegate(val TMember member)
```

When the MainContext receives message *.delegate(variable)*, it will execute the corresponding method and return an instanceof **DelegationSlot**. As already mentioned in Mysidia's unified variable system, all variables in Mysidia are reified as instances of *Slot* class. It can be understood that a delegation field is effectively an instance of the class DelegationSlot. Upon instantiation, the initializer of DelegationSlot automatically handles the implementation of methods from the delegate to its owner class. This reduces a considerable amount of boilerplate code developers have to write themselves in traditional OO languages.

With this in mind, we can rewrite the delegation field declaration using DelegationSlot as shown below, which explicitly declares the instance field member to be an instance of class DelegationSlot:

```
val TMember member -> DelegationSlot
```

At first glance, it does not look better than the original version at all, being way verbose and less intuitive. However, the explicit slot approach is more powerful as we can swap the class DelegationSlot by a different Slot class to achieve various behaviors for the instance field. In fact, DelegationSlot is a subclass of abstract class **ComposableSlot**, which may wrap other slots and alter the way an instance variable is read or written. For the above example, the instance field member is created as a DelegationSlot composed of a NamedSlot internally. For reference, the Slot hierarchy is shown below:



It is also achievable for developers to create their own subclasses of ComposableSlot similar to DelegationSlot, while Mysidia's standard library also comes with a few built-in classes that inherit from ComposableSlot. The most common of such classes are LazySlot and ObservableSlot, and will be explored in the next section in detail.

Lazy and Observable slots

Similar to DelegationSlot, it is possible to use other subclasses of **ComposableSlot** to add/alter behaviors for reading/writing to instance fields. Quite often developers will need to have lazy loaded variables whose values are only initialized or computed when they are first accessed. Many modern programming languages such as Swift and Kotlin have dedicated support for lazy fields/variables, while this same feat can be easily achieved with **LazySlot** in Mysidia. The below code snippet demonstrates how to create a LazySlot, it is quite similar to create a DelegationSlot:

```
class Cart {
    fields: {
        val List<CartItem> items;
        val Float totalPrice -> LazySlot;
    }

    methods: {
        init(List<CartItem> items){
            this.items = items;
            totalPrice onInitialAccess: { items.inject((sum, item) => sum + 
item.price()) }
        }
    }

}
```

The Cart class contains a list of CartItem objects, whose class definition is not shown above, though its clear that it can respond to message *price*. The instance field *totalPrice* is declared to be a LazySlot, and the value will not be computed until it is first accessed. At this point, an instance of class LazySlot is masquerading as totalPrice and it is able to understand messages for LazySlot.

To make the lazy instance field useful, it is necessary to specify its lazy loaded value somehow. To accomplish this, we send message *onInitialAccess: closure* to *totalPrice*. After sending this message to *totalPrice*, the next access to this instance field will evaluate the closure and assign its return value to totalPrice. Beyond this point, it is fully intialized and will behave no different from an ordinary instance variable.

Other than LazySlot, Mysidia also provides **ObservableSlot** class from its standard library. It is another subclass of abstract class ComposableSlot, and it is often used to enable client code to be notified over changes of an instance field. Developers familiar with observer pattern should have a good appreciation of how much it enables them to accomplish with observers/observables, although implementing the observer pattern in traditional OO languages usually means a ton of boilerplate code. Fortunately, ObservableSlot in Mysidia makes observer pattern far more convenient to use, as the below example demonstrates:

```
class Person {
    fields: {
        val String name;
        var Int age -> ObservableSlot;
    }

    methods: {
        init(String name, Int age){
            this.name = name;
            this.age = age;
            this.age onChange: (newAge){
                Transcript.show("The person ${name}'s age has changed to ${newAge}. ")
            }
        }
    }
}
```

The class Person has an ordinary instance field *name*, as well as another instance field *age* declared as ObservableSlot. At this point it does not behave any different from an ordinary MutableNamedSlot, but this instance field is now able to understand message *onChange: listener* which will execute the passed closure as 'event listener'(which accepts one argument for the newly assigned value). It allows us to watch over any changes made to the instance field *age*, effectively achieving the observer pattern with significant less boilerplate.

Instances of ObservableSlot also understand two more common messages. Other than using *onchange: listener* to attach a change listener, it is possible to send a different message *addChangeListener: listener* which allows multiple listeners for the observable instance field. If developers need to make changes to the observable field without triggering the event listener, they can send message *rawValue: newValue* to the instance field which will silently mutate its value. Observable slots are powerful feature that can be used to implement logging as well as domain events for Domain Driven Design.

Summary

Following the introduction of Composition and Delegation, we have completely explored the Mysidia's implementation for the four fundamental features of OOP. In order to make composition easier and more enjoyable to use, Mysidia provides language support for delegation which removes the boilerplate code needed for method forwarding. Developers are still expected to make intelligent decisions on whether to choose composition or inheritance for a subsystem of their application, depending on the nature of the problem they are solving. The next two chapters will take a brief detour from language features/design, as we will introduce the standard library for IO and Error handling which are also vital to write practical applications in Mysidia programming language.

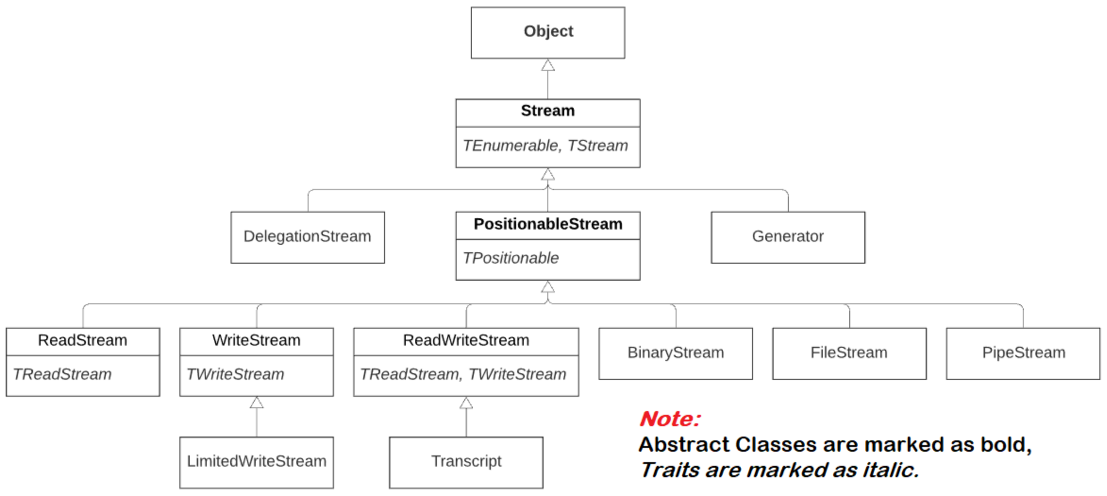
Chapter 16: IO and Streams

Introduction

At this point we have explored the Lang and Collection packages from the Standard library, which provide basic classes for developers to code in Mysidia programming language. Quite often we need to perform input/output related tasks in their application, and Mysidia has its own IO package in the standard library to make it convenient for the developers. The IO package contains a group of classes called streams, which are stateful objects that offer a means to sequentially access elements of a collection or external resource. This chapter will introduce the IO package as well as some of the most commonly used classes.

The IO package

The classes in Mysidia's IO package are usually found in the namespace **Mysidia.Standard.IO** (with rare exceptions such as Transcript which is in namespace Mysidia.Standard.Lang). The abstract class **Stream** (Mysidia.Standard.IO.Stream) at the top of the class hierarchy, as every IO classes inherit functionality from this superclass. This class Stream provides interruptable sequential access to objects, it also uses the trait TEnumerable to allow simple iteration over the contents of its instances.



There are a few subclasses that inherit directly from the base class **Stream**, with the class **PositionableStream** as the most commonly used by developers. PositinableStream is yet another abstract class that permits explicit positioning of its internal content. This class stores an instance field *position*, which advances automatically when reading/writing. There are other subclasses of Stream such as **DelegationStream** and **Generator**. The former allows decorating of other streams using the delegation pattern, while the latter provides a way to use closures to define a stream of many return values(this can come in handy for concurrent programming, which will be explored in later chapters).

The abstract class PositionableStream has a number of specialized subclasses that stream over different types of collections/contents. The class ReadStream and WriteStream deal with in memory objects/collections, and other classes such as

FileStream, **BinaryStream**, **PipeStream**, etc are used to handle streaming over external resources. The next few sections will introduce each of the different stream classes, as developers will get a better understanding of how to perform IO tasks in Mysidia programming language.

Read and Write Streams

It is a common operation to read/write in-memory objects. The class **ReadStream** can be used to read contents from a collection sequentially. There are two ways to create an instance of ReadStream, by sending message *new: collection* to class ReadStream, or message *readStream* to a collection object. The two approaches are equivalent and will produce the same read stream object. It is also possible to send message *new: collection from: fromIndex to: toIndex* to get a read stream of a subcollection.

```
// Creating read stream - approach 1:
val readStream = ReadStream.new(collection);

// Creating read stream - approach 2:
val readStream = collection.readStream();

// Creating read stream from a subcollection:
val readStream = ReadStream.new(collection, from: fromIndex, to: toIndex);
```

Instances of class ReadStream understand a few messages to allow walking over the stream contents. The messages *next* and *next: num* tell the stream to read the next element or next few elements, which will advance the position of the stream. The message *peek* can be sent to find the next element without moving the position of the stream forward, while the message *skip* or *skip: num* will simply advance the stream's position by skipping the elements. To check if an element or elements exist in the stream, one can send message *peekFor: element* which returns boolean true if the element(s) can be found. The message *isAtEnd* will check if the stream has traversed to the end of its content, while message *ifAtEnd: closure* will execute a block/closure incase the stream is at the end. The below sample program demonstrates how to use ReadStream in Mysidia:

```
val stream = ReadStream.new([1, 2.2, false, 'a']);
val a = stream.next();
val b = stream.peek();
val c = stream.next(2);
val d = stream.peek();
Transcript.show(a);.show(b);.show(c);.show(d);

val stream2 = ReadStream.new("Mary had a little lamb".split(" "));
stream2.skip();
val e = stream2.peek();
val f = stream2.peekFor("lamb");
val g = stream2.isAtEnd();
stream2.skip(4);
Transcript.show(e);.show(f);.show(g);
stream2 ifAtEnd: { Transcript.show("Stream is at the end.") }
```

The following output will be printed on the transcript:

```
1
2.2
false
a
had
true
false
Stream is at the end
```

On the other hand, the class **WriteStream** can be used to write contents to an empty collection. Similar to ReadStream, an instance of WriteStream may be created by sending message *new: collection* to WriteStream class, as well as message *writeStream* to a collection object. It is also possible to send message *new* without an argument, in which an empty ArrayList will be used as default collection object:

```
// Creating write stream - approach 1:
val writeStream = WriteStream.new(collection);

// Creating write stream - approach 2:
val writeStream = collection.writeStream();

// Creating write stream without a collection argument(collection defaults to an
ArrayList):
val writeStream = WriteStream.new();
```

Instances of class WriteStream understand a few messages to modify stream contents. The messages *put: element* and *putAll: collection* tell the stream to write an element or an entire collection to the stream at the current position. This will not advance the stream position, as a successive message will overwrite the element/collection. One can also send messages *putNext: element* and *PutAllNext: collection* to WriteStream to not only write an element/collection, but also move the stream position to the next. The code below demonstrates how to work with WriteStream in Mysidia:

```
val stream = WriteStream.new(StringBuilder.new());
stream.put('a');
stream.putAll("Mary had ");
stream.next();
stream.putNext('a');
stream.putNextAll(" little lamb");
Transcript.show(stream.contents());
```

This will print string "Mary had a little lamb" on the Transcript. It is possible to convert between read and write stream by sending message *readStream* to an instance of WriteStream, or message *writeStream* to an instance of ReadStream. Mysidia also provides a class **ReadWriteStream** which combines methods from ReadStream and WriteStream. The VM implementation uses trait TReadStream and TWriteStream for the most common operations. ReadStream implements TReadStream, WriteStream implements TWriteStream, while ReadWriteStream simply implements both traits.

It is worth noting that ReadStream, WriteStream and ReadWriteStream are subclasses of abstract class PositionableStream, thus they all share some common functionalities. As shown from the example above, the message *contents* will retrieve the collection object inside the stream. These stream classes also understand message *position* which returns the current position cursor of the stream, as well as message *setPosition: index* which sets the position index to the provided integer value. The message *reset* allows the streams to reset the position cursor to 0(the beginning), while message *moveToEnd* will move the position to the end of the stream.

Transcript

Before moving on to the external streams of Mysidia IO package, it is worth discussing a widely used stream called **Transcript**. As we have seen from the previous chapters, the class Transcript is frequently used to write contents to the screen like Smalltalk's Transcript, but it actually behaves very differently. Smalltalk's Transcript is a global variable for the stdout stream, while Mysidia's Transcript is a class that serves as both stdin and stdout similar to C#'s Console class. As Mysidia lacks support for global variables, the methods are implemented on the metaclass *Transcript class*.

File and Binary Streams

Object Serialization/Deserialization

Summary