Trent University

Ray Programming Language Reference

Bobby Horth
COIS-4100H
December 12th 2020

Table Of Contents

Abstract

Ray is a general purpose, high-level, object oriented, statically typed, compiled programming language. The language is designed to bridge the gap between simplicity and efficiency by bolstering impressive speeds with minimal barriers for entry so newer and veteran programmers alike can quickly pick up the syntax and begin creating high quality and high efficiency code. The language has no pointers, no manual memory management, minimal constructs, and a clear concise syntax. It brings the simplicity of a language like Python while providing the efficiency of a language like Java.

Ray's performance allows for high efficiency programs unparalleled by languages with similar syntax or learning curves like Python or Ruby making it a strong choice for core systems. It's simple syntax allows for both fast development, making it useful for command line programs, and quick iteration, making it useful for backend web, desktop applications, or games.

Program Structure

Lines and Joining
A program is a .ray file made up of logical lines which end at a NEWLINE token. Logical lines are made of 1 or more physical lines separated by a join token (\) where physical lines are a sequence of characters terminated by a end-of-line sequence.

Formal Syntax:

<physical line> → <character> <end of line> | <physical line> <character> <end of line>
<logical line> → <physical line> | <logical line> \ <physical line>
<program> → <logical line> | <program> <logical line>

Example Program:

```
Integer myInt = 5
Boolean bool = myInt is Integer and \
     myInt < 10
```

Comments
Programs can also contain comments which are characters not parsed by the compiler. Comments cannot use the join token and cannot be embedded within each other. The # token signifies the start of a comment which spans only the remainder of the physical line.

Example:

```
Integer myInt = 5 # declare integer
```

Indentation
Leading whitespace (spaces or tabs) at the start of a given logical line is parsed by the compiler to determine the indentation level of a line. The indentation level of a line

determines which grouping of statements a line belongs to. If inconsistent indentation is detected by the compiler it will throw an error until it is resolved.

Correct Indentation Example:

```
for i in {1, 2, 3}:
      Integer x = i * 5
```

Incorrect Indentation Example:

```
      Integer myInt = 2
           if myInt equals 5:
return true
```

Keywords

Keywords are the reserved words in Ray and thus they cannot be used to represent variables, functions or classes; they are only used as the building blocks of the language. Keywords are used to represent constant values, create classes and objects, manipulate control structures and more.

All keywords are fully lower case.

A full list of key words can be found below but their implementation and other details can be found in their respective sections throughout the language reference.

true	false	null	continue
if	else	else if	break
for	while	not	and
or	return	class	import
of	in	try	catch
open	closed	children	group
is	static	error	inherits
blueprint	generic	equals	

Identifiers

Identifiers are names attributed to classes, functions, or variables. They are used to distinguish between entities, declare variables, instantiate objects, call methods, and manipulate attributes.

Identifiers are a combination of letters (upper or lower case), digits, and underscores but cannot begin with a number. No special symbols or other unicode characters are valid within identifiers.

The case of letters within identifiers is not enforced but camel case is recommended for class names while pascal case is recommended for variables and classes.

Valid examples:

- SomeClass
- someVariable
- someFunction
- Another_Class
- Another_Variable
- aWaCkYfU_nCtio_nName

Formal Syntax:

<identifier> → <_ | a | b | c | … | z | 0 | 1 | … | 9> | <identifier> <_ | a | b | c | … | z | 0 | 1 | … | 9>

Variables

A variable is an abstraction of a memory cell. A variable has an identifier used within statements to invoke methods, pass the variable as a parameter, or perform operations. A variable is also associated with an address which is the location in memory the underlying data is stored at, this address may change during execution. Each variable is also associated with a type which determines which values the variable can store. A variable is declared using the assignment operator and evaluated upon reaching the statement during execution or at compile time for static variables.

There are three types of variables in Ray: static class variables, class variables, and local variables. A static class variable must be declared outside of a function using the static keyword and is common to all instances of a class. A class variable must be declared outside of function without the use of the static keyword and is unique to each instance

of the class. A local variable must be declared within a function and is unique to that function.

All objects in Ray are heap-dynamic and are created either using the constructor for a given class, referenced by its class name followed by parenthesis or through literals which create objects of default data types.

Scope

The scope of a variable represents from where in the program a variable is visible. A local variable is declared within a function and its scope is simply the block of statements contained within the function. A class variable is a variable declared within a class but outside of a function and its scope is the entirety of the statements within the class including within functions.

The scope of a variable has implications on the identifier which is associated with the variable. One cannot identify a variable with a name that already exists within its scope or a larger scope. Static variables take precedence over class variables which take precedence over local variables meaning for example, the compiler will throw an error on local variables declared with the same identifier as class variables.

Once a variable is out of scope, its lifetime is over and it is collected by the garbage collector freeing its location in memory.

Literals

Literals are objects created directly from their values and are backed by the default simple, numerical, and textual data types. The compiler evaluates literals and creates objects from their value. All default data types can be instantiated using literals.

Formal Syntax:

<boolean literal> → true|false
<digits> → <digit>|<digits><digit>
<byte literal> → <digits>
<integer literal> → <digits>
<long literal> → <digits>
<float literal> → <digits>.<digits>
<double literal> → <digits>.<digits>
<characters> → <character>|<characters><character>

<string literal> → “<characters>“
<char literal> → ‘<character>’
<objects> → <object> | <objects>, <object>
<list literal> → {<objects>}

Examples:

```
Boolean a = true
Byte b = 5
Integer c = 20000
Long d = 5129527074012407
Float e = 2.3
Double f = 2.333333333333
Char g = 'c'
String h = "hello"
List of String i = {"hello", "how”, “are”, “you”}
```

Expressions

Expressions create values when they are evaluated and are used for declarations, comparisons, operations, and parameters. Expressions are built from atoms which are the most basic elements in the language such as identifiers or literals.

Formal Syntax:

<expression> → <identifier> | <literal> | <function call> | <expression> <logical operator> <expression> | <expression> <arithmetic operator> <expression>

Data Types

Data Types are the classification of objects within Ray. All (non-null) data types inherit from the root type Object which can be stored in variables, passed as a parameter to functions, and operated on with internal or external methods. The null keyword represents a special data type which is unnamed and represents a null expression. Objects can not have the type null since it has no name but can be instantiated as null as it is a valid expression. The null type can be assigned or cast to any other data type.

Ray is statically typed meaning at compile type every variable and expression must have a known, valid type. The language is also strongly typed which means the data types themselves limit the values which a variable of the type can store, limit the operations

that can be used on objects of the type, and limit the methods an object of the type can invocate.

Ray comes equipped with several default data types which can be found in the *ray.datatypes* group. These data types come fully equipped with constructors and internal methods and act as the building blocks for creating programs.

Default Data Types

Simple

- These types store only a small number of fixed values
- A *Boolean* represents a valid or invalid expression and must be assigned via the *true* or *false* keywords

Imprecise Numerical

- These types only store whole numbers within a given range
- A *Byte* represents a small sized number ranging from -128 to 127
- An *Integer* represents a medium sized number ranging from -2147483648 to 2147483647
- A *Long* is a large sized number ranging from -9223372036854775808 to 9223372036854775807

Precise Numerical

- These types store numbers with a given decimal precision making them much more precise than the previous types but are larger in size
- These types utilize the IEEE 754 standard for Floating-Point Arithmetic
- A *Float* represents a 32-bit floating-point number
- A *Double* represents a 64-bit floating-point number

Textual

- These types store text based data such as letters, words, sentences, or larger text passages
- A *Char* represents a single unicode character
- A String represents a sequence of characters which can be any length

Complex

- A complex type stores some combination of earlier types
- A *List* represents an ordered list of objects such that the whole list is of a single type (ex. list of bytes)

- Lists are Heap-dynamic meaning their subscript range and storage is allocated dynamically and the size can change during execution
- A List is a generic type meaning it can store objects of any type, not only default types

Coercion

Implicit Coercion

Ray supports the concept of implicit conversion which is automatic conversion of a default numerical data type to another default numerical data type where appropriate. This operation is always a widening conversion which means only smaller types can be converted to larger types. This means bytes can be implicitly coerced to any default numerical data type since they are the smallest of the types. This can be done during declaration of a variable by coercing the right hand side of the assignment or invocation of a function by coercing a passed parameter.

Formal Syntax:

<declaration> → <wide type> <name> = <narrow type>
<invocation> → <function>(<narrow type>)

Code Examples:

```
Integer myInt = 5
Float myFloat = myInt


none myFunction(Integer int):
    // some code

Byte myByte = 4
myFunction(myByte)
```

Explicit Coercion: Casting

Ray also supports the concept of casting which is converting an object to a type explicitly declared by the cast operation and can be done on all data types. This is possible because variables in Ray are polymorphic.

There are three types of casts in Ray, an identity conversion, a widening conversion, and a narrowing conversion. All objects support the identity conversion which is casting an object to its own type, this can be done semantically for clarity and has no implications on the object. A widening conversion allows objects to cast to a type inheriting their own type or on the default numerical types in order to store a smaller number in a larger storage type. A narrowing conversion allows objects to a type which they inherit, their parent, or on the default numerical types in order to store as much of a larger number as possible into a small storage type.

Formal Syntax:

<cast> → (<type>) <object>

Code Example:

```
ChildClass obj = 5
ParentClass parent = (ParentClass) obj
```

Operators

Operators are constructs which behave similarly to function and perform an operation on one or more operands and either return a result or perform an assignment. There are three types of operators in Ray: assignment operators, logical operators, and arithmetic operators and they perform very different tasks. Assignment operators assign values to variables, logical operators perform comparisons or inversions and always return a boolean, and arithmetic operators perform mathematical functions on their operands and return a numerical result.

Assignment Operators

- The = operator assigns the value of the expression on the right side of the operator to the variable on the left side
    - Syntax: <assign> → <type> <name> = <expression>

- The *+=*, *-=*, **=*, */=*, *^=*, and *%=* operators perform an arithmetic operation and an assignment in one statement on an existing variable. For example, the += operator would assign the variable on the left side of the operator to its own value added to the expression on the right side of the operator. The syntax for all of these operations is identical.
    - Syntax:

<arithmetic assignment> → <variable> <*+=* | *-=* | **=* | */=* | *^=* | *%=*> <expression>

Logical Operators

- The *not* operator inverts a boolean expression returning *false* for a true expression and *true* for a false expression
    - Syntax: <not> → not <expression>

- The *is* operator takes an object followed by a class as operands and returns true if the object is an instance of the class
    - Syntax: <*is*> → <object> is <class>

- The *equals* operator compares two expressions and returns *true* if the expressions are equal to each other or *false* otherwise
    - Syntax: <*equals*> → <expression> equals <expression>

- The *and* operator compares two expressions and returns *true* if the expressions both evaluate to *true* otherwise it returns *false*
    - Syntax: <*and*> → <expression> and <expression>

- The *or* operator compares two expressions and returns *true* if either of the expressions evaluate to *true* otherwise it returns *false*
    - Syntax: <*or*> → <expression> or <expression>

- The < operator compares two expressions and returns *true* if the left hand expression is strictly less than the right hand expression
    - Syntax: <*less than*> → <expression> < <expression>

- The > operator similarly compares two expressions and returns *true* if the left hand expression is strictly greater than the right hand expression
    - Syntax: <*greater than*> → <expression> > <expression>

- The >= and <= operators perform similarly to the > and < operators in which they compare two expressions and return *true* if the left hand expression is greater than or equal to or less than or equal to the right hand expression respectively
    - Syntax:
        - <*greater than or equal*> → <expression> >= <expression>
        - <*less than or equal*> → <expression> <= <expression>

Arithmetic Operators

- The *+* operator return the sum of the values of two expressions
    - Syntax: <*add*> → <expression> + <expression>

- The - operator returns the difference in value between two expressions
    - Syntax: <*subtract*> → <expression> - <expression>

- The * operator returns the multiplication of two expressions
    - Syntax: <*multiply*> → <expression> * <expression>

- The */* operator returns the division between two expressions
    - Syntax: <*divide*> → <expression> / <expression>

- The ^ operator returns the value of first expression raised to the power of the value of the second expression
    - Syntax: <*raise*> → <expression> ^ <expression>

- The % operator returns the remainder of the division between the first expression and the second expression
    - Syntax: <*modulo*> → <expression> % <expression>

Simple Statements

A simple statement is contained within a singular logical line. Simple statements are expressions, assignments, returns, breaks, continues, import, and locks.

- Return
    - Passes the value given by resolving the return expression back to the caller of the function
    - Syntax: <return> → return <expression>
- Break
    - Stops execution of a loop at the point of the break and resumes execution outside of the loop
    - Syntax: <break> → break
- Continue
    - Stops execution of the iteration of a loop at the point of the continue and resumed execution at the next iteration or outside of the loop if no iterations remain

        - Syntax: <continue> → continue
- Import
    - Allows access to all non-closed classes in a given group
    - Syntax: <import> → import <group>
- Lock
    - See section on Concurrency

See earlier sections for information regarding expression and assignment statements.

Exceptions

An exception occurs when an error is found in the code. Exceptions can occur at runtime or at compile time and the exception is raised at the point where the error is detected. There are many reasons for compiler errors such as incorrect syntax, inconsistent white space, invalid access modifier, or type error and these are the main exceptions the programmer should face. Runtime exceptions occur when errors appear in the code that could not be detected by the compiler such as null points, index out of bounds, or invalid argument exceptions.

Error Handling
Since compile time exceptions occur before runtime they cannot be handled by the code and thus, they must be fixed manually by the programmer. Runtime exceptions however can be handled by the code by either fixing the root cause of the issue or by handling exceptions using the try statement. Specific information regarding the try statement can be found in the next section.

Raising Exceptions
The programmer may also want to raise their own exceptions for cases where other programmers or simply themselves handle their data types incorrectly. For example, if a programmer implemented a new type of array they may need to raise their own runtime exception when the programmer tries to access an index which is out of bounds. The programmer can do this by using the error statement which is composed of the error keyword followed by an exception object. All default exception types can be found in the *ray.exceptions* group but the programmer may choose to create their own exception types inheriting the Exception class (read more on inheritance in the Class Definitions section).

Formal Syntax:
<error statement> → error <exception object>

Code Example:

```
error InvalidArgumentException("x must be > 0")
```

Compound Statements

Compound statements are similar to simple statements except they generally span multiple lines and control the execution of statements in some way. There are four types of compound statements: if, for, while, and try.

If Statement

An if statement evaluates a boolean expression and diverges the execution of statements based on the result. An if statement begins with the if keyword followed by a boolean expression followed by the statements to be executed if the result of the expression is true. An if statement also allows for additional control flow using the else if and else statements. The else if statement allows for evaluating further boolean expressions and diverging execution if the previous expressions have returned false and it returns true. The else statement does not require a boolean expression and can be added to the end of a set of if/if else clauses to diverge execution if all previous statements have returned false.

Syntax:

<if> → if <expression>: <statements>
(else if <expression>: <statements>)*
[else: <statements]

Code Example:

```
Integer x = 5;
if x equals 4:
      // Some code
else if x equals 3:
      // Some code
else:
      // Some code
```

For Statement

The for statement iterates over an iterable object providing one element of the iterable each iteration. For example, one can iterate over a List of Integer in order to sum them up.

Formal Syntax:

<for> → for <element> in <iterable>: <statements>

Code Example:

```
List of Integer iter = {1, 2, 3}
Integer sum = 0
for x in iter:
    sum += x
```

While Statement

The while statement evaluates a boolean expression every iteration and repeats execution of a set of statements as long as the expression stays true.

Formal Syntax:

<while> → while <expression>: <statements>

Example:

```
Integer x = 5;
while x < 10:
    x += 1
```

Try Catch Statement

The Try Catch statement executes a set of statements and upon an exception occurring, instead of halting execution of the program, it calls upon an exception handler in the form of a catch statement which diverges execution to the statements contained in the exception handler.

Formal Syntax:

<try> → try: <statements> (catch <exception type> <identifier>: <statements>)*

Example:

```
Integer x = null
```

```
try:
      Double y = x.toDouble()
catch NullPointerException e:
      e.print()
```

Access Modifiers

Access modifiers control when and if access to an entity is allowed. Access is determined statically at compile time and does not change during execution. Without access to an entity, such as a method, invocation is strictly disallowed. If access to an object is denied, such as an instance of a class or a class member variable, all internal attributes and methods are inaccessible.

There are four types of modifiers:

- Open
  - Can be applied to classes, methods, and variables
  - Classes, methods, and variables declared with the open access modifier are fully accessible
- Children
  - Can be applied to methods and variable but not classes
  - Method and variables declared with the children access modifier within a parent class are only accessible to child classes which inherit from the parent
- Group
  - Can be applied to classes, methods, and variables
  - Classes, methods, and variables declared with the open access modifier can only be accessed by entities within the same group
- Closed
  - Can be applied to inner classes, methods, and variables
  - Classes, methods, and variables declared with the closed access modifier can only be accessed within the class they are declared

Access modifiers are to be declared ahead of class, variable, and method declarations. If no access modifier is declared, the access modifier will default to **closed**.

Formal Syntax:

<modifier> → open | closed | children | group
<method> → <modifier> <return type> <name> (<parameters>)

<variable> → <modifier> <type> <name> = <object>
<class> → <modifier> class <name>

Examples:

- Method declaration

```
children myMethod()
```

- Variable declaration

```
closed Integer number = 5
```

- Class declaration

```
open class MyClass
```

Function Definitions

A function is an isolated set of statements defined within a class that can be executed by calling upon its identifier and returns some value. A function call is an expression which upon evaluation, executes the statements provided by the function and evaluates the return value of the function. Each function can define any amount of parameters which are a set of arguments passed to the function upon function call. The function must declare a type and identifier for each parameter and the caller must pass objects with corresponding types in the correct order when making a call to the function. Each function must also declare a return type, which is the type of object which will be returned upon evaluation of the function. Alternatively, functions can utilize the *none* keyword in place of the return type which designates that the function does not return any value. If the function has a valid return type and a return statement does not appear in the function's statements, a compiler error will occur.

All function calls are dynamically bound to methods unless they are declared static or closed. This dynamic binding allows for inheriting classes to override the function, changing the implementation of the function call while preserving the function signature. A function signature is the set of the function identifier, parameter types, and return type.

Formal Syntax:

<parameters> → <type> <identifier> | <parameters>, <type> <identifier>
<function> → <access modifier> <return type> <identifier>(<parameters>):
<statements>

<passed parameters> → <object> | <passed parameters>, <object>
<function call> → <identifier>(<passed parameters>)

Function Example (with function call):

```
open Integer add(Integer a, Integer b):
       return a + b

Integer sum = add(1, 2)
```

Class Definitions

Classes are the basis of any Ray program, they are custom data types which can be instantiated to create objects. They contain attributes in the form of class variables, constructors to determine how the object is created, and methods which are functions contained in the class which can be called by the objects internally or externally depending on the access modifier. Each class in ray is given its own file and must start with a class declaration statement which is made up of an access modifier, followed by the class keyword, followed by a class identifier which is unique to its group. After the class declaration statement a class can contain any amount of methods or member variables.

Formal Syntax:

<class declaration> → <access modifier> class <identifier>:

Code Example:

```
open class MyCustomDataType:
```

Constructors

Classes are instantiated by calling their constructor which is a function accessed by calling the class identifier in the form of a function call with no arguments. Oftentimes when creating new data types the programmer may want to create their own constructor with additional parameters and handling which can be done by adding a constructor function to the class. Adding a constructor function to a class is as simple as creating a function with an identifier identical to the class identifier. If a class has a custom constructor it must be instantiated using the custom constructor else a compiler error will occur. This means if a class has an access modifier on the

constructor it may not be able to be instantiated by any given class, the modifier must be respected.

Formal Syntax:

<constructor> → <access modifier> <return type> <class identifier>(<parameters>): <statements>

<class instantiation> → <class identifier>(<parameters>)

Code Examples:

```
public MyCustomDataType(Integer x):
    // Some code
MyCustomDataType obj = MyCustomDataType(2)
```

Inheritance

Ray supports inheritance through the *inherits* keyword which can optionally be added to a class declaration statement. The inherits keyword appears at the end of a class declaration and specifies the identifier of an accessible class. This class acts at the basis for this data type and inherits the functions constructor, methods, and member variables. The inheriting class optionally can override the constructor or any methods if the access modifier permits.

Formal Syntax:

<inheriting class declaration> → <access modifier> class <identifier> inherits <class identifier>:

Groups

Groups are Ray's concept of modules which can be thought of as a folder of classes which can be imported and used anywhere. Adding classes to a group is as simple as adding a group declaration statement. A group declaration statement appears at the top-most indentation level of a class, outside of functions, and contains the group keyword followed by an identifier representing the group's names. If 5 separate classes contain the same group declaration, they will all be added to the same group. A class may only have one group declaration statement else the compiler will throw an error. 9

Formal Syntax:

<group declaration> → group <identifier>

Code Example:

```
group MyNewModule
```

Groups also support the concept of sub-groups which are groups created within other groups. In order to create a sub-group one must use the dot notation within the group declaration. For example, a subgroup “simple” of the group “adders” would be declared as “adders.simple” in the declaration statement. If the compiler cannot find the parent group an error will be thrown. Any amount of embedded sub-groups can be created by utilizing multiple dots in the declaration.

Importing Groups

In order to utilize the classes from a given group outside of the group one must use the import statement. Import statements are found at the top-most indentation level of a class, outside of functions and each class may have any amount of import statements. An import statement is made up of the import keyword followed by a group name.

Formal Syntax:

<group> → <group identifier> | <group identifier>.<sub group identifier>
<import> → import <group>

Code Example:

```
import adders.simple
```

Abstract Data Types

Abstract classes are essentially blueprints for creating other classes and cannot be instantiated. An abstract class can act as an incomplete class and is allowed to declare abstract methods which have no body and must be overridden by inheriting classes.

An abstract class is declared by adding the blueprint keyword ahead of the class declaration. Similarly an abstract method can be declared by adding the blueprint keyword ahead of a function declaration but an abstract method cannot contain a function body else a compiler error will occur.

Formal Syntax:

<abstract class declaration → blueprint <access modifier> class <identifier>:
<abstract method declaration → <access modifier> <return type>
<identifier>(<parameters>)

Code Examples:

```
blueprint open class MyCustomDataType:
    blueprint open Integer adder(Integer a, Integer b)
```

Concurrency

Ray has built in support for concurrency which allows for the programmer to asynchronously. Asynchronous code is executed in parallel across multiple threads instead of the standard single-thread approach. Concurrency is accomplished by the programmer using thread objects and the lock statement.

Thread Objects
A new thread can be instantiated by creating a class which inherits the Thread object and overriding the execute method which is the code executed by the thread upon start, then creating an instance of this object. The thread can then be started using the start method which will execute parallel execution of both the main thread and the new thread just created.

Lock Statement
The lock statement accepts a single object as a parameter and executes a set of statements with mutually exclusive access to the object (a lock on the object) then releases the lock once it's finished. All threads are blocked access to the object while it is locked.

Formal Syntax:

<lock> → lock (<object>): <statements>

Code Example:

```
Integer x = 5
lock (x):
    x += 1
```

Generics

Generics are objects of unknown type and can be utilized to create powerful classes and methods with a small amount of code. There exist two concepts of generics in Ray, generic classes and generic type parameters. Generic classes are classes which are

associated with one or more type variables, a classic example of this is the default data type List which is associated with a single type variable which denotes what type of object is stored in the list. Similarly functions can accept generic type parameters and allow functions to accept a wider range of objects than a singular type.

Generic Classes

Generic classes are created by specifying generic terms in the class declaration statement. This can be done by utilizing the *generic* keyword which is followed by parenthesis containing identifiers for any generic types used by the class. These generic type identifiers can then be used in any non-static statements within the class such as method return or parameter types and variable types. When a generic class is instantiated the programmer must provide the types associated with the class using the *of* keyword.

Formal Syntax:

<generic types> → <type identifier> | <type identifier>, <generic types>
<generic class declaration> → <access modifier> generic(<generic types>) class <identifier>:
<types> → <type> | <type>, <types>
<of> → <type> of <types> | <type> of <of>

Code Example:

```
public generic(T) Array:
            T someVariable
            public T print():
                  // Some code

      Array of String = Array of String()
```

Generic Type Parameters

Similarly to generic classes, parameters can also specify their own generic types using the generic keyword. This allows methods to accept parameters of any type.

Formal Syntax:

<generic types> → <type identifier> | <type identifier>, <generic types>
<generic function> → <access modifier> generic(<generic types>) <return type> <identifier>(<parameters>): <statements>

Code Example:

```
public generic(E) none print(E obj):
            obj.print()
```