CHAPTER -2

Lexical Issues

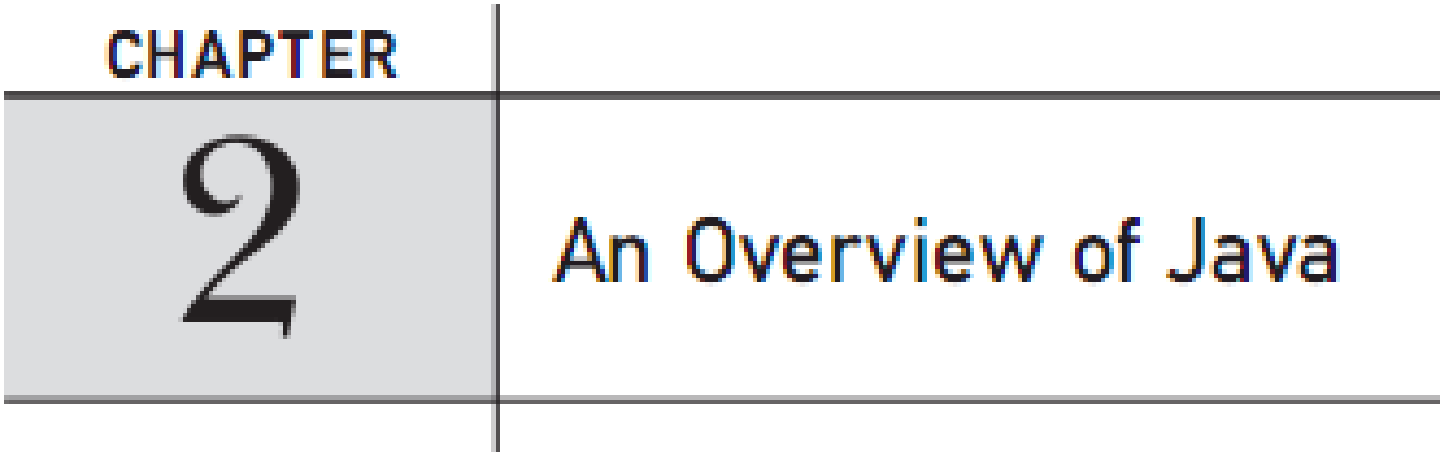
whitespace, identifiers, literals, comments, operators, separators, and keywords.

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

It is not enough just to write code that works. It is as important-perhaps more important to write code well; not merely code that works, but code that is legible, maintainable, reusable, fast, and efficient.

CHAPTER -2

An Overview of JAVA



Lexical Issues: White Space & Identifier

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- **Whitespace**

  In Java, whitespace is a space, tab, or newline.
- **Identifiers**

  Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so VALUE is a different identifier than Value. Some examples of valid identifiers are

AvgTemp	count	a4	$test	this_is_ok

Invalid identifier names include these:

2count	high-temp	Not/ok

Lexical Issues: Literals & Comments

Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals:

100	98.6	'X'	"This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment*. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a */. Documentation comments are explained in the Appendix.

Lexical Issues: Separators

In Java, there are a few characters that are used as separators.

The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference. (Added by JDK 8.)

The Java Keywords

- There are 50 keywords currently defined in the Java language

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

The Java Class Libraries

The sample programs shown in this chapter make use of two of Java's built-in methods: **println( ) and print( ).**

As mentioned, these methods are available through **System.out.**

System is a class predefined by Java that is automatically included in your programs.

In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics.

The standard classes also provide support for a graphical user interface (GUI). Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java.

Indeed, part of becoming a Java programmer is learning to use the standard Java classes.

CHAPTER -3

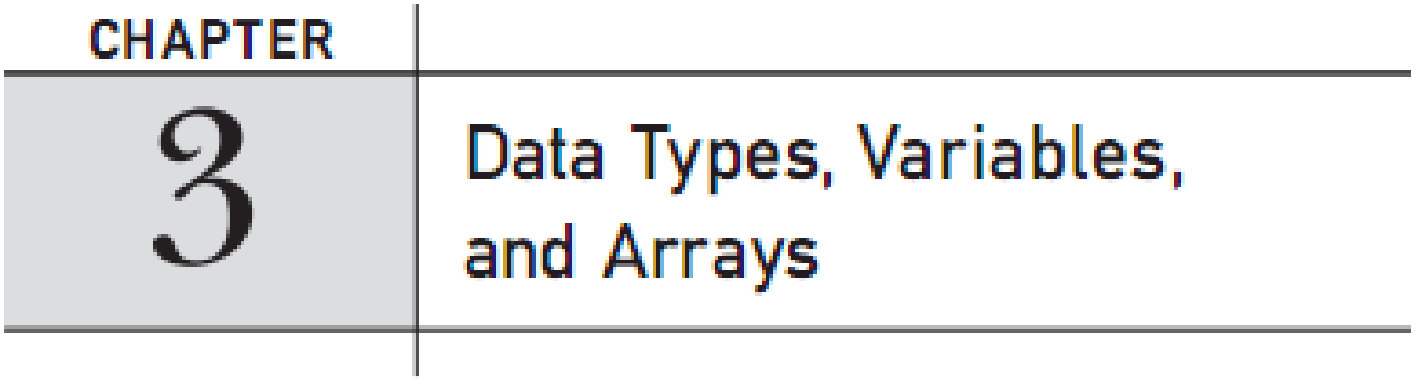
Data Types, Variables, and Arrays

Md. Mamun Hossain
B.Sc. (Engg.) & M.Sc. (Thesis) in CSE , SUST
Assistant Professor, Dept. of CSE, BAUST

It is not enough just to write code that works. It is as important-perhaps more important to write code well; not merely code that works, but code that is legible, maintainable, reusable, fast, and efficient.

CHAPTER -3

Data Types, Variables, and Arrays

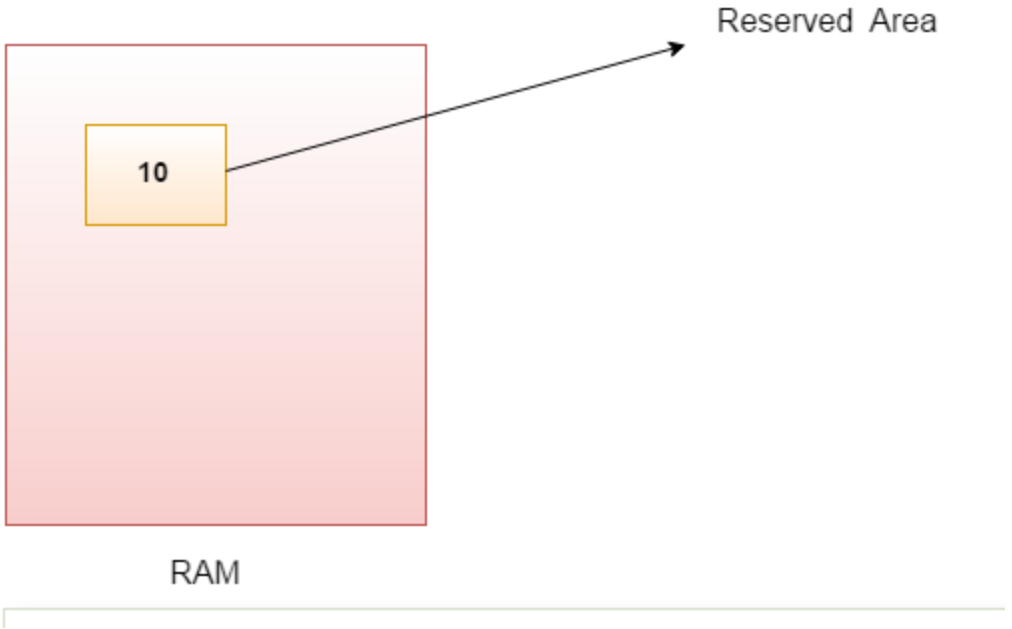


Java Variables

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.
- In addition, all variables have a scope, which defines their visibility, and a lifetime.
- Every variable has a type declared in the source code

Java Variables

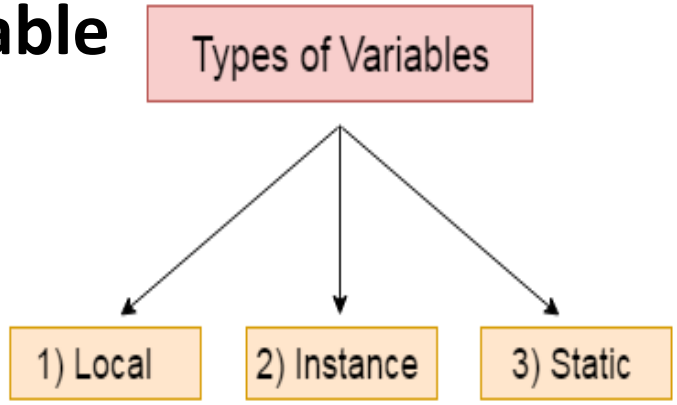
- Variable is name of reserved area allocated in memory.
- In other words, it is a name of memory location.
- It is a combination of "vary + able" that means its value can be changed.



Java Variables : Types

There are three types of variables in java:

- ✓**Local Variable**
  - A variable which is declared inside the method is called local variable.
- ✓**Instance Variable/Class Variable**
  - A variable which is declared inside the class but outside the method, is called instance variable.
- ✓**Static variable**
  - A variable that is declared as static is called static variable. It cannot be local.



Example of Variables

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Variable: Declarations & Initialization

- Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;              // declares three ints, a, b, and c.
int d = 3, e, f = 5;      // declares three more ints, initializing
                          // d and f.
byte z = 22;              // initializes z.
double pi = 3.14159;      // declares an approximation of pi.
char x = 'x';             // the variable x has the value 'x'.
```

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
// Demonstrate dynamic initialization.
class DynInit {
  public static void main(String args[]) {
    double a = 3.0, b = 4.0;

    // c is dynamically initialized
    double c = Math.sqrt(a * a + b * b);

    System.out.println("Hypotenuse is " + c);
  }
}
```

The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

- So far, all of the variables used have been declared at the start of the main() method.
- However, Java allows variables to be declared within any block.
- A block is begun with an opening curly brace and ended by a closing curly brace.
- A block defines a ***scope***. Thus, each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program.
- It also determines the ***lifetime*** of those objects.

The Scope and Lifetime of Variables

Many other computer languages define two general categories of scopes: ***global and local***. However, these traditional scopes do not fit well with Java's strict, object-oriented model.

- ✓ In Java, the two major scopes are those defined by a ***class*** and those defined by a ***method***.
- ✓ Class scope has several unique properties and attributes that do not apply to the scope defined by a method.

For now, we will only examine the scopes defined by or within a method.

Scope &Lifetime of Variables: Example

```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
      int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

Scope &Lifetime of Variables : Example

Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. For example, the following program is illegal:

```
// This program will not compile
class ScopeErr {
  public static void main(String args[]) {
    int bar = 1;

    {                  // creates a new scope
      int bar = 2;   // Compile-time error - bar already defined!
    }
  }
}
```

Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

- ✓ It makes your program **memory efficient** (i.e it saves memory).
- ✓ **Java static property is shared to all objects.**

Understanding the problem without static

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it *static*, this field will get memory only once.

Rules for defining Java Identifiers

There are certain rules for defining a valid java identifier. These rules must be followed, otherwise we get compile-time error. These rules are also valid for other languages like C,C++.

- The only allowed characters for identifiers are all alphanumeric characters([**A-Z**],[**a-z**],[**0-9**]), '**$**'(dollar sign) and '**_**' (underscore).For example "geek@" is not a valid java identifier as it contain '@' special character.
- Identifiers should **not** start with digits(**[0-9]**). For example "123geeks" is a not a valid java identifier.
- Java identifiers are **case-sensitive**.
- There is no limit on the length of the identifier but it is advisable to use an optimum length of 4 – 15 letters only.
- **Reserved Words** can't be used as an identifier. For example "int while = 20;" is an invalid statement as while is a reserved word. There are **53** reserved words in Java.

Valid Identifiers

Examples of valid identifiers :

```
MyVariable
MYVARIABLE
myvariable
x
i
x1
i1
_myvariable
$myvariable
sum_of_array
geeks123
```

Valid Identifiers

Examples of invalid identifiers :

```
My Variable  // contains a space
123geeks   // Begins with a digit
a+c // plus sign is not an alphanumeric character
variable-2 // hyphen is not an alphanumeric character
sum_&_difference // ampersand is not an alphanumeric character
```

Identifiers : Reserved Words

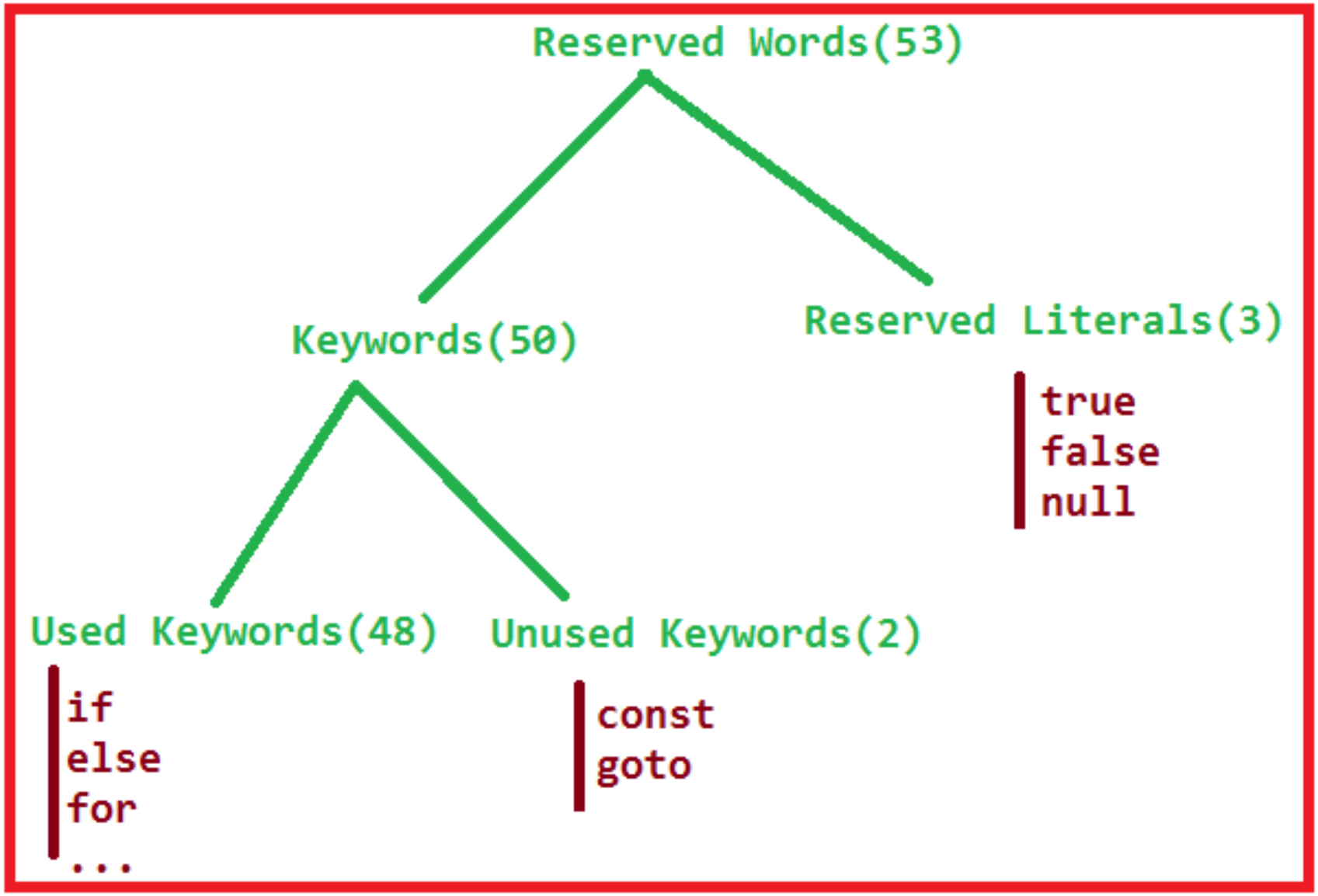
- Any programming language reserves some words to represent functionalities defined by that language. These words are called reserved words.They can be briefly categorised into two parts :

  keywords(50) and **literals**(3).

- keywords define functionalities and literals defines a value.

- Identifiers are used by symbol tables in various analyzing phases(like lexical, syntax, semantic) of a compiler architecture.
- **Note :** The keywords const and goto are reserved, even though they are not currently used. In place of const, final keyword is used.

Identifiers : Reserved Words



Java: Data Types

There are two types of data types in java:

primitive and non-primitive

- **Primitive** types directly contain values.

  There are 8 primitive types:

  byte, short, int, long, char, float, double, boolean.

- **Non-primitive**

  There are two non-primitive types:

  derived type and user defined / Reference Type

  Derived type are Array, String etc

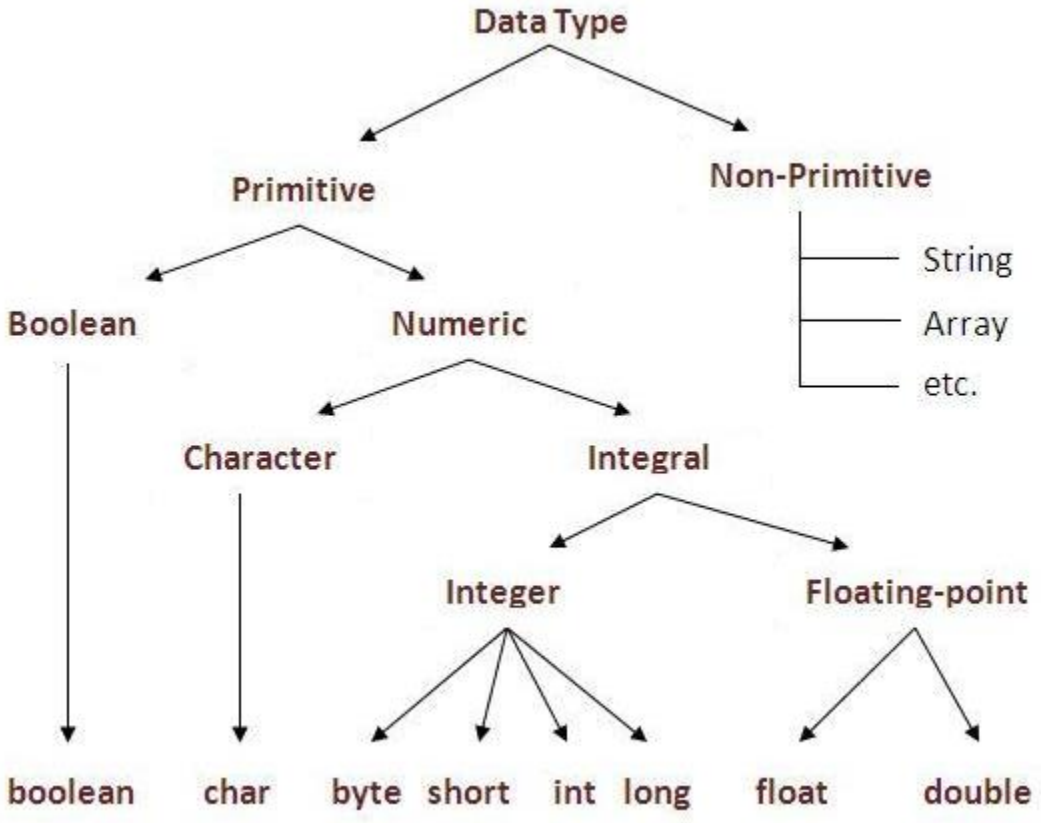
  Reference types are references to objects.

Data Types in Java

- Data types represent the different values to be stored in the variable.

In java, there are two types of data types:

- **Primitive data types**
- **Non-primitive data types**



The Primitive Types

Java defines eight *primitive* types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. The primitive types are also commonly referred to as *simple* types, and both terms will be used in this book. These can be put in four groups:

- **Integers** This group includes **byte**, **short**, **int**, and **long**, which are for whole-valued signed numbers.
- **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.
- **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

Integer

- Java defines four integer types: **byte, short, int, and long**. All of these are ***signed, positive and negative values***. Java does not support unsigned, positive-only integers.
- Java's designers felt that unsigned integers were unnecessary, Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator.

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Integer

- ***Byte***: byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
- ***Short***: It is probably the least-used Java type. Small arithmetic and algebraic calculation but don't forget to type cast when evaluate an expression.
- ***Int***: The reason is that when byte and short values are used in an expression, they are promoted to int when the expression is evaluated. (Type promotion is described later in this chapter.) Therefore, int is often the best choice when an integer is needed.
- ***Long***: Long is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed.

Floating-Point : Default double

There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

- ✓ **float can be useful when representing dollars and cents**
- ✓ **All transcendental math functions, such as sin(), cos(), and sqrt(), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.**

Character : Unicode

- **In C/C++, char is 8 bits wide. This is not the case in Java.**
- Instead, Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. At the time of Java's creation, Unicode required 16 bits.
- **Thus, in Java char is a 16-bit type. The range of a char is 0 to 65,536. There are no negative chars.**
- The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability.

Character : Example

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
  public static void main(String args[]) {
    char ch1, ch2;

    ch1 = 88; // code for X
    ch2 = 'Y';

    System.out.print("ch1 and ch2: ");
    System.out.println(ch1 + " " + ch2);
  }
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

NOTE In the formal specification for Java, **char** is referred to as an *integral type*, which means that it is in the same general category as **int**, **short**, **long**, and **byte**. However, because its principal use is for representing Unicode characters, **char** is commonly considered to be in a category of its own.

Character : Example

- Booleans Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of ***a<b***. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

```
// Demonstrate boolean values.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;

    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
  }
}
```

The output generated by this program is shown here:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Data Type: default value

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Data Types value and size

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Data Types value and size

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Data Types value and size

- **Why char uses 2 byte in java and what is \u0000 ?**
    - It is because java uses Unicode system than ASCII code system.
    - The \u0000 is the lowest range of Unicode system.

Integer Literals

- **Integer Literals:**
  - **decimal** values : Examples are 1, 2, 3, and 42.
  - **octal** (base eight) : Octal values are denoted in Java by a leading zero.
  - **hexadecimal** (base 16): hexadecimal constant with a leading zero-x, (0x or 0X)
  - To specify **long** : appending an upper- or lowercase L to the literal. For example, 0x7fffffffffffffL or 9223372036854775807L is the largest long
  - Beginning with JDK 7, you can also specify integer literals using **binary**. To do so, prefix the value with 0b or 0B.

  For example, this specifies the decimal value 10 using a binary literal: int x = 0b1010;

Integer Literals

- Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

```
int x = 123_456_789;
```

the value given to x will be 123,456,789.

- The underscores will be ignored. Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits. For example, this is valid:

```
int x = 123___456___789;
```

The use of underscores in an integer literal is especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers, and so on.

They are also useful for providing visual groupings when specifying binary literals. For example,

```
int x = 0b1101_0101_0001_1010;
```

Floating-Point Literals

- Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant.
- You can also explicitly specify a double literal by appending a D or d. Doing so is, of course, redundant.
- Hexadecimal floating-point literals are also supported, but they are rarely used. They
- must be in a form similar to scientific notation, but a P or p, rather than an E or e, is used.

  For example, 0x12.2P2 is a valid floating-point literal.

The value following the P, called the binary exponent, indicates the power-of-two by which the number is multiplied. Therefore,

0x12.2P2 represents 72.5

- double num = 9_423_497_862.0; =9,423,497,862.0
- It is also permissible to use underscores in the fractional portion of the number. For example,

  double num = 9_423_497.1_0_9;

is legal. In this case, the fractional part is .109.

Boolean Literals

- Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The values of true and false do not convert into any numerical representation.
- The true literal in Java does not equal 1, nor does the false literal equal 0.
- In Java, the Boolean literals can only be assigned to variables declared as boolean or used in expressions with Boolean operators.

Character Literals

- A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.
- For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as ' \' ' for the single-quote character itself and ' \n' for the newline character.
- There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, ' \141' is the letter 'a'.
- For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, ' \u0061' is the ISO-Latin-1 'a' because the top byte is zero. ' \ua432 ' is a Japanese Katakana character.

String Literals

- String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

  "Hello World"

  "two\nlines"

  " \"This is in quotes\""

String Literals

- The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in some other languages.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal Unicode character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Table 3-1 Character Escape Sequences