LAB EVALUATION 1

LANGUAGE DESCRIPTION

19CSE401 – COMPILER DESIGN

LANGUAGE – COOL

<u>TEAM MEMBERS</u>

S. NO	NAME	ROLL NUMBER
1	PRADEEP KARTHIK M	CB.EN.U4CSE20447
2	KARTHIKEYAN P	CB.EN.U4CSE20429
3	LOGESWARAN S R	CB.EN.U4CSE20435
4	ADITHYA S	CB.EN.U4CSE20403

CONTENTS

GROUP – 14

COOL Language Design Documentation

Overview:

COOL is an acronym for Classroom Object-Oriented Language. It's a pedagogical tool specifically engineered for teaching compiler construction courses. With its well-defined structure and a moderate degree of complexity, COOL provides students with a solid foundation in the principles and techniques used to design and implement modern compilers without overwhelming them with too many intricate language features.

Key Features:

- ➔ **Object-Oriented:** As the name suggests, COOL is an object-oriented language. It allows for class definitions, inheritance, and method overriding, giving students a taste of how OO principles are applied in compiler construction.
- ➔ **Static Typing:** COOL is statically-typed, which means type checking occurs at compile-time, introducing students to the intricacies of type analysis in the context of compilers.
- ➔ **Rich Set of Control Structures:** While COOL is concise, it encompasses a reasonable set of control structures such as loops and conditional statements.
- ➔ **Simplified Memory Model:** To focus on the core aspects of compiler construction, COOL abstracts away many of the complexities associated with memory management.

LEXER:

In this section, we'll discuss the different types of tokens in COOL language and how they are an identified by the Lexer using their RE (Regular Expression).

TOKENS IN COOL:

1. **KEYWORDS:** These are reserved words in a language that have a special meaning and cannot be used for any other purpose:

KEYWORD	REGULAR EXPRESSION
CLASS	'[cC][lL][aA][sS][sS]'
INHERITS	'[iI][nN][hH][eE][rR][iI][tT][sS]'
NEW	'[nN][eE][wW]'
NOT	'[nN][oO][tT]'
TRUE	'true'
FALSE	'false'
IF	'[iI][fF]'
THEN	'[tT][hH][eE][nN]'
ELSE	'[eE][lL][sS][eE]'
FI	'[fF][iI]'
WHILE	'[wW][hH][iI][lL][eE]'
LOOP	'[lL][oO][oO][pP]'
POOL	'[pP][oO][oO][lL]'
CASE	'[cC][aA][sS][eE]'
OF	'[oO][fF]'
ESAC	'[eE][sS][aA][cC]'
LET	'[lL][eE][tT]'
IN	'[iI][nN]'

ISVOID	‘[iI][sS][vV][oO][iI][dD]’

- **CLASS** : Used to define a new class in Cool.
  - Regular Expression - ‘[cC][lL][aA][sS][sS]’
- **INHERITS** : Is a keyword is used to specify that a class inherits attributes and methods from another class.
  - Regular Expression - **‘[iI][nN][hH][eE][rR][iI][tT][sS]’**
- **NEW** : Is a keyword is used to instantiate a new object of a specified class.
  - Regular Expression - **‘[nN][eE][wW]’**
- **NOT** : Is a keyword is used to perform logical negation on a boolean expression.
  - Regular Expression - **‘[nN][oO][tT]’**
- **TRUE** : Represents the boolean value "true."
  - Regular Expression - **‘true’**
- **FALSE** : Represents the boolean value "false."
  - Regular Expression - **‘false’**
- **IF** : Is a keyword introduces a conditional statement that executes a block of code based on whether a given expression evaluates to true or false.
  - Regular Expression - **‘[iI][fF]’**
- **THEN** : Is a keyword follows the IF condition, specifying the block of code to execute if the condition evaluates to true.
  - Regular Expression - **‘[tT][hH][eE][nN]’**
- **ELSE** : Is a keyword introduces a block of code to execute if the preceding IF condition evaluates to false.
  - Regular Expression - **‘[eE][lL][sS][eE]’**

- **FI** : Ends a conditional statement.
  - Regular Expression - **‘[fF][iI]’**
- **WHILE** : Initiates a while loop.
  - Regular Expression - **‘[wW][hH][iI][lL][eE]’**
- **LOOP** : Specifies the body of a while loop.
  - Regular Expression - **‘[lL][oO][oO][pP]’**
- **POOL** : Ends a loop construct.
  - Regular Expression - **‘[pP][oO][oO][lL]’**
- **CASE** : Initiates a case expression.
  - Regular Expression - **‘[cC][aA][sS][eE]’**
- **OF** : Specifies branches in a case expression.
  - Regular Expression - **‘[oO][fF]’**
- **ESAC** : Ends a case expression.
  - Regular Expression - **‘[eE][sS][aA][cC]’**
- **LET** : Introduces local variables in a let expression.
  - Regular Expression - **‘[lL][eE][tT]’**
- **IN** : Specifies the scope of local variables in a let expression.
  - Regular Expression - **‘[iI][nN]’**
- **ISVOID** : Checks if an expression evaluates to "void."
  - Regular Expression - **‘[iI][sS][vV][oO][iI][dD]’**

2. **<u>PUNCTUATION:</u>** These are special characters used to separate statements and structure the code:

PUNCTUATION	**REGULAR EXPRESSION**
SEMICOLON	‘;’
OPENBRACE	‘{‘

CLOSEBRACE	‘}’
COLON	‘:’
COMMA	‘,’
OPENPARANTHESIS	‘(‘
CLOSEPARANTHESIS	‘)’
DOT	‘.’
AT	‘@’
CASEASSIGN	‘=>’
WHITESPACES	‘[\t\r\n]+’

- **SEMICOLON (;)**: Represents the semicolon symbol, used to terminate statements in Cool.
    - Regular Expression - **‘;’**
- **OPENBRACE ({)**: Represents the opening curly brace symbol, used to begin a block of code.
    - Regular Expression - **‘{‘**
- **CLOSEBRACE (})**: Represents the closing curly brace symbol, used to end a block of code.
    - Regular Expression - **‘}’**
- **COLON (:)**: Represents the colon symbol, used to specify types, labels, or in case expressions.
    - Regular Expression - **‘:’**
- **COMMA (,)**: Represents the comma symbol, used to separate items in a list or arguments in function calls.
    - Regular Expression - **‘,’**
- **OPENPARENTHESES (()**: Represents the opening parenthesis symbol, used to begin function arguments and expressions.

- Regular Expression - **‘(‘**

- **CLOSEPARENTHESES ())**: Represents the closing parenthesis symbol, used to end function arguments and expressions.
  - Regular Expression - **‘)’**
- **DOT (.)**: Represents the dot symbol, used to access attributes and methods of objects.
  - Regular Expression - **‘.’**
- **AT (@)**: Represents the "at" symbol, used for annotation or as a special symbol in Cool.
  - Regular Expression - **‘@’**
- **CASEASSIGN (=>)**: Represents the case assignment symbol, used in case branches to map pattern matches to expressions.
  - Regular Expression - **‘=>’**
- **WHITESPACES (" ")**: Represents whitespace characters, such as spaces, tabs, and newlines, used for separating tokens and ignored during parsing.
  - Regular Expression - **‘ ‘**

3. **OPERATORS:** These are symbols that perform operations on one or more operands:

OPERATORS	REGULAR EXPRESSION
INTEGER COMPARISON	‘~’
ADDITION	‘+’
SUBTRACTION	‘-‘
MULTIPLICATION	‘*’
DIVISION	‘/’

EQUAL	‘=’
LESS THAN	‘<’
LESS THAN OR EQUAL TO	‘<=’
ASSIGN	‘<-‘

- **INTEGER-COMPARISON ('~')**: Represents the tilde character '~' used in integer comparisons.
    - Regular Expression - **‘~’**
- **ADDITION ('+')**: Represents the plus sign '+' used for addition operations.
    - Regular Expression - **‘+’**
- **SUBTRACTION ('-')**: Represents the minus sign '-' used for subtraction operations.
    - Regular Expression - **‘-’**
- **MULTIPLICATION ('*')**: Represents the asterisk '*' used for multiplication operations.
    - Regular Expression - **‘*’**
- **DIVISION ('/')**: Represents the forward slash '/' used for division operations.
    - Regular Expression - **‘/’**
- **EQUAL ('=')**: Represents the equal sign '=' used for equality comparisons.
    - Regular Expression - **‘=’**
- **LESS THAN ('<')**: Represents the less than sign '<' used for less than comparisons.
    - Regular Expression - **‘<’**

- **LESS THAN OR EQUAL TO ('<=')**: Represents the less than or equal to sign '<=' used for less than or equal to comparisons.
    - Regular Expression - **‘<=’**
- **ASSIGN ('<-')**: Represents the assignment operator '<-' used for variable assignments.
    - Regular Expression - **‘<-’**

OPERATOR PRECEDENCE IN COOL:

.	Left-associative
@	Left-associative
~	Left-associative
isvoid	Left-associative
*, /	Left-associative
+, -	Left-associative
<=, <, =	Left-associative
not	Left-associative
<-	Right-associative

4. **LITERALS:** Constants that are used directly in the code:

LITERALS	REGULAR EXPRESSION

STRING	'"' (('\\'|'\t'|'\r\n'|'\r'|'\n'|'\\"') | ~('\\'|'\t'|'\r'|'\n'|'"'))* '"'
NUM	[0-9]+
TRUE	‘true’
FALSE	‘false’

- ➢ **STRING**: Represents a string enclosed in double quotes, possibly containing escaped characters.
  - Regular Expression - '"' (('\\'|'\t'|'\r\n'|'\r'|'\n'|'\\"') | ~('\\'|'\t'|'\r'|'\n'|'"'))* '"'
- ➢ **NUM**: Represents a numeric value consisting of one or more digits.
  - Regular Expression - **[0-9]+**
- ➢ **TRUE**: Represents the boolean value 'true'.
  - Regular Expression - **‘true’**
- ➢ **FALSE**: Represents the boolean value 'false'.
  - Regular Expression - **‘false’**

5. **IDENTIFIERS**: Names given to various programming constructs such as variables, classes, etc:

IDENTIFIERS	REGULAR EXPRESSION
ID	[a-z_][a-zA-Z0-9_]*
CLASSTYPE	[A-Z][a-zA-Z_0-9]*

- ➢ **ID**: Represents an identifier starting with a lowercase letter or underscore, followed by alphanumeric characters or underscores.

- Regular Expression - **[a-z_][a-zA-Z0-9_]***

➢ **CLASSTYPE**: Represents a class type identifier starting with an uppercase letter, followed by alphanumeric characters or underscores.

- Regular Expression - **[A-Z][a-zA-Z_0-9]***

6. **COMMENTS**: Used for making explanatory remarks in the code. They don't have any effect on the execution of the program.

COMMENT	REGULAR EXPRESSION
SINGLE COMMENT	'--' ~[\r\n]* -> skip
MULTI COMMENT	\ '(*' .*? '*)' -> skip;

➢ SINGLE COMMENT: Represents a single-line comment starting with '--' and extending until the end of the line.

- Regular Expression - **'--' ~[\r\n]* -> skip**

➢ MULTI COMMENT: Represents a multi-line comment enclosed in '(*' and '*)'.

- Regular Expression - **\'(*' .*? '*)' -> skip;**

SYNTAX ANALYSER (PARSER):

In this section, we will look at the syntax of the COOL language and we'll write parser for understanding the syntax of COOL.

We'll try to define the syntax of the language in the bottoms-up approach where we start by defining a expression and then progressively move on to defining the final COOL program.

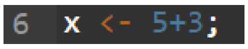
EXPRESSIONS:

EXPRESSION	CFG
ASSIGNMENT	expr ::= ID ASSIGN expr
ACCESSING METHODS OF OBJECT	expr::= expr typedec DOT ID OPENPARENTHESES expr extraparams CLOSEPARENTHESES typedec::= AT TYPE | ε extraparams::= COMMA expr extraparams | ε
CALLING A FUNCTION	expr::= ID OPENPARENTHESES expr extraparams CLOSEPARENTHESES extraparams::= COMMA expr extraparams | ε
IF ELSE	expr::= IF expr THEN expr ELSE expr FI
WHILE LOOP	expr::= WHILE expr LOOP expr POOL
BLOCK	expr::= OPENBRACE stmt CLOSEBRACE stmt::= expr SEMICOLON stmt | ε
LET	expr::= LET ID COLON TYPE ASSIGN expr extralet IN expr

	extralet::= COMMA ID COLON TYPE ASSIGN expr extralet | ε
CASE	expr::= CASE expr OF caserepeat ESAC caserepeat::= ID COLON TYPE CASEASSIGN expr SEMICOLON casecontinue casecontinue::= ID COLON TYPE CASEASSIGN expr SEMICOLON casecontinue | ε
NEW	expr::= NEW TYPE
ISVOID	expr::= ISVOID expr
ARITHMETIC	expr::= expr MUL expr | expr DIV expr | expr ADD expr | expr SUB expr
BOOLEAN	expr::= INTCOMP expr | expr LT expr | expr LTEQ expr | expr EQUAL expr | NOT expr
TERMINAL EXPRESSIONS	expr::= | STRING | NUM | ID

	| TRUE | FALSE | ε
FORMAL PARAMETER DECLARATION	formal ::= ID COLON TYPE
FUNCTIONS	feature ::= ID OPENPARENTHESES formal formalrepeat CLOSEPARENTHESES COLON TYPE OPENBRACE expr CLOSEBRACE | ID COLON TYPE ASSIGN expr formalrepeat::= COMMA formal formalrepeat | ε
CLASS	class ::= CLASS TYPE classinherit OPENBRACE featurepeat CLOSEBRACE SEMICOLON classinherit::= INHERITS TYPE | ε featurepeat::= feature SEMICOLON featurepeat | ε
PROGRAM	program::= classrepeat classrepeat::= class classcontinue classcontinue::= class classcontinue | ε

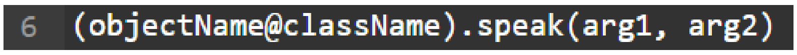
- **Assignment Expression:** Is an expression used to assign a value to a Identifier.
  - Example:



```
x <- 5+3;
```

  - **CFG** - expr ::= ID ASSIGN expr

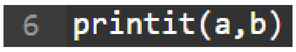
- **Accessing methods of an object:** Is a function call using an object of a certain class.
  - Example:



```
(objectName@className).speak(arg1, arg2)
```

  - **CFG** –
  - expr::= expr typedec DOT ID OPENPARENTHESES expr extraparams CLOSEPARENTHESES
  - typedec::= AT TYPE | ε
  - extraparams::= COMMA expr extraparams | ε

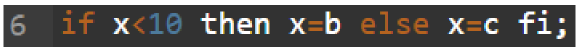
- **Calling a function:** Is a function to a method within the class
  - Example:



```
printit(a,b)
```

  - **CFG** –
  - expr::= ID OPENPARENTHESES expr extraparams CLOSEPARENTHESES
  - extraparams::= COMMA expr extraparams | ε

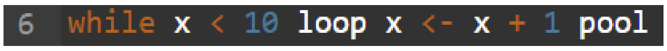
- **If Else:** Is a conditional flow control expression
  - Example:



```
if x<10 then x=b else x=c fi;
```

  - **CFG** - expr::= IF expr THEN expr ELSE expr FI

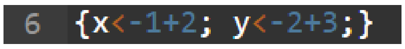
- **While Loop:** A loop that repeatedly evaluates a body of expressions as long as a specified condition remains true.
  - Example:



```
while x < 10 loop x <- x + 1 pool
```

  - **CFG** - expr::= WHILE expr LOOP expr POOL

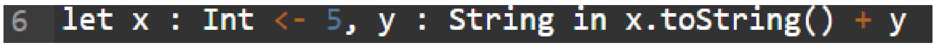
- **Block:** A block of one or more expressions separated by semicolons, enclosed within braces.
  - Example:



```
{x<-1+2; y<-2+3;}
```

  - **CFG** –
  - expr::= OPENBRACE stmt CLOSEBRACE
  - stmt::= expr SEMICOLON stmt | ε

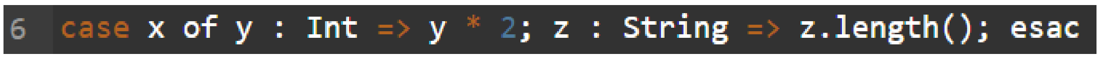
- **LET:** A let binding declaring one or more typed variables, optionally with initial values, for use in a subsequent expression.
  - Example:



```
let x : Int <- 5, y : String in x.toString() + y
```

  - **CFG** –
  - expr::= LET ID COLON TYPE ASSIGN expr extralet IN expr
  - extralet::= COMMA ID COLON TYPE ASSIGN expr extralet | ε

- **CASE:** A case expression evaluating expr and branching based on its type to execute the corresponding expression.
  - Example:



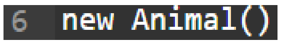
```
case x of y : Int => y * 2; z : String => z.length(); esac
```

  - **CFG** –
  - expr::= CASE expr OF caserepeat ESAC

- caserepeat::= ID COLON TYPE CASEASSIGN expr SEMICOLON casecontinue
- casecontinue::= ID COLON TYPE CASEASSIGN expr SEMICOLON casecontinue | ε

➢ **NEW:** Instantiation of a new object of a specified TYPE in the COOL language.

- Example:

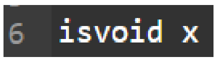


```
new Animal()
```

- **CFG** - expr::= NEW TYPE

➢ **ISVOID:** An expression that checks if the evaluated expr is void in the COOL language.

- Example:

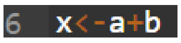


```
isvoid x
```

- **CFG** - expr::= ISVOID expr

➢ **Arithmetic Expression:** An expression that performs mathematical operations on numbers.

- Example:



```
x<-a+b
```

- **CFG** –

  expr::= expr MUL expr

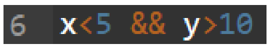
  | expr DIV expr

  | expr ADD expr

  | expr SUB expr

➢ **Boolean Expression:** An expression that evaluates to either true or false.

- Example:



```
x<5 && y>10
```

- **CFG:**

  expr::= INTCOMP expr # invert

  | expr LT expr

  | expr LTEQ expr

  | expr EQUAL expr

  | NOT expr

➢ **Terminal Expressions:** Used to provide a termination to the tree forming the expression.

- **CFG:**

  expr::= | STRING

  | NUM

  | ID

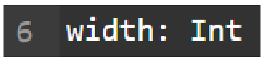
  | TRUE

  | FALSE

  | ε

<u>**FORMAL PARAMETER DECLARATION:**</u> A declaration of a parameter with a specified name (ID) and type (TYPE) in the COOL language.

- Example:

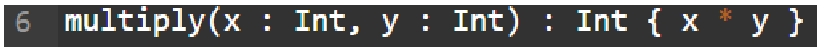


```
width: Int
```

- **CFG:** formal ::= ID COLON TYPE

FUNCTIONS (FEATURES): A feature can be either a method definition with parameters and a return type or an attribute declaration with an optional initialization.

- Example:

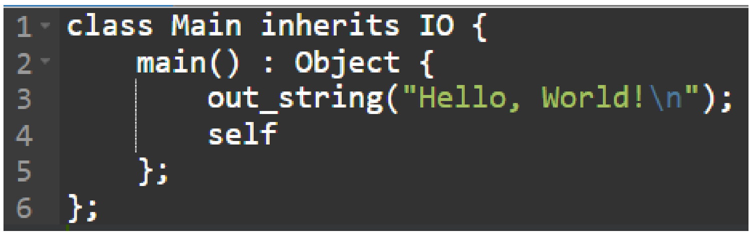


```
multiply(x : Int, y : Int) : Int { x * y }
```

- **CFG:**
- feature ::= ID OPENPARENTHESES formal formalrepeat CLOSEPARENTHESES COLON TYPE OPENBRACE expr CLOSEBRACE | ID COLON TYPE ASSIGN expr
- formalrepeat::= COMMA formal formalrepeat | ε

CLASS: A class is a blueprint for creating objects that encapsulates data for the object and methods to manipulate that data.

- Example:

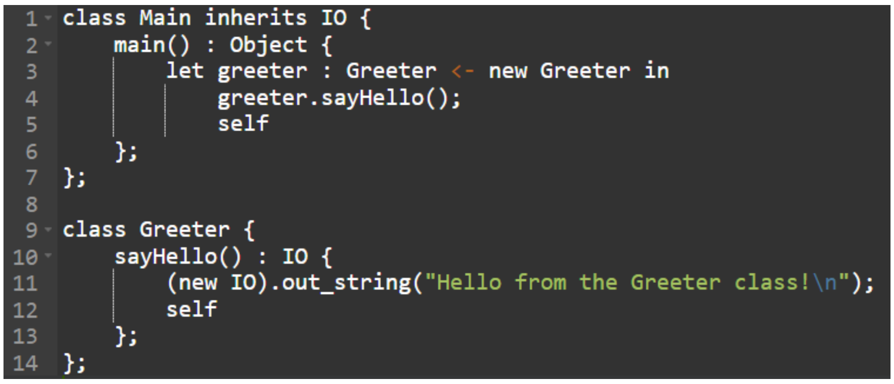


```
class Main inherits IO {
    main() : Object {
        out_string("Hello, World!\n");
        self
    };
};
```

- **CFG:**
- class ::= CLASS TYPE classinherit OPENBRACE featurepeat CLOSEBRACE SEMICOLON
- classinherit::= INHERITS TYPE | ε
- featurepeat::= feature SEMICOLON featurepeat | ε

PROGRAM: A Program is a collection of classes which can be executed to provide an output.

- Example:


```
class Main inherits IO {
    main() : Object {
        let greeter : Greeter <- new Greeter in
            greeter.sayHello();
            self
    };
};

class Greeter {
    sayHello() : IO {
        (new IO).out_string("Hello from the Greeter class!\n");
        self
    };
};
```

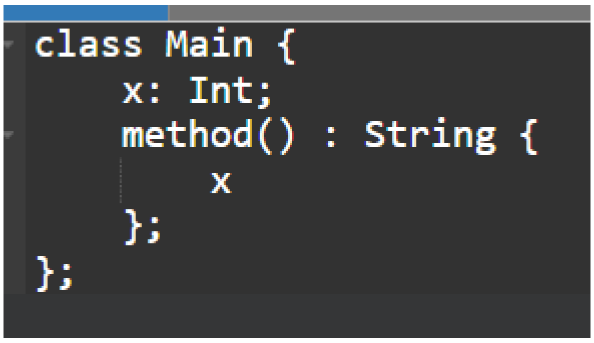
- **CFG:**
- program::= classrepeat
- classrepeat::= class classcontinue
- classcontinue::= class classcontinue | ε

<u>SEMANTIC ANALYSIS:</u> In this section we'll look at the Semantic Rules that need to be followed by a COOL program in order to be error-free, we'll list down the rules and also provide example for each case where the program falters to follow the rule.

<u>TYPE CHECKING:</u>

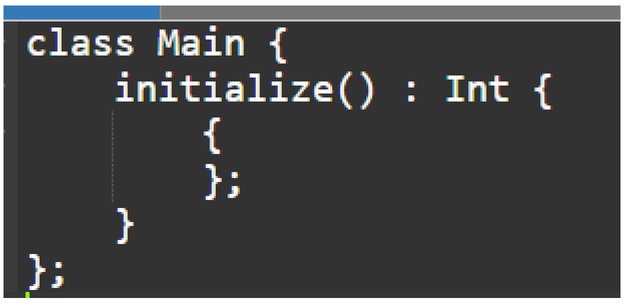
1. **Attributes and Method Return Types**: The type of every attribute and method return expression should conform to the declared type of that attribute or method.

- **Example**:


```
class Main {
    x: Int;
    method() : String {
        x
    };
};
```

2. **Constructor**: A constructor (initializer) for a class should not have a return type.

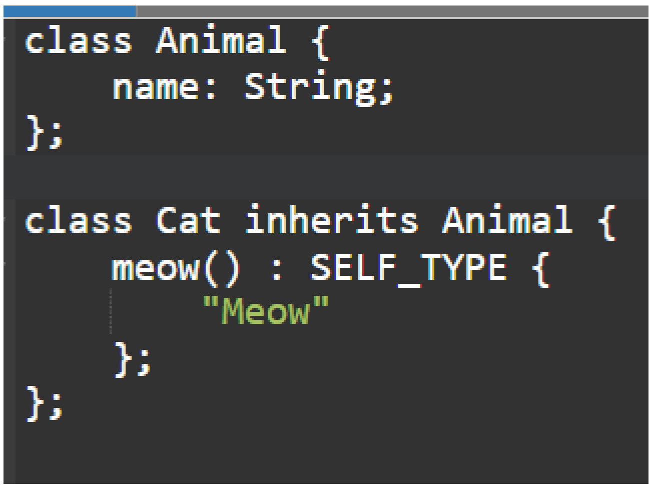
- **Example**:



```
class Main {
    initialize() : Int {
        {
        };
    }
};
```

3. **Self-Type**: If an expression has type **SELF_TYPE**, it takes the type of the current class in which it appears.

- **Example**:



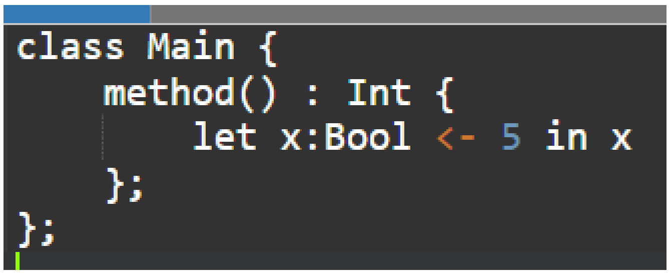
```
class Animal {
    name: String;
};

class Cat inherits Animal {
    meow() : SELF_TYPE {
        "Meow"
    };
};
```

TYPE INFERENCE:

1. **Let Expressions**: For **let x:T <- e1 in e2**, the type of **e1** must conform to **T**. If there's no initialization, default values are used based on type (e.g., **0** for **Int**, **false** for **Bool**).

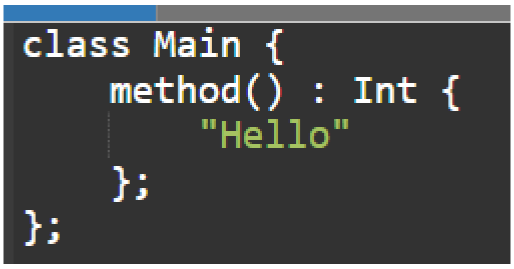
- **Example**:



```
class Main {
    method() : Int {
        let x:Bool <- 5 in x
    };
};
```

2. **Method Return**: The return type of a method can be inferred from its **return** expression.

- **Example**:

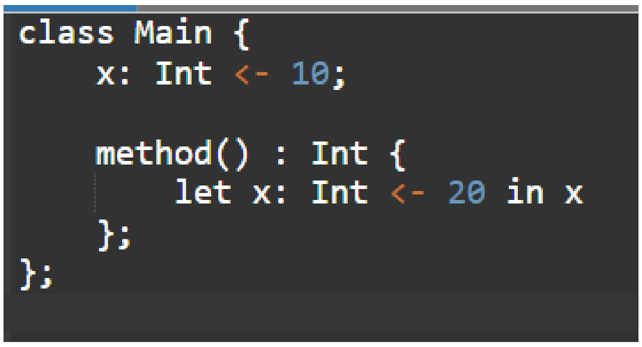


```
class Main {
    method() : Int {
        "Hello"
    };
};
```

SCOPE RESOLUTION:

1. **Nested Scopes**: Variables defined in an inner scope (e.g., inside a **let**, **while**, or method) shadow variables of the same name from an outer scope.

- **Example**:

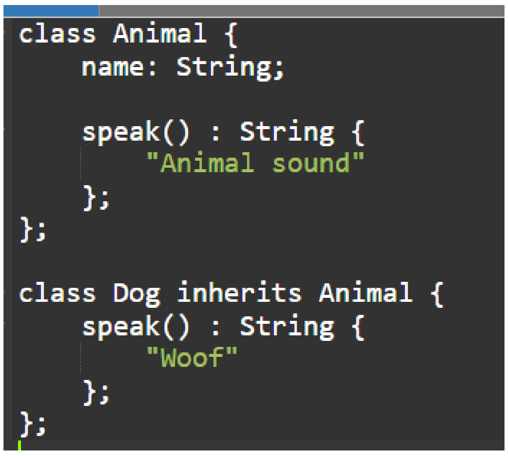


```
class Main {
    x: Int <- 10;

    method() : Int {
        let x: Int <- 20 in x
    };
};
```

2. **Method and Attribute Names**: In the scope of a class, its methods and attributes are accessible. An inherited method or attribute is shadowed if redefined in a subclass.

- **Example**:



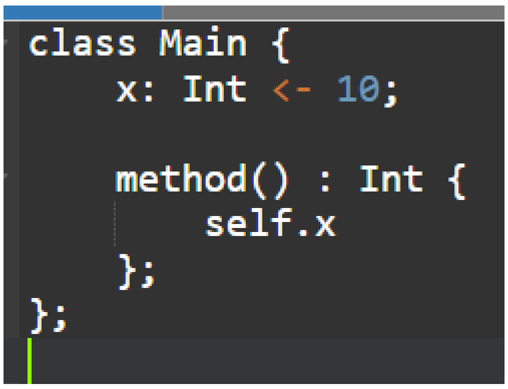
```
class Animal {
    name: String;

    speak() : String {
        "Animal sound"
    };
};

class Dog inherits Animal {
    speak() : String {
        "Woof"
    };
};
```

3. **Self Variable**: The **self** variable is implicitly defined in the scope of every method, referring to the current object.

- **Example**:

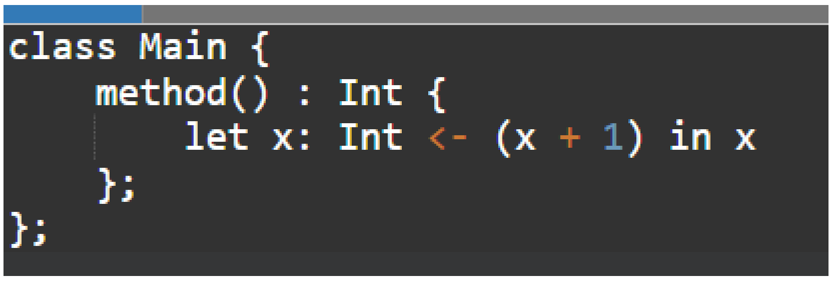


```
class Main {
    x: Int <- 10;

    method() : Int {
        self.x
    };
};
```

4. **Let and Case Scopes**: **let** and **case** introduce new nested scopes. In **let x:T <- e1 in e2**, **x** is in the scope of **e2** but not **e1**.

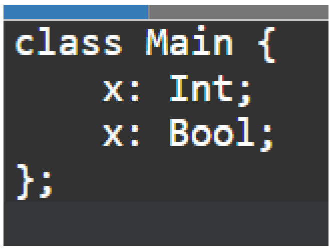
- **Example**:



```
class Main {
    method() : Int {
        let x: Int <- (x + 1) in x
    };
};
```

5. **No Redefinitions**: Within a single scope, no name can be defined more than once, be it a variable, method, or class name.

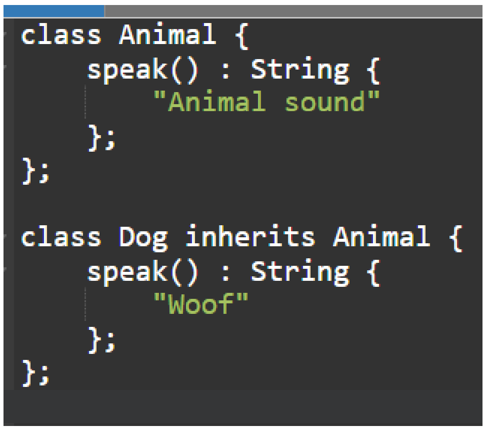
- **Example**:



```
class Main {
    x: Int;
    x: Bool;
};
```

6. **Inherited Attributes and Methods**: A class inherits attributes and methods from its ancestors. If it redefines a method, the new method shadows the old one.

- **Example**:



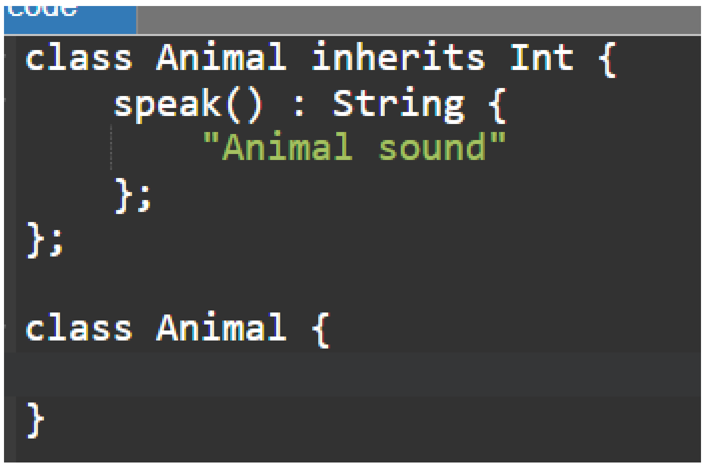
```
class Animal {
    speak() : String {
        "Animal sound"
    };
};

class Dog inherits Animal {
    speak() : String {
        "Woof"
    };
};
```

OTHER SEMANTIC RULES:

1. **Inheritance**:

    - A class cannot inherit from the following classes: **SELF_TYPE**, **String**, **Int**, and **Bool**.
    - No class can be defined more than once.
    - Example:



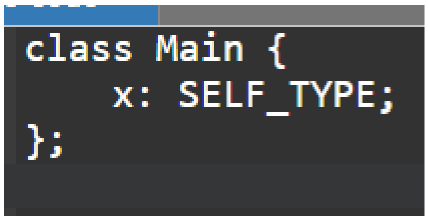
```
class Animal inherits Int {
    speak() : String {
        "Animal sound"
    };
};

class Animal {

}
```

2. **Self**:

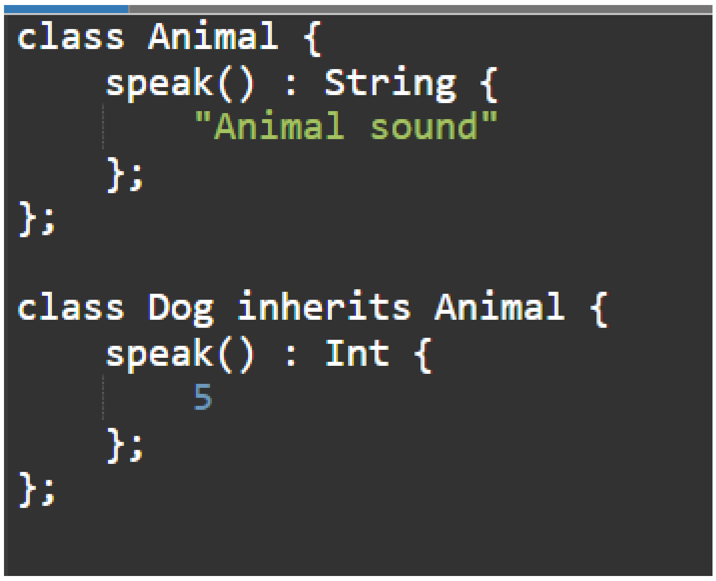
    - The type **SELF_TYPE** is only allowed in certain contexts, such as the return type of a method.
    - Example:



```
class Main {
    x: SELF_TYPE;
};
```

3. **Method Definitions**:

- In a class **C**, the redefined method in its subclass must have the same number of formal arguments as in **C**, and the types of the formal arguments in the redefined method must be the same as those in **C**.
- If **f** in class **B** overrides **f** in class **A**, then the return type of **f** in **B** must conform to the return type of **f** in **A**.
- Example:

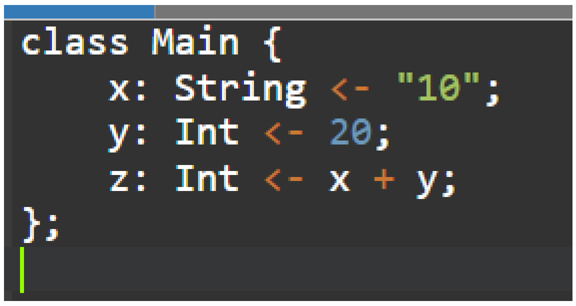


```
class Animal {
    speak() : String {
        "Animal sound"
    };
};

class Dog inherits Animal {
    speak() : Int {
        5
    };
};
```

4. **Arithmetic Operations**:

- Both sides of arithmetic operations (**+**, **-**, *****, **/**) must be of type **Int**.
- Example:

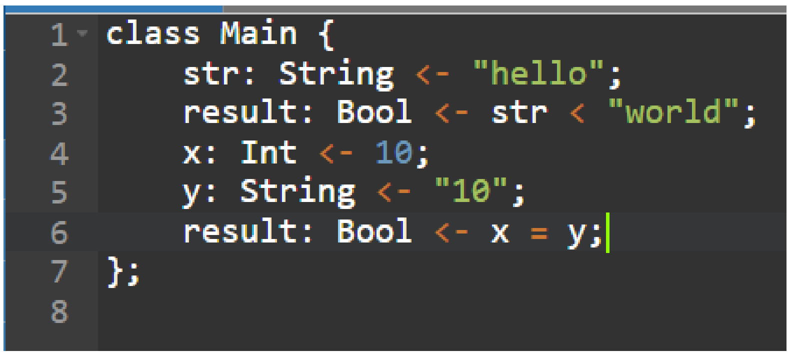


```
class Main {
    x: String <- "10";
    y: Int <- 20;
    z: Int <- x + y;
};
```

-

5. **Comparison Operations**:

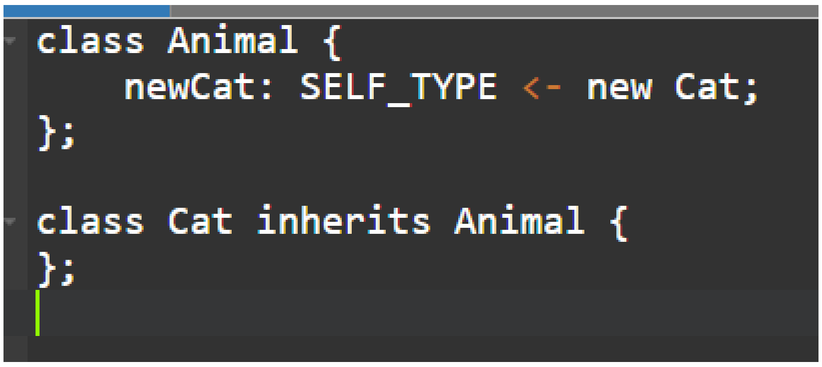
- Both sides of comparison operations (**<**, **<=**) must be of type **Int**.
- The **=** comparison can be used to compare two integers, two strings, two Booleans, or two objects of the same class. Comparing objects of different types is not allowed.
- Example:



```
class Main {
    str: String <- "hello";
    result: Bool <- str < "world";
    x: Int <- 10;
    y: String <- "10";
    result: Bool <- x = y;
};
```

6. **Object Instantiation**:

- An object of type **T** can be instantiated with the **new** keyword followed by **T**. If **T** is **SELF_TYPE**, the instantiated object has the same type as the current class.
- Example:

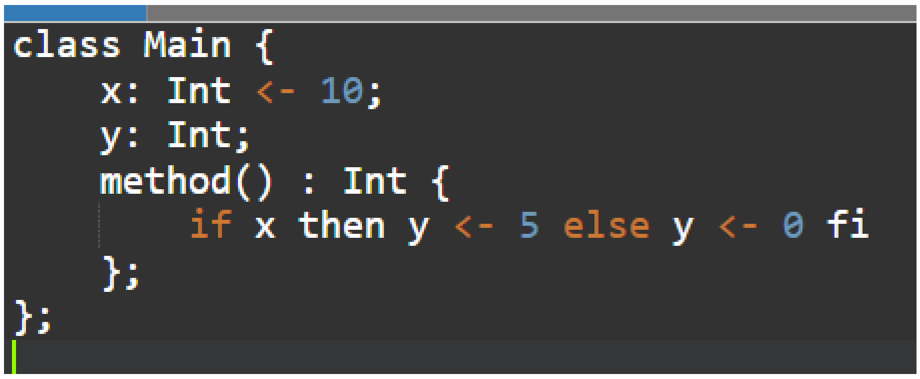


```
class Animal {
    newCat: SELF_TYPE <- new Cat;
};

class Cat inherits Animal {
};
```

7. **Conditional (If) Expressions**:

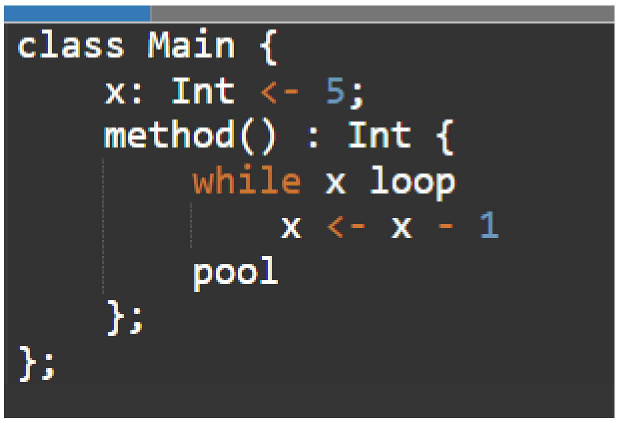
- The predicate of an **if** expression must have type **Bool**.
- The type of the entire **if** expression is the least upper bound of the then and else branches.
- Example:



```
class Main {
    x: Int <- 10;
    y: Int;
    method() : Int {
        if x then y <- 5 else y <- 0 fi
    };
};
```

8. **Loops (While)**:

   - The condition of a **while** loop must be of type **Bool**.
   - The type of a **while** loop expression is always **Object**.
   - Example:

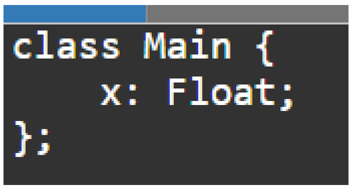


```
class Main {
    x: Int <- 5;
    method() : Int {
        while x loop
            x <- x - 1
        pool
    };
};
```

<u>ERROR CHECKING:</u>

1. **Type Existence:**

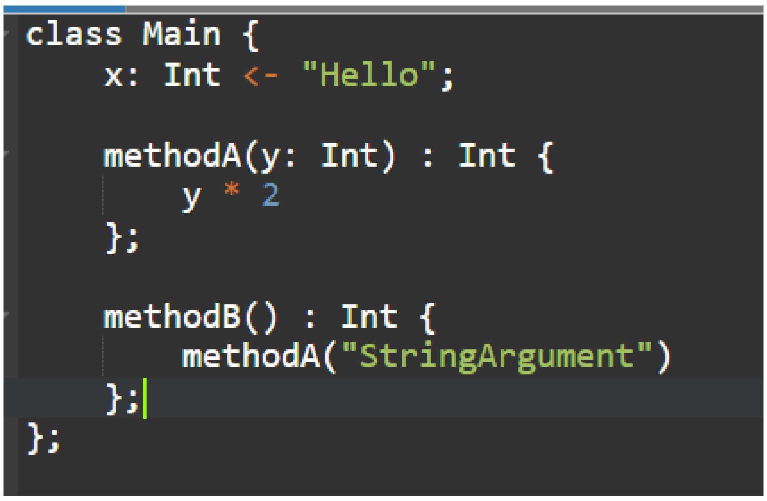
   - Check if all types/classes referenced in the program exist. This includes checking the types in variable declarations, inheritance relationships, and method calls.
   - Example:



```
class Main {
    x: Float;
};
```

2. **Type Matching**

- Ensure that the type of the right-hand side of an assignment matches the type of the left-hand side.
- Ensure actual method arguments match the expected types of formal parameters.
- Example:



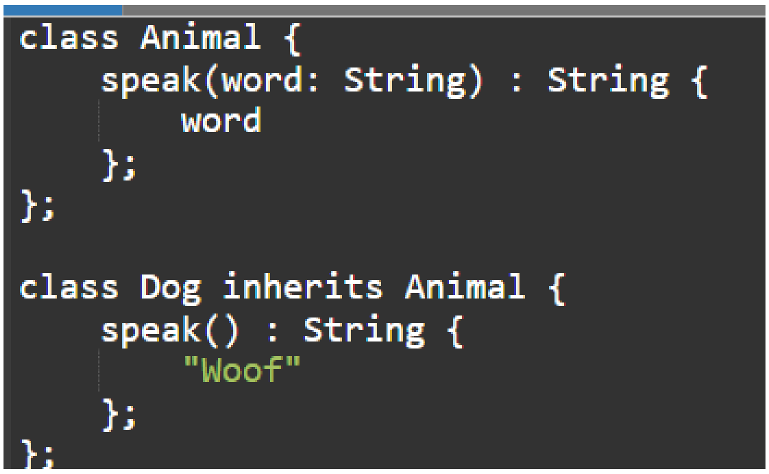
```
class Main {
    x: Int <- "Hello";

    methodA(y: Int) : Int {
        y * 2
    };

    methodB() : Int {
        methodA("StringArgument")
    };
};
```

3. **Method Overriding**

- In the COOL language, a class can override methods of its parent class. When this happens, the child class method should have the same signature as the parent class method. Ensure that overridden methods have matching argument types and return types.
- Example:

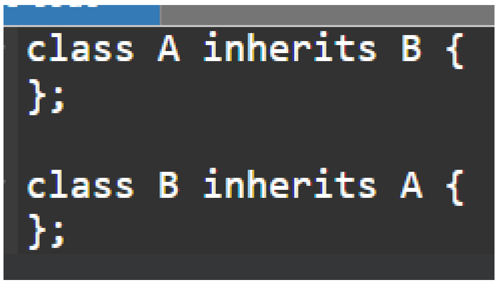


```
class Animal {
    speak(word: String) : String {
        word
    };
};

class Dog inherits Animal {
    speak() : String {
        "Woof"
    };
};
```

4. **Inheritance Cycles**

    - The COOL language does not allow cyclic inheritance, so you should check for cycles in the inheritance graph.
    - Example:

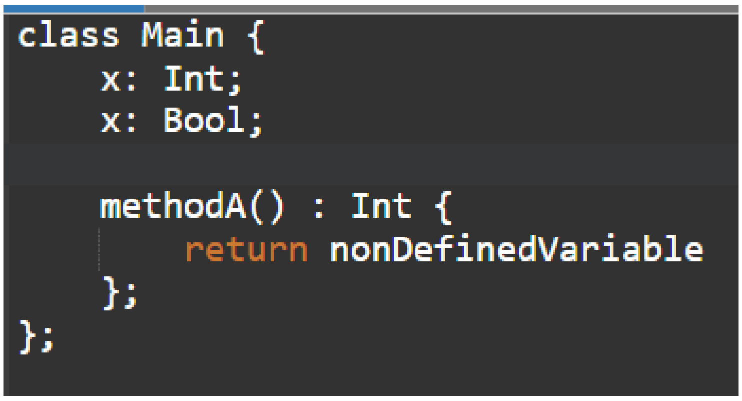


```
class A inherits B {
};

class B inherits A {
};
```

5. **Variable Binding**

    - Check that every identifier (like variable or method name) used is defined.
    - Ensure variables aren't redefined in the same scope.
    - Example:

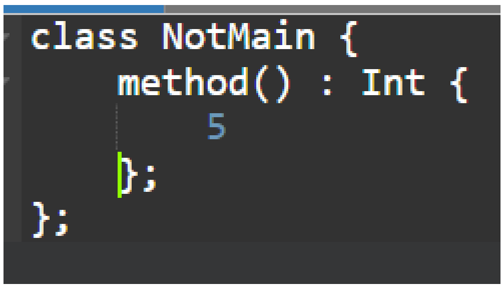


```
class Main {
    x: Int;
    x: Bool;

    methodA() : Int {
        return nonDefinedVariable
    };
};
```

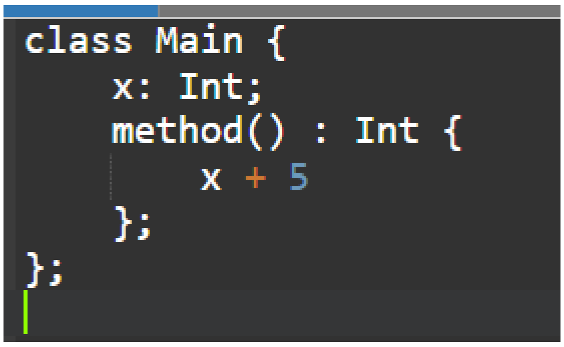
6. **Main Class and Method Existence**

    - Every COOL program must have a **Main** class with a **main** method that has no parameters. This rule ensures that the program has an entry point.
    - Example:


```
class NotMain {
    method() : Int {
        5
    };
};
```

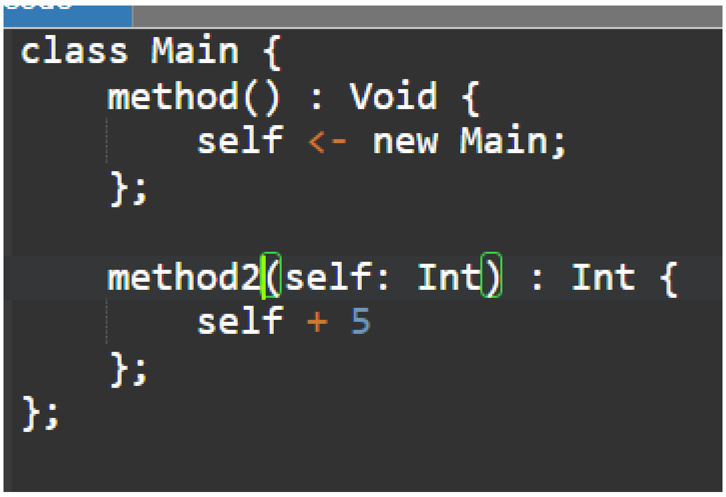
7. **Initialization Before Use**

   - Ensure that variables are initialized before they are used.
   - Example:


```
class Main {
    x: Int;
    method() : Int {
        x + 5
    };
};
```

8. **Correct Self Keyword Usage**

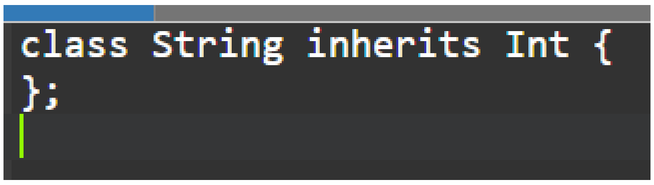
   - The keyword **self** is reserved in COOL. It should not be assigned to.
   - The type of **self** should not be explicitly named in a let binding or as the type of a formal parameter.
   - Example:


```
class Main {
    method() : Void {
        self <- new Main;
    };

    method2(self: Int) : Int {
        self + 5
    };
};
```

9. **Native Classes Immutability**

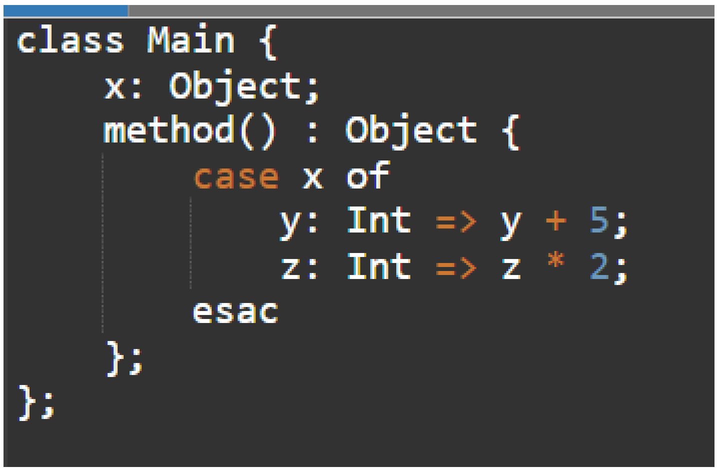
   - Classes like **Int**, **String**, and **Bool** are considered native/built-in, and you cannot redefine or inherit from them.
   - Example:



```
class String inherits Int {
};
```

10. **Case Expression**

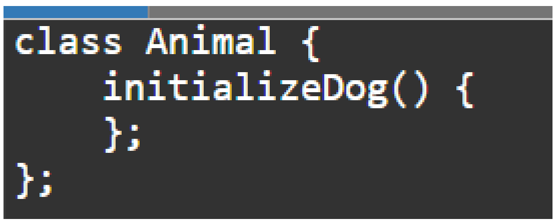
- The branches in a **case** expression must have distinct types.
- The type of the case expression is the least upper bound of the types of its branches.
- Example:



```
class Main {
    x: Object;
    method() : Object {
        case x of
            y: Int => y + 5;
            z: Int => z * 2;
        esac
    };
};
```

11. **Constructor Name**

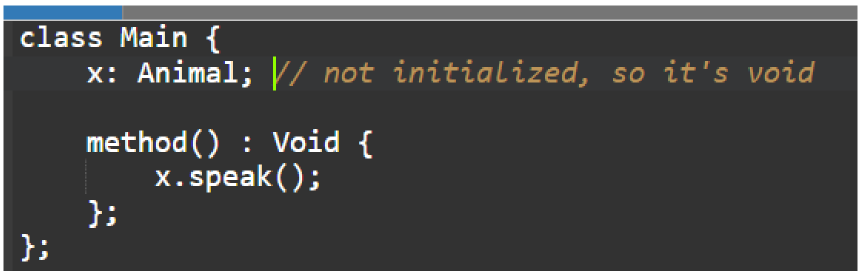
- The name of a class constructor should match the name of the class.
- Example:



```
class Animal {
    initializeDog() {
    };
};
```

12.**No Void Dispatch**

- A method should not be dispatched on a **void** object. For example, if **a** is **void**, then **a.method()** is an error. However, detecting this error during static analysis might not always be feasible, so runtime checks may also be necessary.
- Example:

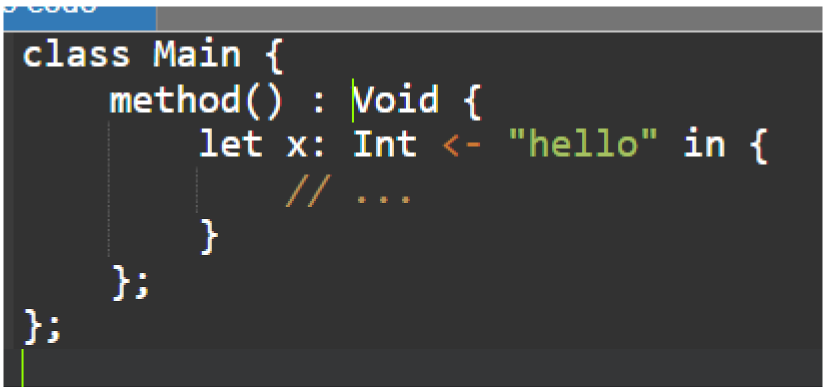


```
class Main {
    x: Animal; // not initialized, so it's void

    method() : Void {
        x.speak();
    };
};
```

13.**Let Initialization**

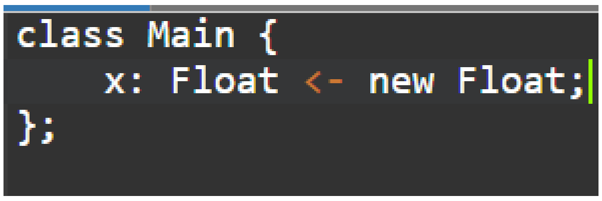
- Ensure the type of the initialization in a **let** binding matches the declared type of the identifier.
- Example:



```
class Main {
    method() : Void {
        let x: Int <- "hello" in {
            // ...
        }
    };
};
```

14.**Type of new**

- The type named in a **new** expression must be defined.
- Example:



```
class Main {
    x: Float <- new Float;
};
```