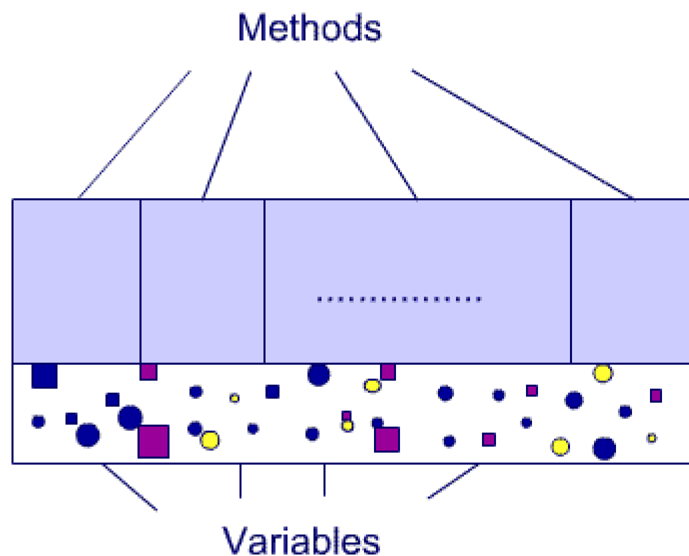
Objects and Classes

Object oriented programming is a programming approach in which all computations are done in the context of objects. A program written in an object oriented programming language can be seen as a collection of objects collaborating to perform a given task. An object is an entity that encapsulates data describing its current state and the methods through which it provides services to the outside world and modify its current state. A set of similar objects are said to be instances of a single class. You can think of a class as a blueprint for the objects of the same kind.
A class can be declared in the Java programming language using the keyword `class`. States of the instances are declared as member variables. A member variable can be simply declared by specifying a type and a name. It is also possible to initialize member variables while declaring. Methods are declared as functions with zero or more parameters and a return value. The return value of a method may be of type `void` meaning nothing is returned. It is also possible to declare more than one method with the same name as long as they require different parameters.
It is possible to restrict access to member variables and methods of a class by using `public`, `protected`, and `private` keywords. Only public member variables and public methods are accessible outside the class declaration. These access restriction mechanisms will be explained in detail in later chapters.
In the following, a class declaration template is given.

```
class ClassName{
        public variableType1 variableName1;
        public variableType2 variableName2=initialValue;
        public returnType MethodName1(parameterType1
parameterName1,
                                        ..., parameterTypeN
parameterNameN){
                ...
                return ReturnValue;
        }
        public void MethodName2(parameterType1
parameterName,
                                        ..., parameterTypeN
parameterNameN){
                ...
        }
}
```

Following figure shows a graphical representation of relationships between objects, variables and methods.



Creating and Using Objects

Any object must be explicitly created before it is used. To create an object, a special method of the corresponding class called `constructor` must be called. The constructor allocates necessary resources for an object and return an instance of the created object. In the Java programming language, you can declare the constructor by creating a method whose name is the same with the class. More than one constructors may be defied as long as they require different parameters.

```
class ClassName{
        public ClassName(zero or more parameters){
                ...
        }
}
```

In order to create an instance of a class the `new` operator is used. The instances can be kept in variables of reference type which are declared and can be initialized as:

```
ClassName VariableName = new ClassName(<parameters>);
```

Where *<parameters>* are the parameters required by the constructor if there are any. Member variables and methods of an instance can be accessed directly using their names within the class declaration or through its reference variable outside the class declaration as:

```
VariableName.MemberVariableName
VariableName.MethodName(<parameters>)
```

Class Variables and Class Methods

Member variables may keep different values in different instances of a class as every instance may have a different state. However, it is possible to declare a special kind of member variable called class variable, whose value is the same across all instances of that class, using the `static` keyword. For example:

```
class ClassName{
        public VariableType VariableName = InitialValue;
        ...
}
```

A class variable can be accessed through instances like any other member variable, or it can be accessed using the class name as `ClassName.VariableName` . Similarly a class method may also be declared using the `static` keyword as:

```
class ClassName{
        public static ReturnType MethodName(parameters){
                ...
        }
        ...
}
```

Like class variables, class methods can be called through a reference variable or directly using the class name without requiring an instance. That is, class methods may be used without creating an instance. Since they can be called without an instance, they can not modify member variables except the class variables.

Java Program

Since the Java Programming Language is a purely object oriented language, everything within a program must be contained in the class definitions. In a program, multiple classes may cooperate by calling methods of each other to perform the requested task.
You can create a standalone application by creating a program source file which contains at least one class declaration. Every program needs an entry point for the first statement to be executed. Therefore, you must specify the entry point for a program as a method of a class. A special class method `main` is used for this purpose. Since class methods do not need any instances to be called, specifying the class which has a main method to execute is enough to execute the program. The format of the main method is as follows:

```
public static void main(String[] args){
...
}
```

The `main` method takes single parameter `args` and does not return any value. The `args` parameter is of type array of `String` and it contains the parameters entered from the command line when the program is run. Since the `main` method must be a class method

which can be called without an instance outside the class, `public` and `static` keywords are used.

Comments

The Java programming language supports three types of comment: single line comments with `//`, multi-line comments with `/*` and `*/`, and `javadoc` comments with `/**` and `*/`. The first two types are simple comments, and third one is used for embedded documentation of a program that is processed by the `javadoc` tool. Single line comments can be put anywhere in the program by using `//` pattern. The comment starts at the first `//` and continues to the end of that line. The following are the examples for a single line comment.

```
int i=7; // a single line comment
i=i*7;
// another single line comment
i=1;
```

Multi line comments can be used when a comment spans multiple lines in a program. The compiler will discard anything between the starting tag `/*` and the first ending tag `*/`. The following are the examples for a multi-line comment:

```
int i=7;
/* a
comment */
i=i*7; /*
another
comment
*/
i=i+1;
```

The `javadoc` comments may span multiple lines and the compiler will discard anything between the starting tag `/**` and the first ending tag `*/`. The `javadoc` utility can read `javadoc` comments and turn them into HTML documentation.
You can add class and interface-level `javadoc` comments as well as method, constructor and field-level comments. Each comment appears just before the corresponding entity and consists of a description followed by one or more tags. If desired, you can use HTML formatting in your `javadoc` comments. The general format of a `javadoc` comment is as follows:

```
/**
  * A class description
  *
  * @tagname descriptive text
  * ...
  * @tagname descriptive text
  */
public class DocTest {
  /** A variable description */
  public int i;
```

```
/**
  * A method description
  *
  * @tagname descriptive text
  * ...
  * @tagname descriptive text
  */
  public void f() {}
}
```

There are many different kinds of tag defined for `javadoc` comments such as `@param parameter_name parameter_description`, `@author author_name`, `@version version_information` For complete listing of tags refer to the Java documentation.

Variables

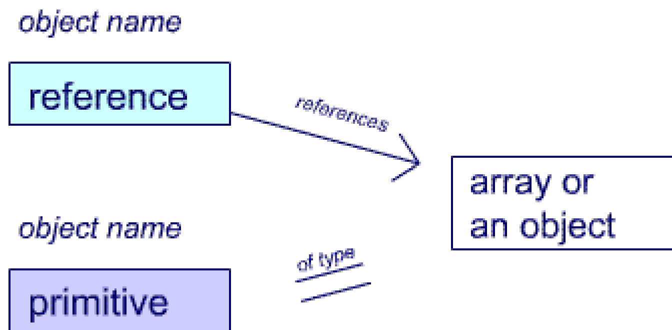
A variable is as a named container for data (and objects). Memory space is allocated for a variable at runtime and its name is used to get or set its value. That is, you can use the name of the variable to read or write the memory location.

You must declare a variable before using it. That is, you must declare the variable by giving a name and a type to it. In Java, all variables must be typed, therefore, compiler knows how to interpret the data it contains. A variable can be declared as `TypeName VariableName`.

A variable can be of a primitive type or reference type. A variable of a primitive type can contain a single whole number (`byte`, `short`, `int`, `long`), a decimal number (`double`, `float`), a single or unicode character (`char`), or a single on/off state (`boolean` which contains either `true` or `false`). A variable of a reference type points to an instance of a `class` or an `array`.

A variable's name must be a valid `identifier` That is, must be all one word (no spaces or hypens inside) starting with a letter. It can actually begin with a unicode currency symbol or an underscore (_), but it is best to start it with a letter. It may contain letters, numbers, underscores, unicode currency symbols (such as `$`), but no other special characters can be used. Since java is case-sensitive, its name must be used exactly in the same form after it is declared.

Following figure shows differnec of reference and primitive type variables:



Variable Initialization

Method parameters and exception handler parameters are initialized by the caller of the method. A member or local variable may be initialized in its declaration using the assignment operator `=` as:

```
typename identifier = initial_value;
```

Or its value can be set after its declaration as:

```
typename identifier;
...
identifier = initial_value;
...
```

A member variable is guaranteed to get a default value (zero for numeric variables, `false` for `boolean` variables, and unicode character zero for characters and `null` for reference types) even you don't initialize it. However, you must explicitly initialize local variables before using them.
You can also declare a variable as a `final` variable whose value can not be changed after it is initialized. A `final` variable can be declared in any context as:

```
final float PI = 3.14; //initialization at declaration
...
final char firstLetter; // blank final
...
firstLetter = 'A' // deferred initialization
```

`final` variables are actually correspond to constants in other languages. Any attempt to change a `final` variable's value after it is initialized will result in a compile time error. For example, the value of variable `PI` can not be changed after it is declared and the value of `firstLetter` can not be changed after it is initialized.

Scope

A variable is accessible within its scope. A variables scope depends on the context of its declaration. A variable can be declared in mainly four different contexts: within a class as a member variable (outside any methods), within method declaration as a

method parameter, within a method as a local parameter, and within an exception handler as an exception handler parameter (exception handlers will be explained in later chapters).
A member variable's scope covers the entire declaration of a class. A method parameter's scope covers the entire a method. A local parameter's scope extends from its declaration to the end of the enclosing code block (A code block, which will be defined later in this chapter, includes everything between the left, `{`, and right, `}`, curly braces including the expression that introduces the curly brace part).

Primitive Data Types

A variable of a primitive type can contain a single value of predefined size and format. The format and size of primitive data types do not change with the system on which the program is running. This contributes to the portability of programs written in the Java programming language. There are three kinds of primitive data type: numeric (integer or decimal), boolean, and character. The table on the right lists the primitive data types supported by the Java programming language. Note that all integer data types are signed and size of the boolean data type is not specified (it can only contain true or false). You should choose an appropriate integer data type for your variable according to the range of values it may contain. Otherwise, results of your computations may not be correct (there may be overflows). Similarly, if you don't choose a suitable decimal number type, you may loose precision or computations may give an `infinite` result.

Primitive Data Types

Category	Type Name	Size	Format / Range of Values
Integers	`byte`	1 byte integer	2's complement `-128` to `127`
	`short`	2 byte integer	2's complement -2^{15} to $2^{15}-1$
	`int`	4 byte integer	2's complement -2^{31} to $2^{31}-1$
	`long`	8 byte integer	2's complement -2^{63} to $2^{63}-1$
Decimal Numbers	`float`	4 byte floating point number	IEEE 754
	`double`	8 byte real number	IEEE 754
Characters	`char`	2 byte Unicode character	Unicode `0` to Unicode $2^{16}-1$
Booleans	`boolean`	-	`true` or `false`

The value of a primitive data type can be converted to an other primitive data type implicitly by the compiler or you can convert its type explicitly by typecasting. Widening type of conversions (e.g., `byte` to `int`, `int` to `float`, etc.) can be implicitly done by the compiler. However, narrowing type of conversions (e.g., `double` to `int`, `int` to `short`, etc.) requires explicit casting. Explicit typecasting has the following form:

```
(type) value
```

Here, `type` corresponds to `char` or any numeric data type, and `value` can be anything (a directly written value - literal, a variable, or an expression returning a value) that has a value of primitive data type (except `boolean`).

Literals

A literal is the actual representation of a number, a character, a boolean, or a string that can be used directly in a program (to initialize a variable, while making computations, comparisons, etc.).
The table on the right summarizes the format and examples for each kind of literal.
Note that you can place a `-` sign in front of numerical literals to have negative values, and exponent part of decimal numbers can be made negative by inserting a `-` sign after `e` (or `E`). The character literals are enclosed in single quotes, `' '`, and string literals are enclosed in double quotes, `" "`.

Literals

Type	Format	Examples
`int`	A sequence of digits `0-9` (`0-9,A,B,C,D,E,F` characters for hexadecimal and `0-7` characters for octal)	`123` `0123` (octal-base 8) `0x123` (hexadecimal - base 16)
`long`	A sequence of digits followed by a `l` or `L` letter.	`123L`

`double`	A sequence of digits with one decimal point symbol `.` and/or `e` (or `E`) letter. The literal may be followed by `d` or `D` letter.	`123D`
		`1.23`
		`1.23D`
		`1.2E-3`
		`12.3E4D`

`float`	>A sequence of digits with one decimal point symbol `.` and/or `e` (or `E`) letter. The literal must be followed by `f` or `F` letter.	`123F`
		`1.23F`
		`1.2E-3F`
		`1.2E3F`

`boolean`	`true` or `false`	`true` `false`
`char`	A character in single quotes	`'a'`
`String`	A sequence of characters in double quotes	`"abc"`

Escape Characters

Character	Escape character
Backslash	`\\`
Backspace	`\b`
Carriage Return	`\r`
Single Quote	`\'`

For non printable characters, escape characters may be used in character or string literals. The table on the left lists such escape characters. Examples: `'\''` is a character literal contains a

Operators

Operators perform a function on its operands to produce a value. Operators behave like predefined functions in the Java programming language. For, example the expression `A+B`, which is built from the operands `A` and `B`, and the operator `+` If `A` and `B` both are of type `int`, `A+B` returns their sum. An operator may require one, two or three operands. An operator requiring one operand is called a unary operator, requiring two operands is called a binary, and requiring three operators is called a ternary operator.
Most of the operators operate on expressions returning primitive data types. Here *expression* refers to variables, literals, value of a primitive type returned from a method, or other expressions formed by operators. Only the operators `=`, `==`, and `!=` work also with reference types and `+` and `+=` works also on `String` objects (these will be explained in the following sections). There are six basic kinds of operators: arithmetic, logical, bitwise, assignment, comparison, and ternary if-else operators. It is also possible to combine expressions formed by operators to form compound expressions with more than one operators.
Precedence rules are applied to determine the order of evaluation in a compound statement built using more than one operators. For example, * and / are evaluated before + and -. Usually it is not easy to remember other precedence rules. Therefore, it is preferred to group simple expressions by enclosing parentheses to form compound expressions. By this way, you can reduce the ambiguity in the expression evaluation. For example, in the following lines, the values of `i` and `j` will be different although they seem to be equal to each other.

```
i=a+b+c/d+e;
j=a+(b+c)/(d+e);
```

Arİthmetİc Operators

Arithmetic operators can perform mathematical functions on numeric data types (Integers and Decimal Numbers). The following table lists the arithmetic operators supported by the Java programming language.
Note that `++` and `--` operators also change the value of their operands after the operation. The result of the operation is determined according to the position of the operator.

Expression	Function
`++A`	Pre increment: This expression is equivalent to `A=A+1`, and the result is equal to `A+1`.
`--A`	Pre decrement: This expression is equivalent to `A=A-1`, and the result is equal to `A-1`.
`A++`	Post increment: The result is equal to A, but after the expression evaluated value of `A` is set to `A+1`.
`A--`	Post decrement: The result is equal to A, but after the expression evaluated value of `A` is set to `A-1`.
`+A`	Promotion: If `A` is of type `byte` or `short`, the result will be `A` of type `int`.
`-A`	Negation: Result is negative of `A`.
`A+B`	The result is the sum of `A` and `B`.
`A-B`	The result is `B` subtracted from `A`.
`A*B`	The result is `A` multiplied by `B`.
`A/B`	The result is `A` divided by `B`. If both `A` and `B` are integers, integer division is performed (i.e., The result is the integer part of `A/B`).
`A%B`	The result is remainder from the integer division `A/B`.

Comparison Operators

Comparison operators compare the values of two operands and produce a `boolean` result. The following table lists the comparison operators supported by the Java programming language.

Expression	Function
`A==B`	Equal to: If the value of `A` is equal to the value of `B` then the result will be `true` otherwise, the result will be `false.`
`A!=B`	Not equal to: If the value of `A` is not equal to the value of `B` then the result will be `true`, otherwise the result will be `false.`
`A<B`	Less than: If the value of `A` is less than the value of `B` then the result will be `true`, otherwise the result will be `false.`
`A<=B`	Less than or equal to: If the value of `A` is less than or equal to the value of `B` then the result will be `true`, otherwise the result will be `false.`
`A>B`	Greater than: If the value of `A` is greater than the value of `B` then the result will be `true`, otherwise the result will be `false.`
`A>=B`	Greater than or equal to: If the value of `A` is greater than or equal to the value of `B` then the result will be `true` otherwise the result will be `false.`

Logical Operators

Logical operators operate on boolean values to perform boolean operations like NOT, AND, OR, or XOR (exclusive or). The following table lists the logical operators supported by the Java programming language.
Note that conditional AND and conditional OR operations may not evaluate the second operand depending on the value of the first operand.

<table>
<tr><th>Expression</th><th>Function</th></tr>
<tr><td>!A</td><td>NOT: if A is true then the result will be false; if A is false then the result will be true.</td></tr>
<tr><td>A&B</td><td>AND: If both A and B are true then the result will be true, otherwise the result will be false.</td></tr>
<tr><td>A&&B</td><td>Conditional AND: If A is true, then the result will be the value of B, otherwise the result will be false (B will not be evaluated!).</td></tr>
<tr><td>A|B</td><td>OR: If both A and B are false then the result will be false, otherwise the result will be true.</td></tr>
<tr><td>A||B</td><td>Conditional OR: If A is false then the result will be the value of B, otherwise the result will be true (B will not be evaluated!).</td></tr>
<tr><td>A^B</td><td>XOR: The result will be true if only one of A or B is true. If both A and B are true or both A and B are false then the result will be false.</td></tr>
</table>

Ternary Operator

There is only one ternary operator, ?: in the Java programming language. The expression A?B:C returns the value of B or C depending on the value of A. The first operand, A, must be an expression that returns a boolean value, and if A is true then B is evaluated and its value will be the result of the operation. If A is false then C is evaluated its value will be the result of the operation. Note that, depending on the value of A, one of the operands (B or C) may not be evaluated!

Bitwise Operators

Bitwise operators perform bit by bit operations, such as NOT (~), AND (&), OR (|), XOR (^), or shift (left or right), on operands. Operands in these operators must be integers (& and | also operate on booleans as we have seen before), and operations are carried out on binary representations of the values of operands.

If bit-n of a value is the bit at the nth position of the value's binary representation, the bit-n of the result for ~A, A&B, A|B, A^B are evaluated according to:

bit-n of A	bit-n of B	bit-n of ~A	bit-n of A&B	bit-n of A|B	bit-n of A^B
0	0	1	0	0	0
1	0	0	0	1	1
0	1	1	0	1	1
1	1	0	1	1	0

Shift operators shift the bits of the first operand to left or right by distance specified in the second operand, and fills the empty bits with 0 (except right shift operator >>). The following shift operators are suppored by the Java programming language.

Expression	Function
`A<<B`	Left Shift: Shifts the bits of A to the left by B positions, with 0's shifted in from the right (high order bits are lost)
A>>B	Signed Right Shift: Shifts the bits of A to the right B positions. If A is negative, 1's are shifted in from the left; if it is positive, 0's are shifted in (sign of A is preserved)
A>>>B	Unsigned Right Shift: Shifts the bits of A to the right by B positions, with 0's shifted in from the right.

Assignment Operators

The basic assignment operator, `=`, assigns left operand the value of the right operand. For example, expression `A=B` sets the value of `A` to the value of the expression `B`. In this expression, the right operand, `B`, can be any expression producing a value. However, the left operand must be a variable.
The arithmetic, logical and bitwise operators may also be used with `=` operator to perform an operation with operands and assign the value to the left operand. For example, the operator `+=` can be used as `A+=B` to add two operands `A` and `B` and assign result to `A`. In this case, like `=` operator, the right operand can be any expression producing a value, and the left operand must be a variable. The following table lists such shortcut assignment operators and their equivalents.

Expression	Equivalent Expression
`A+=B`	`A=A+B`
`A-=B`	`A=A-B`
`A*=B`	`A=A*B`
`A/=B`	`A=A/B`
`A%=B`	`A=A%B`
`A&=B`	`A=A&B`
`A|=B`	`A=A|B`
`A^=B`	`A=A^B`
`A<<=B`	`A=A<<B`
`A>>=B`	`A=A>>B`
`A>>>=B`	`A=A>>>B`

Reference Data Types

A variable of a reference data type may refer to an object or array (As it will be explained in upcoming lectures it may also refer to interfaces and classes). Since variables of reference data types is only a reference to the actual item to which they refer, their use is very different from variables of primitive data types.
Declaration of a variable of reference type is very similar to declaration of a variable of primitive type.

```
TypeName variableName
```

Here, ***TypeName*** is the name of the type for the entity that will be referred to, ***variableName*** must be a valid identifier. Like primitive data types, method parameters are initialized by the caller.
Member variables are initialized to `null` value, which means an empty reference, and they may be initialized later. But you must explicitly initialize local variables before using them.
By declaring the variable, you simply create a placeholder (a reference storage area) for the variable only, not for the entity that will be referred to. Therefore, a simple initialization could be copying the reference of an entity from another variable by using `=` operator or creating a new entity with `new` operator and passing its reference to the variable using `=` operator. The following example illustrates the declaration and initialization of such variables.

```
MyClass x = new MyClass();
MyClass y = x;
```

In this example, variable `x` of type `MyClass` is declared and initialized by creating an instance of `MyClass`. The `new` operator is used to create an instance of an entity. Then, another variable, `y`, is declared and initialized by `x`. Therefore both variables refer to the same object. If the entity is of type `class` then while using the `new` operator you actually call the constructor for that class. The constructor allocates necessary resources for the object and returns an instance of the object.
The assignment operator `=`, and comparison operators `==` and `!=` may also be used with reference variables. But, you shouldn't forget that reference variables only contains a reference to actual entities. So, `=` operator only copies the reference of an entity to another variable, and `==` and `!=` operators only compares references (not their contents!) and return `true` if they are equal to each other.

Garbage Collector

After you created an entity using `new` operator, resources are allocated and a reference for that object is returned. In many other programming languages, you have to cleanup the resources (or destroy) for any explicitly created entity when they are no longer needed. However, in the Java, you don't need to destroy entities that are created by `new` operator. Garbage collection mechanism destroys them for you, when it is detected that they will be no longer used.
When there is no variables referencing an object, the object is put into a garbage collector. That means, when all the variables referencing the entity go out of scope, assigned to different entities, or assigned to `null` value, then the entity is automatically destroyed.

Wrapper Classes

Wrapper classes for primitive data types, arrays and strings are some simple reference data types that are frequently used in programs.

In some cases, you need an object containing a value of primitive data type (one use of wrapper classes will be explained later when Collections are explained). The following table lists the wrapper classes corresponding to each primitive data type.
Variables of these classes can be declared and initialized like ordinary reference variables. The objects of type any wrapper class can be created by using the `new` operator and passing one value of the corresponding type in the `constructor`.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

In order to use the value contained in an object of type wrapper class, you may call the object's `xxxValue()` methods, where *xxx* is the name of the corresponding primitive data type name (e.g., `byteValue()`, `intValue()`, `booleanValue()`, etc).
For example, the following code creates two objects of type `Integer` and assign them to two variables `i`, and k and `s`:

```
Integer i = new Integer(5);
int j = 5;
Integer k = new Integer(j);
boolean b = i==k;
Integer s = k;
boolean e = s==k;
j = i.intValue() + k.intValue();
```

Note that, in the above example, although they contain the same value, `5`, the variables `i` and `k` are not equal to each other (i.e., `b` is `false`). However, since they refer to the same object, variables `s` and `k` are equal to each other (i.e., `e` is `true`).

ARRAYS

An array can contain a group of values of the same data type. In order to declare an array, you may put square brackets, `[]`, either after the variable name or after the data type. For example,

```
int[] j
int k[]
```

Declare variable `j`, and `k` as one dimensional array of elements of primitive type `int`. Like other reference types, declaring an array does not allocate memory for that array. Arrays must be created before they are used (so the variable is initialized). Arrays also created by `new` operator as `new elementType[size]`. For example,

```
int[] j = new int[10];
int k[];
k = new int[5];
```

creates arrays of size `10` and `5` and assigns references to them to variables `i` and `j`, respectively. Array members may also be accessed by brackets as `variableName[index]`. The `index` starts at `0` and last element is the `size-1`. The following example, creates an array of `char` of size `3` and initializes its members:

```
char[] c = new char[3];
c[0] = 'a';
c[1] = 'b';
c[2] = c[1];
```

There is also a shortcut way of creating and initializing members of an array using curly braces, `{}`. The following example illustrates shortcut initialization for an array.

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

Size of an array can be learned by the `length` variable. For example, `c.length` will return `3` for the array in above example. It is possible to create multi-dimensional arrays and arrays containing elements of reference types. Arrays will be explained in detail later.

STRINGS

In the Java programming language, `String` objects are used to carry sequence of characters. Unlike other programming languages, the string contained in the `String` object cannot be changed after assignment. That is, `String`s are constant once they are created. Instead, you can assign a new reference to a variable of type `String`. The following example demonstrates different ways of creating string objects and assigning values to variables of type `String`.

```
String s = new String("abc"); // Create using constructor
String t = "abc"; // shortcut initialization
s = "abccd"; // shortcut assignment
```

Operators `+` and `+=` operate also on `String`s to create new `String` objects from the values of the operands. Operator `+` may also be used with any operand of any type. For example, the value of `s` after the following code executed will be `"abc105"`.

```
String s;
s = "abc";
s += 1;
s += '0';
int j=5;
s = s+j;
```

In order to compare the values contained in two `String` variables, you should use `String`'s `compareTo()` method (since comparing variables directly only check the references, not the values

contained in). For example, if you want to compare the values in `A` and `B` of type `String`, you may call `A.compareTo(B)` which returns `0` if they contain the same string, returns `-1` if `A<B`, and returns `1` if `A>B`.
Primitive data types and `String`s can be converted to each other using the appropriate methods of the `String` class and wrapper classes for the primitive types. Primitive types can be converted to the `String` using the primitive type's corresponding wrapper class's `toString(value)` method (since `toString` is a class method, you can use it without any instance using the class name directly), and `String`s can be converted to the a primitive type using the `String`'s `valueOf(string)` method. For example:

```
int i = 5;
String s = Integer.toString(i);
i = String.valueOf(s);
```

Statements

A statement is a single command in a program. An individual statement ends with a semicolon (`;`). A single statement may cover multiple lines of code, but everything up to semicolon is processed as a single command. A statement may

Declare a variable,
Create a reference data type,
Assign a value to a variable,
Increment or decrement a variable (using `++` or `--` operators)
Call a method,
Complete the execution of a method (`return return_value;`).

Multiple statements may be enclosed by curly braces (`{` and `}`) to form compound statements. Such statements can be used where only one statement is allowed (in execution control statements that will be explained). The following example illustrates a simple statement block

```
...
{
   int i = 0;
   i += i+1;
   String s = i.toString();
   s = "2*2=" + s;
}
...
```

While the program executes, statements are executed in the order they are written. However, It is possible to control the execution of statements according to some condition using control flow statements. There are three main groups of control flow

statements: Loop statements, conditional statements, and branching statements. Since loop and conditional statements are themselves also statements, they can be nested in any depth (they can contain other control flow statements in their body).

Loop Statements

Loop statements enable execution of the same statement (or statements) zero or more times according to a given condition expression. There are three kinds of loop statements in the Java programming language: `while`, `do-while`, and `for`.

A `while` statement executes a single statement or a statement block (if you want to execute more than one statements) as long as the condition given to it is `true`. The condition expression must return a `boolean` value. The format of `while` statement is as follows:

```
...
while (condition) statement;
// or
while (condition){
  statement 1;
  ...
  statement n;
}
```

A `do-while` statement does a similar execution. It executes one statement or statement block as long as the condition given to it is `true`. However, it differs from `while` statement in that the statement(s) following `do` is executed at least once, whereas the statement(s) following `while` may not be executed depending on the value of the condition expression. The format of the `do-while` statement is as follows:

```
...
do statement while(condition);
// or
do {
  statement 1;
  ...
  statement n;
} while (condition);
...
```

A `for` statement is used to iterate over a range of values. The format of the `for` statement is as follows:

```
...
for(initialization;condition;iteration) statement;
// or
for(initialization;condition;iteration){
  statement 1;
  ...
  statement n;
};
...
```

Here *initialization* is a statement that is executed at the beginning once. It may contain a variable declaration and initialization, or it may only initialize a control variable. *iteration* statement is executed each time the following statement(s) is executed until the *condition* expression evaluates a `false`.

Conditional Statements

Conditional statements enable deciding which statement (or statements) to execute according to the given condition expression. There are three kinds of loop statements in the Java programming language: `if`, `if-else`, and `switch`.
An `if` statement is used to decide whether to execute or not a statement or a statement block. If the condition is `true`, the statement(s) is executed, otherwise, the execution continues from the next statement following `if`. The format of the `if` statement is as follows:

```
...
if(condition) statement;
// or
if(condition){
  statement 1;
  ...
  statement n;
}
...
```

The condition expression must return a `boolean` value. A `do-while` statement does a similar execution. It
An `if-else` statement is used to decide which statement or statement block to execute. If the condition is `true`, the first statement or statement block is executed, otherwise the second statement or statement block is executed. The format of the `if-else` statement is as follows:

```
if(condition) statement1; else statement2;
// or
if(condition){
  statement 1_1;
  ...
  statement 1_n;
} else {
  statement 2_1;
  ...
  statement 2_m;
}
```

The `else` part may also contain other `if` statements:

```
if(condition 1){
  statement 1_1;
  ...
  statement 1_n;
} else if (condition 2){
```

```
  statement 2_1;
  ...
  statement 2_m;
} else if (condition 3) {
  statement 3_1;
  ...
  statement 3_k;
} else {
  statement 4_1;
  ...
  statement 4_t;
}
```

A `switch` statement can be used to choose a statement or statements to execute according to an integer or character value (coming from a variable or expression). The format of the `switch` statement is as follows:

```
switch(expression){
case <value1>:
  statement 1;
  break;
case <value2>:
  statement 2;
  break;
...
case <value n>:
  statement n;
  break;
default:
  statement n+1;
}
```

Where, *expression* produces an integer or character value, and the statement to execute is determined according to its value. For example, if its value equal to <value 2> the section `case <value 2>` is executed until the `break` statement. The `break` statement causes termination of `switch` block and execution continues with the next statement following `switch` block. When none of the case <value i> lines match, the `default` part is executed (if not required `default` section may be omitted).
It is possible to combine multiple `case` sections as:

```
switch(expression){
case <value1>:
case <value2>:
case <value3>:
  statement 1;
  break;
case <value4>:
case <value5>
  statement 2;
  break;
...
case <value n>:
  statement n;
  break;
```

```
default:
  statement n+1;
}
```

In this case, if *expression* is equal to one of <*value1*>, <*value2*> or <*value3*>, the *statement 1* is executed.

Branching Statements

`break`, and `continue` statements can control the execution of statements in `while`, `do-while`, and `for` blocks. There are two forms of these statements: plain and labeled. Plain form is used to control execution of the current innermost statement block, and labeled form is used to control execution in nested statement blocks. A label can be given to a statement block as:

```
label:
control_flow_statement;
```

`break` statement terminates the innermost block if used in the plain form, and terminates the specified block if used in labeled form. For example;

```
...
for_loop:
for(int i=0; true; i++) {
  ...
  int j=0;
  while(true){
    j++;
    ...
    if(i==100) break for_loop;
    ...
    if(j>=200) break;
    ...
  }
}
```

In this example, the `break for_loop;` statement causes outer for loop to terminate and the plain `break;` statement causes inner while loop to terminate (for loop continues execution).
The `continue` statement is used to skip current iteration of `while`, `do-while`, or `for` loops. If it is used in the plain form, it skips the current iteration of the current innermost block. If it is used in the labeled form, it skips current iteration of the specified block. The execution continues with the execution of the statement control part and according to the value of condition expression, further iterations may continue, or not.