Basics of JAVA

The PYPL PopularitY of Programming Language Index is created by analysing how often language tutorials are searched on Google.

Worldwide, Dec 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.71 %	+4.1 %
2		Java	19.29 %	-2.2 %
3		Javascript	8.33 %	+0.0 %
4		C#	7.27 %	-0.4 %
5		PHP	6.32 %	-1.0 %
6		C/C++	6.0 %	-0.3 %
7		R	3.79 %	-0.2 %
8		Objective-C	2.61 %	-0.6 %
9		Swift	2.5 %	-0.1 %
10		Matlab	1.84 %	-0.2 %

Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

The Top Programming Languages 2019 according to IEEE Spectrum.

What is Java

- Java is a **programming language** and a **platform**.
- Java is a high level, robust, object-oriented and secure programming language.
- **Platform**: Any hardware or software environment in which a program runs, is known as a platform.
- Since Java has a runtime environment (JRE) and API, it is called a platform.

Types of Java Applications

- There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

- Standalone applications are also known as desktop applications or window-based applications.

2) Web Application

- An application that runs on the server side and creates a dynamic page is called a web application.

3) Enterprise Application

- An application that is distributed in nature, such as banking applications, etc. is called enterprise application.

4) Mobile Application

- An application which is created for mobile devices is called a mobile application.

Java Platforms / Editions

1) Java SE (Java Standard Edition)

- It is a Java programming platform.

2) Java EE (Java Enterprise Edition)

- It is an enterprise platform which is mainly used to develop web and enterprise applications.

3) Java ME (Java Micro Edition)

- It is a micro platform which is mainly used to develop mobile applications.

History of Java

- Java language project was initiated by **James Gosling**, **Patrick Naughton, Chris Warth, Ed Frank**, and **Mike Sheridan** at Sun Microsystems, Inc. in 1991.
- This small team of sun engineers called **Green Team**.
- Primary motivation was the need for a platform-independent (that is, architecture-neutral)language that could be used to create software to be embedded in various consumer electronic devices, such as set-top boxes, microwave ovens and remote controls.
- Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
- After that, it was called **Oak** and was developed as a part of the Green project.
- In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"

- The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc.
- Java name was chosen by James Gosling while having coffee near his office.
- According to James Gosling, "Java was one of the top choices along with **Silk**".
- Since Java was so unique, most of the team members preferred Java than other names.
- Java is an island of Indonesia where the first coffee was produced (called java coffee).
- Notice that Java is just a name, not an acronym.


The Green Team
James Gosling

Java version history

As of 29 November 2019, both Java 8 ,11 and 13 are officially supported. Major release versions of Java, along with their release dates:

- JDK 1.0 (January 23, 1996)
- JDK 1.1 (February 19, 1997)
- J2SE 1.2 (December 8, 1998)
- J2SE 1.3 (May 8, 2000)
- J2SE 1.4 (February 6, 2002)
- J2SE 5.0 (September 30, 2004)
- Java SE 6 (December 11, 2006)
- Java SE 7 (July 28, 2011)
- Java SE 8 (March 18, 2014)
- Java SE 9 (September 21, 2017)
- Java SE 10 (March 20, 2018)
- Java SE 11 (September 25, 2018)
- Java SE 12 (March 19, 2019)
- Java SE 13 (September 17, 2019)

Source wikipedia

Features of Java (Buzz Words)

- The primary objective of Java programming language creation was to make it portable, simple and secure programming language
- The features of Java are also known as java *buzzwords*.
- A list of most important features of Java language is
  - Simple
  - Object-Oriented
  - Portable
  - Platform independent
  - Secured
  - Robust
  - Architecture neutral
  - Interpreted
  - High Performance
  - Multithreaded
  - Distributed
  - Dynamic

Simple

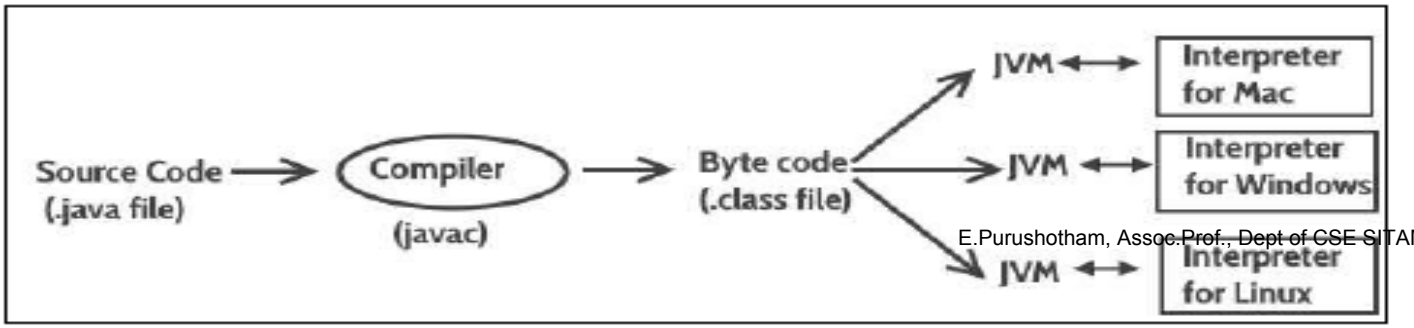
- Java is very easy to learn, and its syntax is simple, clean and easy to understand.
- According to Sun, Java language is a simple programming language because:
  - Java syntax is based on C++ (so easier for programmers to learn it after C++).
  - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
  - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

- Java is an object-oriented programming language.
- Everything in Java is an object.
- Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
- Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.
- Basic concepts of OOPs are:
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

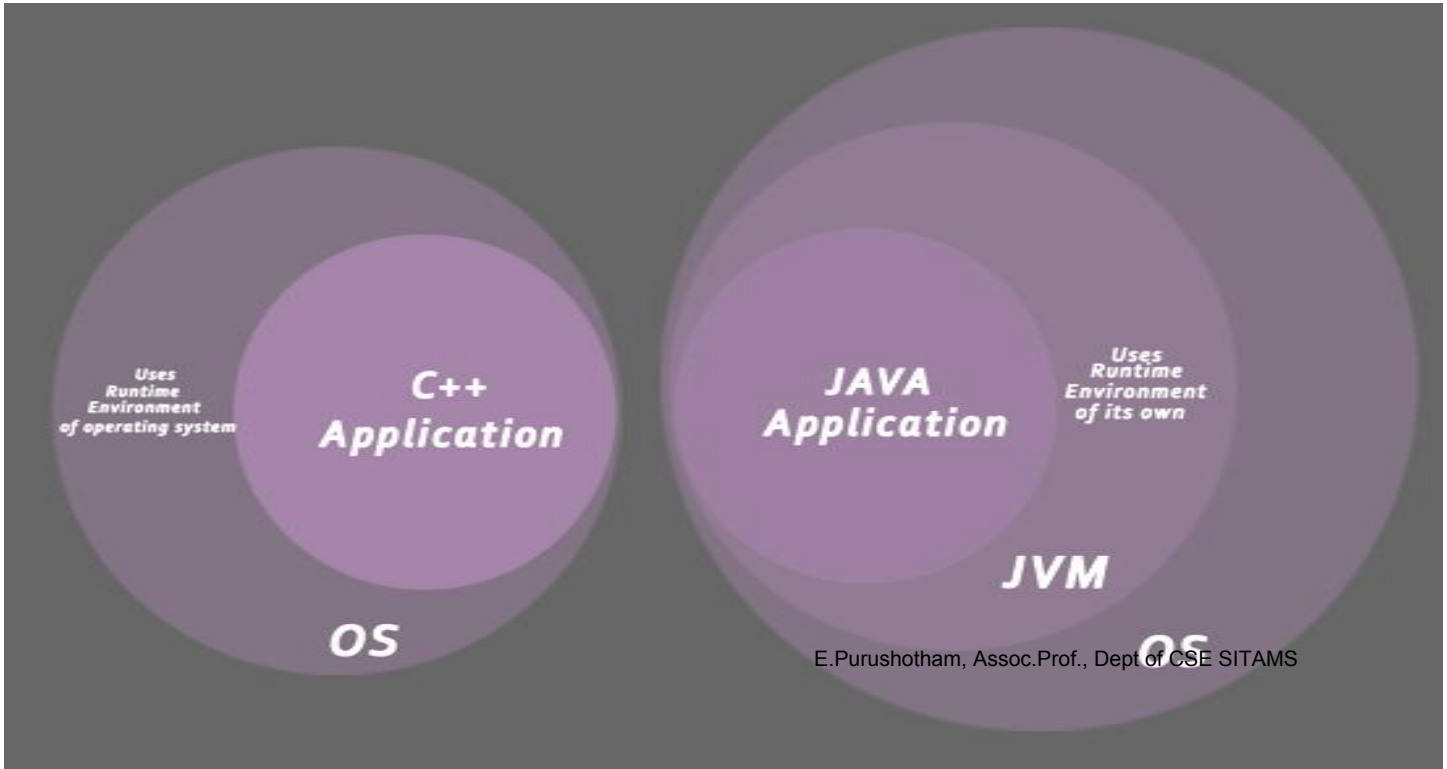
Platform Independent

- Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language.
- Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc.
- Java code is compiled by the compiler and converted into bytecode.
- This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).



Secured

- Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
- No explicit pointer
- Java Programs run inside a virtual machine



Robust

- Robust simply means strong.
- Java is robust because:It uses strong memory management.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

- Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
- In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture.
- However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

- Java is portable because it facilitates you to carry the Java bytecode to any platform.
- No implementation dependent features.

Interpreted and High-performance

- The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.
- Java is faster than other traditional interpreted programming languages.

Distributed

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- In fact, accessing a resource using a URL is not much different from accessing a file.
- Java also supports *Remote Method Invocation (RMI). This feature enables a program to* invoke methods across a network.

Dynamic

- Java is a dynamic language.
- It supports dynamic loading of classes. It means classes are loaded on demand.
- Java supports dynamic compilation and automatic memory management (garbage collection).

Multi Threaded

- Java multithreading feature makes it possible to write program that can do many tasks simultaneously.
- Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

Java Program Structure

```
//  comments about the class
public class MyProgram
{
}
```

class header

class body

Comments can be placed almost anywhere

Java Program Structure

```
//  comments about the class
public class MyProgram
{
   //  comments about the method
   public static void main (String[] args)
   {
   }
}
```

method body

method header

Java Hello World! Program

```
public class Hello
{

   public static void main (String[] args)

   {

       System.out.println ("Hello World program");

   }

}
```

- **class** : class keyword is used to declare classes in Java
- **public** : It is an access specifier. Public means this function is visible to all.
- **static** : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The **main()** method here is called by JVM, without creating any object for class.
- **void** : It is the return type, meaning this function will not return anything.

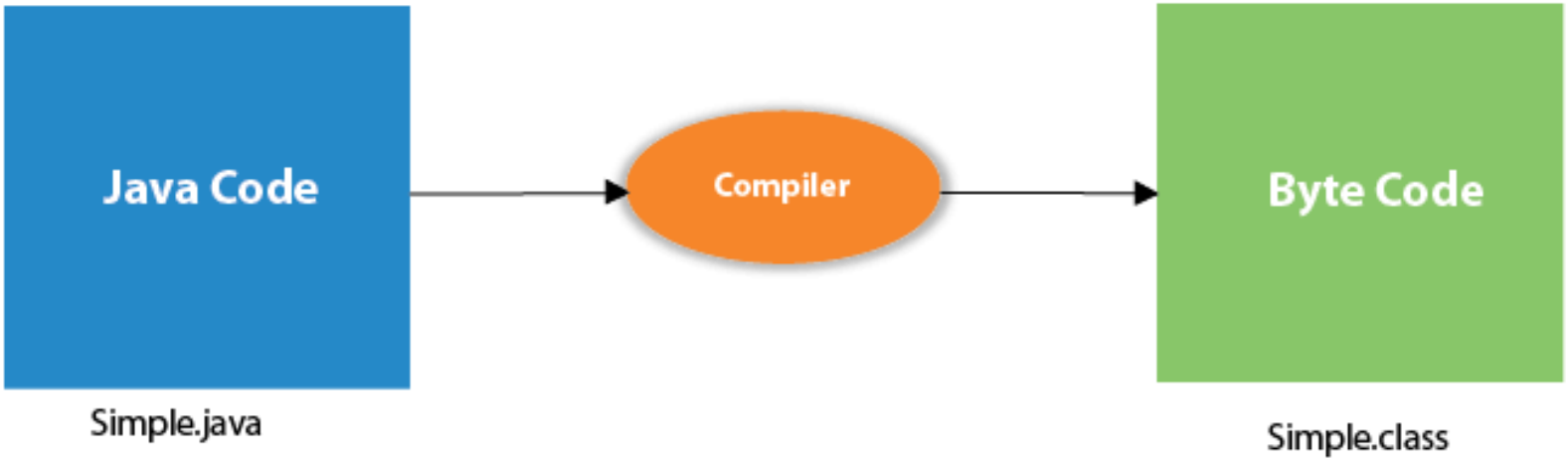
- **main** : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method.
- **String[] args** : This represents an array whose type is String and name is args. We will discuss more about array in Java Array section.
- **System.out.println** : This is used to print anything on the console like *printf* in C language.

Steps to Compile and Run your first Java program

- **Step 1:** Open a text editor and write the code as above.
- **Step 2:** Save the file as Hello.java
- **Step 3:** Open command prompt and go to the directory where you saved your first java program assuming it is saved in C:\
- **Step 4:** Type javac Hello.java and press Return**(Enter KEY)** to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.
- **Step 5:** Now type java Hello on command prompt to run your program.

You will be able to see **Hello world program** printed on your command prompt.

How Java Works?



Lexical Issues

- Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.
- **Whitespace:** space, tab, or newline.
- **Identifiers:** An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.

  They must not begin with a number
- Java is case-sensitive, so **VALUE is a different identifier than Value.**

- **Some examples of valid identifiers are**

  Avg Temp count a4 $test this_is_ok
- **Invalid identifier names include these:**

  2count high-temp Not/ok
- **Literals**

A constant value in Java

- For example, here are some literals:

  100 98.6 'X' "This is a test"

Separators

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

Symbol	Name
( )	Parentheses
{ }	Braces
[ ]	Brackets
;	Semicolon
,	Comma
.	Period

Comments

- Comments in a program are called *inline documentation*
- They should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take three forms:

```
// single line comment

/*  Multiple
 line
comment
        */
/** this is a javadoc comment   */
```

The Java Keywords

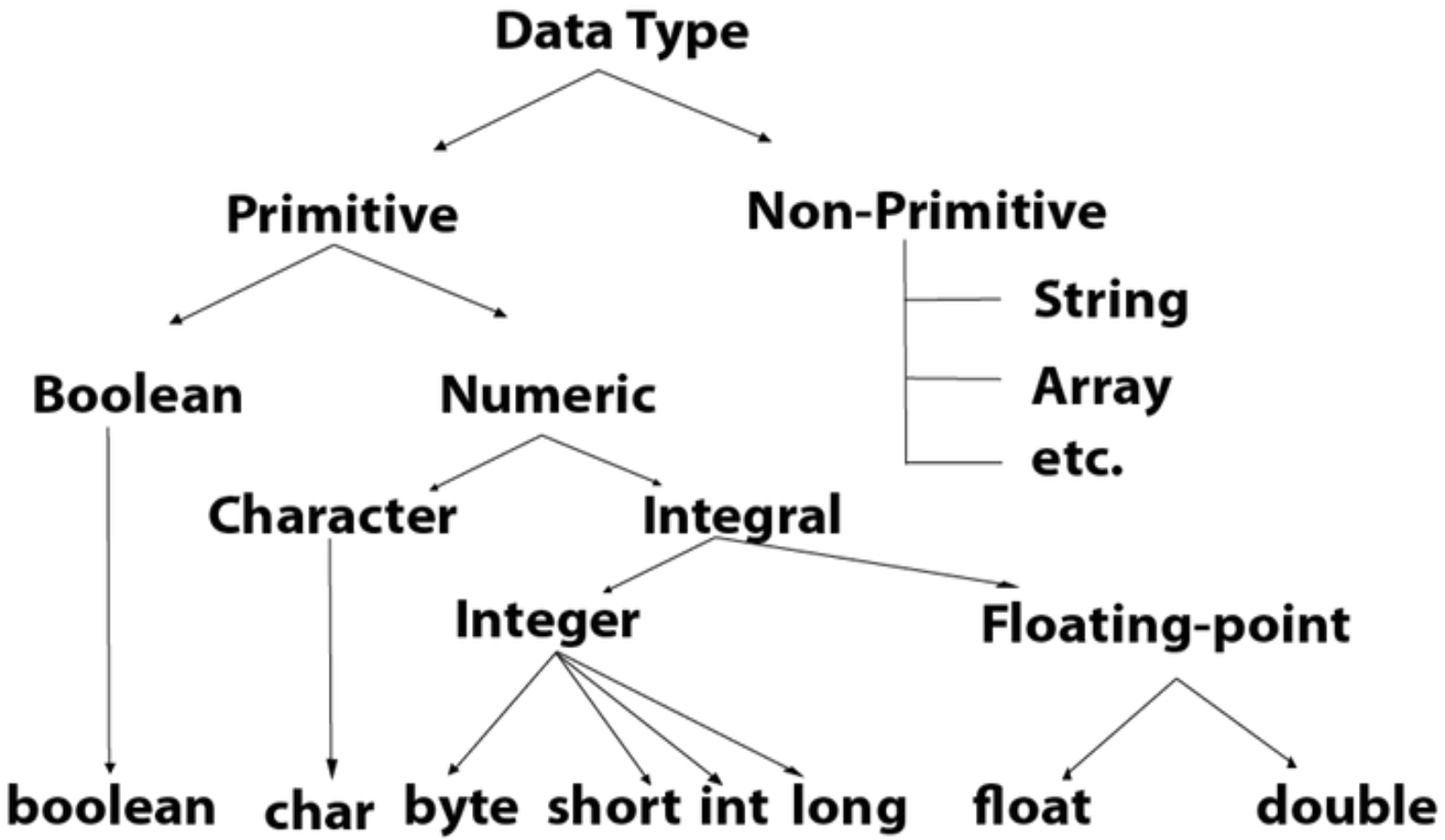
- There are 50 keywords currently defined in the Java language
- These keywords cannot be used as names for a variable, class, or method.

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

- Java reserves the following: **true, false, and null.**
- These are values defined by Java.
- You may not use these words for the names of variables, classes and so on.

Data Types in Java

- Data types specify the different sizes and values that can be stored in the variable.
- There are two types of data types in Java:
- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.
-


Data Type
Primitive
Non-Primitive
String
Array
etc.
Boolean
Numeric
Character
Integral
Integer
Floating-point
boolean
char
byte
short
int
long
float
double

Java Primitive Data 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*
- 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 includes **boolean,** which is a special type for representing true/false values.

Integers

- 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.

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

byte

- The smallest integer type is **byte.**
- This is a signed 8-bit type that has a range from –128 to 127.
- Byte variables are declared by use of the **byte** keyword.
- Example: byte b, c;

Short

- short is a signed 16-bit type.
- It has a range from –32,768 to 32,767.
- It is probably the least-used Java type.
- Example: short s;

int

- The most commonly used integer type is **int.**
- **It is a signed 32-bit type that has a range** from
  –2,147,483,648 to 2,147,483,647.

long

- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value.

Floating-Point Types

- Floating-point numbers, also known as *real numbers*, *are used when evaluating expressions* that require fractional precision.
- There are two kinds of floating-point types, **float** and **double**
- which represent single- and double-precision numbers respectively

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

- The type **float** specifies a *single-precision value that uses 32 bits of storage.*
- Example: float hightemp, lowtemp;

double

- Double precision, as denoted by the **double** keyword, uses 64 bits to store a value.

Characters

- the data type used to store characters is **char.**
- **char** in Java is not the same as char in **C** or **C++.**
- In C/C++, char is 8 bits wide. This is *not the* case in Java.
- 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.
- In Java **char** is a 16-bit type. The range of a char is 0 to 65,536.

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

Literals

Integer Literals:

- **Decimal integer:** base 10 numbers
- **Octal integer: base 8 numbers,** Octal values are denoted in Java by a leading zero.
- Ex:045, 0126
- ***Hexadecimal****:* **base 16 numbers,** *Hexadecimal* values are denoted in Java by a leading zero-x(0x or 0X)
- *Ex: 0x1ab, 0x459f, 0xfff*

- to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long.**
- **You** do this by appending an upper or lowercase *L to the literal.*
- *For example, 0x7fffffffffffffL* or 9223372036854775807L

Floating-Point Literals

- They can be expressed in either standard or scientific notation.
- *Standard notation consists of a whole number* component followed by a decimal point followed by a fractional component.
- For example: 2.0,3.14159, and 0.6667

- *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied.
- The exponent is indicated by an *E or e followed by a* decimal number, which can be positive or negative.
- Examples: 6.022E23, 314159E–05 and 2e+100.
- 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.*

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.

Character Literals

- Characters in Java are indices into the Unicode character set.
- Aliteral 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,

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

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\””“

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.

Declaring a Variable:

- In Java, all variables must be declared before they can be used.
- The basic form of a variable declaration is shown here:
- *type identifier [= value][, identifier [= value] ...] ;*
- **Ex:**
- int a, b, c;
- int d = 3, e, f = 5;
- byte z = 22;
- double pi = 3.14159;

Dynamic Initialization

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

```
int  c=a+b;
```

```
public class PrimitiveDemo {
    public static void main(String[] args) {
     byte b =100;
     short s =123;
     int v = 123543;
     int calc = -9876345;
     long amountVal = 1234567891;
     float intrestRate = 12.25f;
     double sineVal = 12345.234d;
     boolean flag = true;
```

```
boolean val = false;
char ch1 = 88; // code for X
char ch2 = 'Y';
System.out.println("byte Value = "+ b);
System.out.println("short Value = "+ s);
System.out.println("int Value = "+ v);
System.out.println("int second Value = "+calc);
System.out.println("long Value = "+ amountVal);
System.out.println("float Value = "+intrestRate);
System.out.println("double Value = "+ sineVal);
```

```
    System.out.println("boolean Value = "+ flag);
    System.out.println("boolean Value = "+ val);
    System.out.println("char Value = "+ ch1);
    System.out.println("char Value = "+ ch2);
    }
}
```

```
byte Value = 100
short Value = 123
int Value = 123543
int second Value = -9876345
long Value = 1234567891
float Value = 12.25
double Value = 12345.234
boolean Value = true
boolean Value = false
char Value = X
char Value = Y
```

The Scope and Lifetime of Variables

- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- Many other computer languages define two general categories of scopes: global and local.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- The scope defined by a method begins with its opening curly brace.
- However, if that method has parameters, they too are included within the method's scope.
- variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.

- Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope.
- objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true.
- Objects declared within the inner scope will not be visible outside it.

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

- Within a block, variables can be declared at any point, but are valid only after they are declared.

  count = 100; //wrong!

  int count;

- another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left.
- This means that a variable will not hold its value once it has gone out of scope.
- Therefore, variables declared within a method will not hold their values between calls to that method.
- Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope.

- If a variable is initialized, then that variable will be reinitialized each time the block in which it is declared is entered.

```
class LifeTime {
public static void main(String args[]) {
        int x;
        for(x = 0; x < 3; x++) {
            int y = -1;       // y is initialized each time block is entered
            System.out.println("y is: " + y);   // this always prints -1
           y = 100;
          System.out.println("y is now: " + y);
        }
    }
}
```

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

Type Conversion and Casting

- It is common to assign a value of one type to a variable of another type.
- If the two types are compatible, then Java will perform the conversion automatically.
- For example, it is always possible to assign an **int** value to a **long** variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed.
- It is possible to perform an explicit conversion between incompatible types with type casting

Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place iff
- The two types are compatible.
- The destination type is larger than the source type.
- When these two conditions are met, a *widening conversion takes place.*
- there are no automatic conversions from the numeric types to **char** or **boolean.**
- **char** and **boolean** are not compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte, short, long, or char.**

Casting Incompatible Types

- What if you want to assign an **int** value to a **byte** variable?
- This conversion will not be performed automatically, because a **byte** is smaller than an **int.**
- This kind of conversion is sometimes called a *narrowing conversion*,
- To create a conversion between two incompatible types, you must use a cast.
- A *cast is* simply an explicit type conversion.
- general form:

  (*target-type) value*

- int a;
- byte b;
- // ...
- b = (byte) a;
- *truncation* occurs when a floating-point value is assigned to an integer type
- if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

```
class Conversion {
public static void main(String args[]) {
byte b;
int i = 257;
double d = 323.142;
System.out.println("\nConversion of int to byte.");
b = (byte) i;
System.out.println("i and b " + i + " " + b);
System.out.println("\nConversion of double to int.");
i = (int) d;
System.out.println("d and i " + d + " " + i);
```

```
System.out.println("\nConversion of double to
  byte.");
b = (byte) d;
System.out.println("d and b " + d + " " + b);
}
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1

Conversion of double to int.
d and i 323.142 323

Conversion of double to byte.
d and b 323.142 67
```

Automatic Type Promotion in Expressions

- In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand.

  byte a = 40;

  byte b = 50;

  byte c = 100;

  int d = a * b / c;

- The result of the intermediate term **a * b** easily exceeds the range of either of its byte operands.
- To handle this kind of problem, Java automatically promotes each **byte, short** or **char** operand to **int** when evaluating an expression.

- For example, this seemingly correct code causes a problem:

  byte b = 50;

  *b = b * 2; // Error! Cannot assign an int to a byte!*

- The code is attempting to store 50 * 2, a valid **byte** value into a byte variable.
- As the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int.**
- Thus, the result of the expression is now of type **int,** which cannot be assigned to a byte without the use of a cast.
- This is true even if the value being assigned would still fit in the target type.

- we should use an explicit cast to avoid error

byte b = 50;

*b = (byte)(b * 2);*

which yields the correct value of 100.

Type Promotion Rules

- Java defines several *type promotion rules that apply to expressions.*
- *A*ll **byte, short, and char** values are promoted to **int**
- If one operand is a **long,** the whole expression is promoted to **long.**
- If one operand is a **float,** the entire expression is promoted to **float.**
- If any of the operands is **double,** the result is **double.**

```
class Promote {
public static void main(String args[]) {
  byte b = 42;
  char c = 'a';
  short s = 1024;
  int i = 50000;
  float f = 5.67f;
  double d = .1234;
  double result = (f * b) + (i / c) - (d * s);
  System.out.println((f * b) + " + " + (i / c)
  + " - " + (d * s));
  System.out.println("result = " + result);
}
}
```