An instance of class String is a sequence of characters. The class String supports the following built-in methods:

- charAt(int index)
  This method returns the character located at the String's specified index but returns an error if the index is out of range. Indexes start from zero.
- length()
  This method returns the length of a string.

```
String w = "Liva";
String x = new String("Liva");
int y = x.length();
char z = x.charAt(1);
```

3.2.3 Arrays

Arrays can be seen as a special type. An array object contains a number of variables. All elements in an arrays must have the same type.

```
int[] ai;
char ac[] = { 'a', 'b', 'c', ' ' };
```

CHAPTER 4 Classes

A class declaration defines a new reference type and how it is implemented. Classes contain fields and methods. Field declarations describe class variables while method declarations describe programs that may be invoked by other programs

4.1 Class Declarations

Classes are defined in the following way. The optional extends clause in a class declaration specifies the superclass of the current class.

```
class MyClass extends SuperClass{
    //field, constructor
    //method declarations
}
```

LIVA

A Lite Version of Java

- Sections 1, 2, 3 are pretty good.
- Way more information needed on Classes / Objects
  - no description of how to use/create Objects
  - more examples
  - need to define OO terms more
- Consistency + undefined material

Shanqi Lu
sl4017

Jiafei Song
js4984

Zihan Jiao
zj2203

Yanan Zhang
yz3054

An instance of class String is a sequence of characters. The class String supports the following built-in methods:

- charAt(int index)
  This method returns the character located at the String's specified index but returns an error if the index is out of range. Indexes start from zero.
- length()
  This method returns the length of a string.

```
String w = "Liva";
String x = new String("Liva");
int y = x.length();
char z = x.charAt(1);
```

3.2.3 Arrays

Arrays can be seen as a special type. An array object contains a number of variables. All elements in an arrays must have the same type.

```
int[] ai;
char ac[] = { 'a', 'b', 'c', ' ' };
```

CHAPTER 4 Classes

A class declaration defines a new reference type and how it is implemented. Classes contain fields and methods. Field declarations describe class variables while method declarations describe operations that may be invoked by other programs

4.1 Class Declarations

Classes are defined in the following way. The optional extends clause in a class declaration specifies the superclass of the current class.

```
class MyClass extends SuperClass{
    //field, constructor
    //method declarations
}
```

4.2 Class Members

What is a member?

Members of a class consist of:

- Members inherited from its superclass
- Members declared in the class declaration

Give an example of each.

4.3 Field Declarations

Field Declarations specify variables of a class type.

4.4 Method Declarations

↳ This is not clear. Give an example.

Method declarations specify executable code that might be invoked.

↳ How?

```
/* Field declarations and method declarations */
class Calculation{
    int z;

    void addition(int x, int y){
        z = x+y;
    }

    void Subtraction(int x,int y){
        z = x-y;
    }
}
```

None of this syntax has been introduced yet! Either explain it more or simplify it.

4.5 Constructor Declarations

If the constructor is not defined, the compiler generates a default one.

What does a constructor do?
What does it require syntactically?
What is a "default constructor"?

```
/* User defined constructor*/
class Calculation{
    int z;
    constructor (int z){
        this.z = z;
    }
    void addition(int x, int y){
        z = x+y;
    }

    void Subtraction(int x,int y){
        z = x-y;
    }
}
```

Undefined syntax

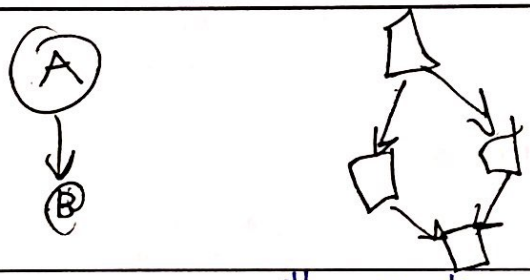
4.5 Inheritance

Explain. Implications? Multi-inheritence? Polymorphism?

Inheritance is that a subclass acquires all the behaviors and properties of a super class.

```
class My_Calculation extends Calculation {

    void multiplication(int x, int y) {
        z = x * y;
    }
}
```



CHAPTER 5 Statements

these are keywords, not statements?

Statements include: *if, else, for, break, continue, return*, as well all expressions which are explained in the following. Except as indicated, statements are executed in sequence.

Period!

5.1 Expression Statements

An expression statement consists of an expression followed by a semicolon.

```
expression;
```

Wrong Font... unless 'expression' should be understood as code?

Usually expression statements are assignments or function calls.

You have functions? Undefined!

```
/* Object creation expressions */
Student e1 = new Student();

/* Object creation expressions */
c = 8933.234;
```

→ This is not creating an object.

5.2 Declaration Statements

A declaration statement declares a variable by specifying its data type and name. It also could initialize the variable during the declaring.

```
/* declare a variable with data type and name */
char a;
int b =10;
float c;

int array1[] = { 2, 5, -2, 6, -3, 8, 0, -7, -9, 4 };
String name= "class";

boolean isMatch = false;
```

5.3 Control Flow Statements

5.3.1 If-then and If-then-else

There are two forms of conditional statements.

For the first case, the conditional expression that is evaluated is enclosed in balanced parentheses. The section of code that is conditionally executed is specified as a sequence of statements enclosed in balanced braces. If the conditional expression evaluates to false, control jumps to the end of the if-then statement.

any expression?

```
if (expression) {
    statement
}
```

only one statement...

type boolean

In the second case the second sub-statement is executed if the expression is false. As usual the 'else' ambiguity is resolved by connecting an else with the last encountered elseless if.

```
if (expression) {
      statement1
} else {
      Statement2
}
```

5.3.2 Looping: for

The 'for' condition will also run in a loop so long as the condition specified in the 'for' statement is true. The 'for' statement has the following format:

```
for (expression1; expression2; expression 3) {
      statement
 }
```

The first expression specifies initialization for the loop and it is executed once at the beginning of the 'for' statement; the second specifies a test, made before each iteration, such that the loop is terminated when the expression becomes false; the third expression ~~typically~~ specifies an ~~increment or decrease~~ which is performed after each iteration.

The following example uses a 'for' statement to print the numbers from 0 to 10:

```
for (int num=0; num < 11; num ++) {

    print(num);

}
```

5.3.3 Looping: while

The 'while' statement has the form:

```
while(expression) {
      statement
}
```

The 'while' statement will be executed in a loop as long as the specified condition in the while statement is true. The expression must have type boolean, or a compile-time error occurs.

- If the value for expression is true, then the contained statement is executed
  - If execution of the statement completes normally, then the entire 'while' statement is executed again, beginning by re-evaluating the expression.
- If the value of the expression is false, no further action is taken and the 'while' statement completes normally.

5.3.4 Branching: break, continue, and return

The break statement causes termination of the smallest enclosing for statement; control passes to the statement following the terminated statement. The expression for 'break' statement is shown below:

```
break;
```

The continue statement causes control to pass to the loop-continuation portion of the smallest enclosing for statement; that is to the end of the loop. The expression for 'continue' statement is show below:

```
continue;
```

A function returns to its caller by means of the 'return' statement, which has one of the forms:

```
return;
return(expression);
```

In the first case no value is returned when a method is declared void. For the first case, the users could specify no return statement for simplification. In the second case, simply put the value (or an expression that calculates the value) after the return Keyword, then the value of the expression is returned to the caller of the function.

5.4 Method Creation and Method Call

The user could write the user-defined methods.

```
returnType nameOfMethod (Parameter List) {
 // method body
}
```

Answered my question! But use a different font.

- `returnType`: Method may return a value. Does it have to?
- `nameOfMethod`: This is the method name. The method signature consists of the method name and the parameter list.
- `Parameter List`: The list of parameters, it is the type, order, and number of parameters of a method. These are optional, method may contain zero parameters. → Syntax? Actual list?
- `method body`: The method body defines what the method does with statements.

For using a method, it should be called. There are two ways in which a method is called i.e. method returns a value or returning nothing (no return value).
↳ this is how a method returns, not how it's called.
Following is the example to demonstrate how to define a method and how to call it with a returned value:

```
void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    print("Minimum Value = " + c);
}

int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
        min = n2;
    else
        min = n1;
    return min;
}
```

How does method call work? Pass-by-value vs. pass-by-reference?

5.5 Print to Console

The *print()* function takes one or more parameters and prints them one by one to standard output. The parameter type may be string, number, or object. It is in the following form:

```
print (parameters);
```

Here is an example to accept an *int* and print the *int* to the console.

```
print (1);
```

Another example to accept a string and print the string to the console:

```
print ("CS4115 is fun!")
```

5.6 Empty Statement

An empty statement does nothing and has the following form:

```
;
```

CHAPTER 6 Expressions

Much of the work in a program is done by evaluating *expressions*, such as assignments to variables, or for their values, which can be used as arguments or operands in larger expressions, or to affect the execution sequence in statements, or both.

This chapter specifies the meanings of expressions and the rules for their evaluation.

6.1 Evaluation, Denotation, and Result

Unnecessary

Liva evaluates a larger expression by evaluating smaller parts of it. ~~So the result of an expression is important.~~ When an expression in a program is *evaluated* (*executed*), the result denotes one of three things:

- A variable
- A value
- Nothing (for void functions and methods)

Example of each

↳Difference? You've only formally introduced methods

An expression denotes nothing if and only if it is a method invocation that invokes a method that does not return a value, that is, a method declared void. Such an expression can be used only as an expression statement (in statement chapter), because every other context in which an expression can appear requires the expression to denote something.

If an expression denotes a variable, and a value is required for use in further evaluation, then the value of that variable is used. In this context, if the expression denotes a variable or a value, we may speak simply of the *value* of the expression. In this way, we may say each expression denotes a value in a certain type.

6.2 Type of an Expression

For an expression that denotes to a variable, the value stored in a variable is always compatible with the type of the variable. In other words, the value of an expression whose type is T is always suitable for assignment to a variable of type T.

The rules for determining the type of an expression that denotes to a value are explained separately below for each kind of expression. Including arithmetic operations, relation operations, bitwise/conditional operations, assignment.

6.3 Evaluation Order

Liva guarantees that the operands of operators ~~appear to be~~ are evaluated in a specific *evaluation order*, namely, from left to right.

6.3.1 Left-Hand Operand First

The left-hand operand of a binary operator ~~appears to be~~ is fully evaluated before any part of the right-hand operand is evaluated.

6.3.2 Evaluate Operands before Operation

Liva guarantees that every operand of an operator (except the conditional operators &, |) ~~appears to be~~ are fully evaluated before any part of the operation itself is performed.
For example, in an assignment expression, the assignment will not be evaluated until the right hand operands (if it is another expression) is evaluated.

6.3.3 Evaluation Respects Parentheses and Precedence

Liva respects the order of evaluation indicated explicitly by parentheses and implicitly by operator precedence.

6.4 Lexical Literals

A literal denotes a fixed, unchanging value. This kind of expression could be evaluated without being broken into small expressions.

~~The type of a literal is determined as follows:~~

- The type of an integer literal is int.
- The type of a floating-point is float.
- The type of a boolean literal is boolean.
- The type of a character literal is char.
- The type of a string literal is String.
- The type of the null literal null is the null type; its value is the null reference.

Evaluation of a lexical literal always completes normally.

6.5 The Arithmetic Operations

The value of an equality expression is numeric (int or double, depends on the operands). The operators +, -, *, /, and % are called the arithmetic operators. They have the same precedence and are syntactically left-associative (they group left-to-right). The type of the arithmetic expression is the promoted type of its operands.

The type of each of the operands of arithmetic operators must be a type that is convertible to a primitive numeric type, or a compile-time error occurs. For example: "+" two objects of a user-defined class is prohibited.

+	Adds values on either side of the operator
-	Subtracts right hand operand from left hand operand
*	Multiplies values on either side of the operator
/	Divides left hand operand by right hand operand
%	Divides left hand operand by right hand operand and returns remainder

6.6 The Relational Operations

The value of an equality expression is always boolean. The equality operators may be used to compare two operands that are convertible to numeric int type, or two operands of type boolean. All other cases result in a compile-time error.

==	(Addition) Adds values on either side of the operator
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.

>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to (no less than) the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to (no greater than) the value of right operand, if yes then condition becomes true.

6.7 The Bitwise and Conditional Operations:

Unlike Java, Liva uses &, |, ^, ~ for both bitwise and conditional operations. That depends on the operands. The operands of a Bitwise/Conditional Operation should both be `int` or `boolean`.

&	Binary AND if both operands are `int` type / Logic AND if both operands are `boolean` type
|	Binary OR if both operands are `int` type / Logic OR if both operands are `boolean` type
^	Binary XOR if both operands are `int` type / Logic XOR if both operands are `boolean` type
~	Binary Complement if both operands are `int` type / Logic INVERT if both operands are `boolean` type

6.8 Method Invocation Expressions

Still undefined...

A method/function invocation expression is used to invoke an instance method (declared in previous chapters). The result type of the chosen method is determined as follows: If the chosen method/function is declared with a return type of void, then the result is void. Otherwise, the result type is the method/function's declared return type.

Verbose way

Redundant!

6.9 Array Access Expressions

An array access expression contains two subexpressions, the *array reference expression* (before the left bracket) and the *index expression* (within the brackets).

Note that the array reference expression may be a name or any primary expression that is not an

What is this?

Undefined

array creation expression The type of the array reference expression must be an array type. For the index expression, the promoted type must be `int`, or a compile-time error occurs.

The result of an array access expression is a variable of type T, namely the variable within the array selected by the value of the index expression.

Example! It should not just be a variable.

6.10 Assignment

In Liva, the only assignment operator is "=".

The result of the first operand of an assignment operator must be a variable. This operand may be a named variable, such as a local variable or a field of the current object or class, or it may be a computed variable, as can result from a field access or an array access (defined previously).

Example of each!

```
class Main {
    void main() {
        print("hello world");
    }
}
```