CS335A: Compiler Design
Milestone 1: Specs

Group-16

Atharv Singh Patlan (190200)
Bhagwat Garg (190229)
Rishabh Dugaye (190701)
Urbi Ghosh (190924)

January 2022

Contents

1 Introduction

Project proposal for the features we wish to support in the source language, for our compiler `JxPy`.
Source Language (Inpute File Language): `Java`
Implementation Language: `Python`
Target Language (Output/End Language): `x86`

2 Blocks and Statements

2.1 Blocks

A block is a sequence of statements and local variable declaration statements within braces.

```
Syntax:
{
\\ code
}
```

A block is executed by executing each of the local variable declaration statements and other statements in order from first to last (left to right).

2.2 Statements

2.2.1 Empty Statement

An empty statement does nothing.

```
Syntax:
    ;
```

2.2.2 Labelled Statements

Statements have label prefixes.

```
Syntax:
Identifier : Statement
```

A statement labeled by an identifier must not appear anywhere within another statement labeled by the same identifier, or a compile-time error will occur. Two statements can be labeled by the same identifier only if neither statement contains the other.

2.2.3 Expression Statements

Certain kinds of expressions may be used as statements by following them with semicolons.

```
Syntax:
StatementExpression;
```

2.2.4 if Statement

You can use the if statement to conditionally execute part of your program, based on the truth value of a given expression.

```
Syntax:
 if (test)
     then-block
 else
     else-block
```

If test evaluates to true, then then-block is executed and else-block is not. If test evaluates to false, then else-block is executed and then-block is not. The else clause is optional.
We can use a series of if statements to test for multiple conditions.

```
Syntax:
if (test1)
    then-block1
else if(test2)
    then-block2
else if(test3)
    then-block3
else
    else-block4
```

2.2.5 switch Statement

You can use the switch statement to compare one expression with others, and then execute a series of sub-statements based on the result of the comparisons.

```
Syntax:
switch (test)
{
case compare-1:
if-equal-block-1
case compare-2:
if-equal-block-2
....
default:
default-block
}
```

The switch statement compares test to each of the compare expressions, until it finds one that is equal to test. Then, the blocks following the successful case are executed. Optionally, you can specify a default case. If test doesn't match any of the specific cases listed prior to the default case, then the block for the default case are executed.

2.2.6 while Statement

The while statement is a loop statement with an exit test at the beginning of the loop.

```
Syntax:
 while (test)
     code-block
```

The while statement first evaluates test. If test evaluates to true, block is executed, and then test is evaluated again. block continues to execute repeatedly as long as test is true after each execution of statement.

2.2.7 do Statement

The do statement is a loop statement with an exit test at the end of the loop.

```
Syntax:
do
  code-block
while (test);
```

The do statement first executes code-block. After that, it evaluates test. If test is true, then code-block is executed again. code-block continues to execute repeatedly as long as test is true after each execution of code-block.

2.2.8 for Statement

The for statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops.

```
Syntax:
  for (initialize; test; step)
      code-block
```

The for statement first evaluates the expression initialize. Then it evaluates the expression test. If test is false, then the loop ends and program control resumes after code-block. Otherwise, if test is true, then code-block is executed. Finally, step is evaluated, and the next iteration of the loop begins with evaluating test again.

2.2.9 break Statement

You can use the break statement to terminate a while, do, for, or switch statement.

```
Example Syntax:
 while(test){
     if(test1)
     then-block
     else
     break;
 }
```

If in any iteration test1 turns out to be false then the while loop is terminated.

2.2.10 continue Statement

You can use the continue statement in loops to terminate an iteration of the loop and begin the next iteration.

```
Example Syntax:
while(test){
if(test1)
continue;
else
else-block;
}
```

If in any iteration test1 turns out to be true then the current iteration is terminated and loop goes to next iteration.

2.2.11 return Statement

A return statement returns control to the invoker of a method.

```
Syntax:
return return-value
```

return-value is an optional expression to return. If the function's return type is void, then it is invalid to return an expression. You can, however, use the return statement without a return value.

3 Methods

A method declares executable code that can be invoked, passing a fixed number of values as arguments.

3.1 Method Declaration

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.
More generally, method declarations have five components, in order:

1. Modifiers—such as public, private, and others.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

```
Example:
public double calculateAnswer(double wingSpan, int numberOfEngines,double length, double grossTons)
{
    \\do the calculation here\\
}
```

The signature of a method consists of the name of the method and the number and types of formal parameters to the method. A class may not declare two methods with the same signature, or a compile-time error occurs.

3.2 calling methods

You can call a method by using its name and supplying any needed parameters.

```
Syntax:
method-name (parameters);
```

3.3 recursive methods

You can write a function that is recursive—a function that calls itself.

```
Example:
int factorial (int x)
{
if (x < 1)
    return 1;
else
    return (x * factorial (x - 1));
}
```

3.4 method overloading

If two methods of a class have the same name but different signatures, then the method name is said to be overloaded. This fact causes no difficulty and never of itself results in a compile-time error. There is no required relationship between the return types or between the throws clauses of two methods with the same name but different signatures.

4 Lexical Elements

Java programs are written in Unicode, which after preprocessing form the set of lexical elements supported by Java.

These lexical elements are namely: tokens, white space, and comments.

4.1 Tokens

Tokens are further divided into identifiers, keywords, literals, separators and operators.

4.1.1 Identifiers

An identifier is an unlimited-length sequence of ASCII letters, ASCII digits, underscores (_) and dollar signs ($), the first of which cannot be an ASCII digit. An identifier cannot have the same spelling as a keyword, a boolean literal or a null literal. Identifiers are case-sensitive.

4.1.2 Keywords

These are the pre-defined reserved words in Java. Each of these, in a lowercase form, has a special meaning and can not be used as identifiers

01. boolean	02. byte	03. break	04. class	05. case
06. catch	07. char	08. continue	09. default	10. do
11. double	12. else	13. final	14. finally	15. float
16. for	17. if	18. implements	19. import	20. instanceof
21. int	22. interface	23. long	24. native	25. new
26. package	27. private	28. protected	29. public	30. return
31. short	32. static	33. super	34. switch	35. synchronized
36. this	37. throw	38. throws	39. transient	40. try
41. void	42. volatile	43. while	44. enum	45. strictfp

4.1.3 Literals

4.1.3.1 Integer Literal

Can be:

- A decimal numeral is either the single ASCII character 0, representing the integer zero, or consists of an ASCII digit from 1 to 9, optionally followed by one or more ASCII digits from 0 to 9, representing a positive integer
- A hexadecimal numeral consists of prefix `0x` or `0X` and follows the same rules with additional characters : `a-f` and `A-F` representing decimals numerals 10-15
- An octal numeral consists of an ASCII digit 0 followed by one or more of the ASCII digits 0 through 7 and can represent a positive, zero, or negative integer

Integer literals can be suffixed with `l` or `L` to get long integer literals

Declaration

```
int     // range -2^31 to 2^31 - 1 (inc.)
long    // range -2^63 to 2^63 - 1 (inc.)
```

4.1.3.2 Floating Point Literal

A floating-point literal has the following parts:

- a whole-number part
- a decimal point (represented by an ASCII period character)

- a fractional part
- an exponent
- a type suffix

At least one digit, in either the whole number or the fraction part, and either a decimal point, an exponent, or a float type suffix are required. All other parts are optional. The exponent, if present, is indicated letter `e` or `E` followed by an optionally signed integer.

The Java types float and double are 32-bit and 64-bit floating point values respectively with their ranges defined by IEEE 754 standard. Float literals can be suffixed with `f` or `F` and double literals can be suffixed with `d` or `D`

Declaration

```
float
double
```

4.1.3.3 Boolean Literal

The boolean type has two values, represented by the literals `true` and `false`

Declaration

```
boolean
```

4.1.3.4 Character Literal

A character literal is expressed as a character or an escape sequence, enclosed in ASCII single quotes.

Declaration

```
char
```

4.1.3.5 String Literal

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence
Supported escape sequence:

```
\b
\t
\n
\f
\r
\"
\'
\\
\u0000 to \u00ff: from octal value
```

Declaration

```
String
```

4.1.3.6 Null Literal

The null type has one value, the null reference

Declaration

```
null
```

4.1.4 Separators

The following nine ASCII characters are the Java separators (punctuators):
( ) { } [ ] ; , .

4.2 Operators

The following ASCII characters form the Java operators which follow a set precedence order (decreases from up to down):

postfix	expr++	expr–										
unary	++expr	–expr	+expr	-expr	~	!						
multiplicative	*	/	%									
additive	+	-										
shift	<<	>>	>>>									
relational	<	>	<=	>=								
equality	==	! =										
bitwise AND	&											
bitwise XOR	^											
bitwise OR	|											
logical AND	&&											
logical OR	||											
ternary	? :											
assignment	=	+=	-=	*=	/=	%=	&=	^=	|=	<<=	>>=	>>>=

4.3 White Spaces

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators

4.4 Comments

Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:

- Single line comments: It begins with a pair of forward slashes (//).
- Block comments: It begins with /* and continues until a */ occurs.

5 Types, Values, and Variables

5.1 Types and Values

There are two kinds of types in Java:

5.1.1 Primitive

The types `int, long, char, float, double` are primitive. Primitive values do not share state with other primitive values. A variable whose type is a primitive type always holds a primitive value of that same type.

5.1.2 Reference

There are two kinds of reference types: class types, and array types. The reference values are pointers to these objects, and a special null reference, which refers to no object.

5.2 Variables

A variable is a storage location and has an associated type, sometimes called its compile-time type, that is either a primitive type or a reference type

5.2.1 Variables of Primitive Type

A variable of a primitive type always holds a value of that exact primitive type.

5.2.2 Variables of Reference Type

A variable of reference type can hold either a null reference or a reference to any object

5.3 Special Classes

5.3.1 The Class `Object`

The standard class Object is a superclass of all other classes. A variable of type Object can hold a reference to any object, whether it is an instance of a class or an array

5.3.2 The Class `String`

Instances of class String represent sequences of Unicode characters. A String object has a constant (unchanging) value. String literals are references to instances of class String.

6 Expressions

When an expression in a Java program is evaluated (executed), the result denotes one of three things:

- A variable
- A value
- Nothing

6.1 Evaluation Order

Java guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right

6.1.1 Evaluate Left-Hand Operand First

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

6.1.2 Evaluate Operands before Operation

Java also guarantees that every operand of an operator (except the conditional operators &&, ||, and ? :) appears to be fully evaluated before any part of the operation itself is performed.

6.1.3 Evaluation Respects Parentheses and Precedence

Java implementations must respect the order of evaluation as indicated explicitly by parentheses and implicitly by operator precedence.

6.1.4 Argument Lists are Evaluated Left-to-Right

In a method or constructor invocation or class instance creation expression, argument expressions may appear within the parentheses, separated by commas. Each argument expression appears to be fully evaluated before any part of any argument expression to its right.

6.2 `this` keyword

The keyword `this` may be used only in the body of an instance method or constructor, or in the initializer of an instance variable of a class. When used as a primary expression, the keyword this denotes a value, that is a reference to the object for which the instance method was invoked, or to the object being constructed.

6.3 Class Instance Creation Expressions

A class instance creation expression is used to create new objects that are instances of classes.
Syntax

```
new ClassType ( ArgumentList)
```

Here, `ClassType` is the type of the creation expression. The arguments in the argument list, if any, are used to select a constructor declared in the body of the named class type

6.4 Field Access Expressions

A field access expression may access a field of an object or array, a reference to which is the value of an expression
Syntax

```
Primary . Identifier
```

The type of the Primary must be a reference type `T`, or a compile-time error occurs.

- If the identifier names several accessible member fields of type T, then the field access is ambiguous and a compile-time error occurs
- If the identifier does not name an accessible member field of type T, then the field access is undefined and a compile-time error occurs.
- Otherwise, if the field is static, then the result is the specified class variable
- If the field is not static, then the result is the specified instance variable

6.5 Method Invocation Expressions

A method invocation expression is used to invoke a class or instance method.
Syntax

```
MethodName ( ArgumentList )
Primary . Identifier ( ArgumentList )
```

Resolving a method name at compile time is complicated because of the possibility of method overloading.

Compile-time processing of a method invocation involves determining method signature using the name of the method and the types of the argument expressions to locate method declarations that are both applicable and accessible, that is, declarations that can be correctly invoked on the given arguments.

6.6 Postfix Expressions

Postfix expressions include uses of the postfix `++` and `--` operators. The result of the postfix expression must be a variable of a numeric type, or a compile-time error occurs

6.7 Unary Operators

The unary operators include `+, -, ++, --,` $\sim$, `!`, and cast operators. Expressions with unary operators group right-to-left, so that $- \sim x$ means the same as $-(\sim x)$.

6.8 Cast Expressions

A cast expression converts, at run time, a value of one numeric type to a similar value of another numeric type; or confirms, at compile time, that the type of an expression is `boolean`
Syntax

```
( PrimitiveType Dims) UnaryExpression
```

6.9 Multiplicative Operators

The operators `*, /, and %` are called the multiplicative operators. They have the same precedence and are syntactically left-associative (they group left-to-right).

6.10 Additive Operators

The operators + and - are called the additive operators. They have the same precedence and are syntactically left-associative (they group left-to-right). If the type of either operand of a + operator is String, then the operation is string concatenation. Otherwise, the type of each of the operands of the + operator must be a primitive numeric type, or a compile-time error occurs.

6.10.1 String Concatenation Operator +

Concatenates two strings. If only one operand expression is of type String, then string conversion is performed on the other operand to produce a string at run time. The result is a reference to a newly created String object that is the concatenation of the two operand strings

6.11 Shift Operators

The shift operators include left shift <<, signed right shift >>, and unsigned right shift >>>; they are syntactically left-associative (they group left-to-right). The left- hand operand of a shift operator is the value to be shifted; the right-hand operand specifies the shift distance.

6.12 Relational Operators

The relational operators include `<, <=, >, and >=`. The relational operators are syntactically left-associative (they group left-to- right), but this fact is not useful; for example, `a<b<c` parses as `(a<b)<c`, which is always a compile-time error, because the type of `a<b` is always boolean and `<` is not an operator on boolean values. The type of each of the operands of a numerical comparison operator must be a primitive numeric type, or a compile-time error occurs.

6.13 Equality Operators

The equality operators include `==, !=`. The equality operators are syntactically left-associative. The equality operators may be used to compare two operands of numeric type, or two operands of type boolean, or two operands that are each of either reference type or the null type. The type of an equality expression is always boolean.

6.14 Bitwise and Logical Operators

The bitwise operators and logical operators include the AND operator `&`, exclusive OR operator `^`, and inclusive OR operator `|`. Each of these operators is syntactically left-associative (each groups left-to-right).

6.15 Conditional-And Operator `&&`

The `&&` operator evaluates its right-hand operand only if the value of its left-hand operand is true. It is syntactically left-associative (it groups left-to-right). Each operand of `&&` must be of type boolean, or a compile-time error occurs. The type of a conditional-and expression is always boolean.

6.16 Conditional-Or Operator `||`

The `||` operator evaluates its right-hand operand only if the value of its left-hand operand is false. It is syntactically left-associative (it groups left-to-right). Each operand of `||` must be of type boolean, or a compile-time error occurs.

6.17 Conditional Operator ? :

The conditional operator `? :` uses the boolean value of one expression to decide which of two other expressions should be evaluated. The conditional operator is syntactically right-associative (it groups right-to-left), so that `a?b:c?d:e?f:g` means the same as `a?b:(c?d:(e?f:g))`.

6.18 Assignment Operators

There are 12 assignment operators ie: `= *= /= %= += -= <<= >>= >>>= &= ^ = |=`. all are syntactically right-associative. The result of the assignment expression is the value of the variable after the assignment has occurred.

6.19 Constant Expression

A compile-time constant expression is an expression denoting a value of primitive type or a String that is composed using only the following:

- Literals/casts of primitive type and literals of type String
- Operators

7 Arrays

An array in Java is a dynamically created object containing some (or zero, in which case the array is called *empty*) number of variables. All the variables must have same type, also called the component type of the array.
These variables (called *components*) are referenced by non-negative integer values called array access expressions. The length of an array (say t) is the number of components in it. The components are numbered from *0* to *t-1*.
All methods of class Object can be used on an array.

Arrays can be nested, *i.e.* the component type of an array can be an array type itself. Thus the sub arrays are referenced within the outer array. The component type of the innermost array is called the *element type* of the original array.
In the case when element type is `Object`, some or all elements of the array can be arrays.

7.1 Array Types

An array type is written as an element type name followed by i pairs of square brackets [ ] denoting the depth of nesting. The element type could be referenced or primitive. Array types are used in declarations and cast expressions.

7.2 Array Variables

A variable of array type holds a reference to an object. Declaring a variable of array type does not create an array object or allocate any space for array components. The initializer part of a declarator may create an array, a reference to which then becomes the initial value of the variable. A single variable of array type can contain references to arrays to different lengths as array length is immaterial for defining array type.

may appear as part of the type at the beginning of the declaration, or as part of the declarator for a particular variable, or both.
Once an array object is created its length never changes.

7.3 Array Creation

Array Creation Expression: This specifies the element type, the number of levels of nested arrays, and the length of the array for at least one of the levels of nesting. The array's length is available as a final instance variable `length`.
Syntax:

7.3.1 Run Time Evaluation

The run time evaluation of an array creation expression follows the order:

- Dimension expressions are evaluated from left to right. An expression to the right isn't evaluated if an expression to the left stops abruptly.
- The values in dimension expressions are checked.
  Error code: `NegativeArraySizeException` is shown for negative values.
- New array is allocated new space. Failing to to do because of insufficient space leads to `OutOfMemoryError` and abrupt completion of array creation expression.
- For single dimension arrays, the array of desired length is created and each component is initialised to default value.
- For d dimensional arrays, a for loop of depth *d-1* is created to create the array of arrays.

7.3.2 Evaluation Order

: Example:
When there are more than one dimensions expressions in an array creation expression, each bracket bound dimension expression is evaluated fully before moving to expressions to its right. If the evaluation of a dimension expression stops abruptly then no expression to the right of it will be evaluated.

7.3.3 Out-of-Memory Detection

: Insufficient memory to perform operations during the evaluation of array creation expression, gives rise to and `OutOfMemoryError`. This is checked after evaluation of all dimension expressions has completed normally.

An array can also be created using **Array Initialiser** (Section 6.6) that creates an array and provides initial values for all its components.

7.4 Array Access

Array access expressions are expressions with array refernce as value followed by an indexing expression enclosed in [and] eg: A[i]. Indices can range from `1` to `n-1` for an array of length `n`. Indices must be `int` but `short, byte,` or `char` values may also be used. `long` will give Compile-time error.

```
ArrayAccess:

    ExpressionName [ Expression ]

    PrimaryNoNewArray [ Expression ]
```

`IndexOutOfBoundsException`: when index is negative or greater than (length of array)-1. Array access is checked at runtime.

7.5 Array Initialisers

Array Initialiser is a declaration for creating new array and initialising its components. It is written as a list of expressions, comma separated and enclosed in braces. The number of expression denotes the size of the array.
Array initialisers can be nested. If any of these expressions isn't compatible to the component type of the array, there will be compile-time error. A trailing comma may appear after the last expression in an array initializer and is ignored.
Syntax:

```
ArrayInitializer:
    { VariableInitializers_(opt , opt)}


VariableInitializers:
    VariableInitializer
    VariableInitializers , VariableInitializer
```

and

```
VariableInitializer:
        Expression
        ArrayInitializer
```

7.6 Array Members

Array members can be of types:

- `public final` field `length`: contains the number of components of the array
- `public` method `clone`: overrides method of the same name in class `Object`, throws no checked exceptions
- members are inherited from class `Object`; the only method of `Object` not inherited is its `clone` method

array thus has the same methods as the following class:

```
class A implements Cloneable {
    public final int length = X;
    public Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
        throw new InternalError(e.getMessage());
        }
    }
}
```

7.7 Class Object for Array

Every array a class `Object` associated with it. `Object` is the superclass of an array type.

```
class Test {
    public static void main(String[] args) {
        int[] ia = new int[3];
        System.out.println(ia.getClass());
        System.out.println(ia.getClass().getSuperclass());
    }
}
```

prints `class [I`
`class java.lang.Object` and string "[I" is the run-time type signature for the class object "array with component type int"

8 Packages

Java programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A type is accessible outside the package that declares it only if the type is declared public.

The naming structure for packages is hierarchical. The members of a package are class types, which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package consists of a number of compilation units. A compilation unit automatically has access to all types declared in its package

A compilation unit has three parts, each of which is optional:

- A package declaration, giving the fully qualified name of the package to which the compilation unit belongs
- import declarations that allow types from other packages to be referred to using their simple names
- Type declarations of class types

A package can have members of either or both of the following kinds:

- Subpackages of the package
- Types declared in the compilation units of the package

If the fully qualified name of a package is `P`, and `Q` is a subpackage of `P`, then `P.Q` is the fully qualified name of the subpackage.

Java would transform a package name into a pathname by concatenating the components of the package name, placing a file name separator (directory indicator) between adjacent components. Eg:

```
jag.scrabble.board
```

would be transformed into the directory name:

```
jag/scrabble/board
```

8.1 Compilation Units

A compilation unit consists of three parts

- A package declaration, giving the fully qualified name of the package to which the compilation unit belongs
- import declarations (optional) that allow types from other packages to be referred to using their simple names
- Type declarations of class

8.1.1 Package Declarations

A package declaration appears within a compilation unit to indicate the package to which the compilation unit belongs.

Syntax

```
package PackageName;
```

8.1.2 Import Declarations

An import declaration allows a type declared in another package to be referred to by a simple name that consists of a single identifier. Without the use of an appropriate import declaration, the only way to refer to a type declared in another package is to use its fully qualified name.
Syntax

```
import TypeName ;
```

Eg: `import java.util.Vector;` causes the simple name `Vector` to be available within the class declarations in a compilation unit.

8.1.3 Type Declarations

A type declaration declares a class type

9 Classes

Class declarations define new reference types and describe how they are implemented. It can be made of other variables (including other classes and methods).

9.1 Defining a class

A class is defined using the class keyword, followed by the name of the class. A block follows which specifies all the members of the class.

A class can contain variables corresponding to primitive data types, references, other classes as well as methods as its members. These variables are declared within the block of the class.

```
class Point {
    int x, y;
}
```

This defines a class `Point` which contains two integer members `x, y`.
However if a class is defined within a package, like so:

```
package geo;
class Point {
    int x, y;
}
```

The fully qualified name of the class is `geo.Point`

The scope of a class name is the entire package in which the class is declared. Thus, another class with the same name can not be declared.

9.2 Class modifier

A class declaration may include class modifier : `public` before the `class` keyword. This allows the members of the class to be accesses by any Java code that can be access the package in which it is declared.

9.3 Declaring class variables

A class variable can be declared by using the name of the class as any ordinary datatype, to preceed the variable name, and then assigning the value of a new object of the class to the variable, like `Point a = new Point();`

This calls the default constructor. However, if values are supplied and a constructor member is present in the class, then the values can be passed along in the initialization.

As an example, `Point a = new Point(5, 2);` assigns the value 5 to `a.x` and 2 to `a.y`.

9.4 Accessing class members

Any public class member can be accessed directly specifying the name of the member after the name of the class variable, with a '.' in between.

For the above example, `a.x = 10;` can be used to set the value of the variable x in a to 10.

Accessing private methods using such a mechanism is not possible.

Inside of a class declaration, another member is accessed using the keyword `this`. (Example in the next section)

9.5 Constructors

A constructor is used in the creation of an object that is an instance of a class. It is declared much like a method, with the name of the constructor being same as the name of the class, specifying the arguments of the constructor, although return type is not provided in the constructor and there is no return statement.

```
class Point {
    int x, y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Like any ordinary method, constructors can be overloaded as well, and each class has a default constructor which takes no arguments, and does not need to be coded.

10 Input/Output statements

10.1 Input from standard input device

The `Scanner` class is used to get user input, and it is found in the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class.

`Example:`

```
import java.util.Scanner;  // Import the Scanner

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

Other methods in `Scanner` class are:

- `nextInt()` to take int as input.
- `nextDouble()` to take double as input.
- `nextFloat()` to take float as input.
- `nextLine()` to take string as input.
- `nextByte()` to take byte as input.

10.2 Output to console

The output can be print on console using `System.out.println()`.

```
Example:
System.out.println("Welcome");
System.out.println("To");
System.out.println("Java");
```

10.3 File I/O

The `File` class from the `java.io` package, allows us to work with files.
To use the `File` class, create an object of the class, and specify the filename or directory name:

To create a file in Java, you can use the `createNewFile()` method. This method returns a boolean value: true if the file was successfully created, and false if the file already exists.

```
Syntax:
File myObj = new File("filename.txt");
myObj.createNewFile();
```

We use the `FileWriter` class together with its write() method to write some text to the file we created in the example above. Note that when you are done writing to the file, you should close it with the close() method:

```
Example:
FileWriter myWriter = new FileWriter("filename.txt");
myWriter.write("Files in Java might be tricky, but it is fun enough!");
myWriter.close();
```

We use the `Scanner` class to read the contents of the text file we created.

```
Example:
File myObj = new File("filename.txt");
Scanner myReader = new Scanner(myObj);
while (myReader.hasNextLine()) {
    String data = myReader.nextLine();
    System.out.println(data);
}
myReader.close();
```

11 String Functions

The class `String` has several methods pre-defined which provide various different functionalities.

- `public boolean equals(String b)`

  Returns `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the `String` object.

- `public int length()`

  The length of the sequence of characters represented by this `String` object is returned.

- `public char charAt(int idx)`

  This method returns the character indicated by the `idx` argument within the sequence of characters represented by this String. The input string is assumed to be zero-indexed.

- `public Boolean equalsIgnoreCase(String b)`

  Returns `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the `String` object, ignoring their case.

- `public int indexOf(int ch)`

  If a character with value `ch` occurs in the character sequence represented by this `String` object, then the index of the first such occurrence is returned-that is, the smallest value k such that:

  `this.charAt(k) == ch`

  is true. If no such character occurs in this string, then -1 is returned.

- `public String substring(int startIndex, int endIndex)`

  Returns a contiguous subsequence of characters from the string from the `startIndex` to the `endIndex`.

- `public String concat(String str)`

  Returns a new `String` object formed by appending `str` to the original string.

- `public String replace(char oldChar, char newChar)`

  Returns a `String` object in which every occurence of `oldChar` is replaced by newChar.

- `public String toLowerCase()`

  Returns a `String` object in which all alphabet are converted to lower case.

- `public String toUpperCase()`

  Returns a `String` object in which all alphabet are converted to upper case.

12 Math Functions

- `public double/float/int/static abs(num)`
  Returns the absolute value of `num`.
- `public int ceil(double num)`
  Returns the value of `num` rounded up to the nearest `int`.
- `public int floor(double num)`
  Returns the value of `num` rounded down to the nearest `int`.
- `public double/float/int/static max(num1, num2)`
  Returns the larger value of `num1` and `num2`.
- `public double/float/int/static min(num1, num2)`
  Returns the smaller value of `num1` and `num2`.
- `public int round(double num)`
  Returns the value of `double num` rounded off to nearest `int`.
- `public double signum(double num)`
  Returns the sign of `double num`.
- `public double copySign(float num1,float num2)`
  Returns first floating point `num1` with the sign of floating point `num2`.

13 References

- http://titanium.cs.berkeley.edu/doc/java-langspec-1.0
- https://www.w3schools.com/java/default.asp
- https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html
- https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html