Introduction to Java

Language Basics

High-level, class-based, object-oriented programming language

History of Java

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called **Green Team.**
- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
- 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.
- **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
- Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.
- Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- JDK 1.0 was released on January 23, 1996.
- Java SE 18 (released at March 2022).

Topics

01

Variables

02

Operators

03

Expressions, Statements and Blocks

04

Control Flow Statements

Variables

Instance Variables (Non-Static Fields)

Objects store their individual states in "non-static fields" (also known as instance variables). Because their values are unique to each instance of a class.

Class Variables (Static Fields)

A class variable means there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

Local Variables

A method will often store its temporary state in local variables. The syntax for declaring a local variable is like declaring a field (for example, int count = 0;)

Parameters

The signature for the main method is public static void main(String[] args). Here, the args variable is the parameter to this method.

Primitive Data Types

Kinds of values that can be stored and manipulated

- ❑ Boolean: Truth value (true or false).
- ❑ Byte
- ❑ Short
- ❑ Char
- ❑ int: Integer (0, 1, -47).
- ❑ Float
- ❑ Long

- ❑ double: Real number (3.14, 1.0, -2.1).
- ❑ String: Text (“hello”, “example”)

Declaring Primitives

- **boolean myBooleanPrimitive;**
- **byte b;**
- **int x, y, z; // declare three int primitives**

```
boolean t = true; // Legal
boolean f = 0; // Compiler error!
```

Literal Values for All Primitive Types:

```
'b' // char literal
42 // int literal
false // boolean literal
2546789.343 // double literal
```

Declaring Primitives

Decimal Literals:

```
int length = 343;
```

Hexadecimal Literals:

```
int y = 0x7fffffff;
int z = 0xDeadCafe;
```

Octal Literals:

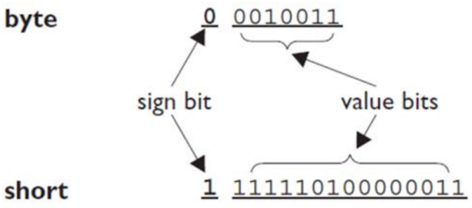
```
int seven = 07; // Equal to decimal 7
int eight = 010; // Equal to decimal 8
int nine = 011; // Equal to decimal 9

int length = 343;
long jo = 110599L;
long so = 0xFFFFl; // Note the lowercase 'l'
```

```
double d = 11301874.9881024;
float g = 49837849.029847F;
float g = 49837849.029847f;
```

Primitive Ranges

All six number types made up of a certain number of 8-bit bytes, and are signed or unsigned, meaning they can be negative or positive.

The leftmost bit (the most significant digit) is used to represent the sign, where a 1 means negative and 0 means positive.



sign bit: 0 = positive
1 = negative

value bits:

byte: 7 bits can represent 2^7 or 128 different values:
0 thru 127 -or- –128 thru –1

short: 15 bits can represent 2^{15} or 32768 values:
0 thru 32767 -or- –32768 thru –1

Ranges of Numeric Primitives:

Data Type	Size(bits)	Range		Default
Keyword		Minimum	Maximum	
boolean	1 bit	false	true	false
byte	8 bits	-128	127	0
short	16 bits	-32,768	32,767	0
char	16 bits	'\u0000' (0)	'\uFFFF' (65535)	'\u0000'
int	32 bits	-2,147,483,648	2,147,483,647	0
long	64 bits	-2^{63}	$2^{63}-1$	0
float	32 bits	32-bit IEEE 754 Floating Point $\sim 1.4e^{-045}$	$\sim 3.4e^{+038}$	0.0
double	64 bits	64-bit IEEE 754 Floating Point $\sim 4.9e^{-324}$	$\sim 1.8e^{+308}$	0.0

Value Assignments

```
char a = 0x892; // hexadecimal literal

char b = 982; // int literal

char c = (char)70000; // The cast is required; 70000 is // out of char range

char d = (char) -98; // Ridiculous, but legal

char e = -29; // Possible loss of precision; needs a cast
```

```
char f = 70000 // Possible loss of precision; needs a cast

char c = '\"'; // A double quote

char d = '\n'; // A newline
```

Basic Mathematical Operators

* / % + - are the mathematical operators
*/ % have a higher precedence than + or –

double myVal = a + b % d – c * d / b;

Is the same as:

double myVal = (a + (b % d)) – ((c * d) / b);

int x = 7; // literal assignment

int y = x + 2; // assignment with an expression
// (including a literal)

int z = x * y; // assignment with an expression

byte b = 27;

byte b = x;

Basic Mathematical Operators

Narrowing requires an explicit cast:

```
byte a = 3; // No problem, 3 fits in a byte
byte b = 8; // No problem, 8 fits in a byte
byte c = b + c; // Should be no problem, sum of the two bytes
// fits in a byte
byte c = (byte) (a + b);
int a = 100;
long b = a; // Implicit cast, an int value always fits in a long
float a = 100.001f;
```

```
int b = (int)a; // Explicit cast, the float could lose info Integer values
double d = 100L; // Implicit cast
int x = 3957.229; // illegal
```

Basic Mathematical Operators

Auto Widening flows:

byte -> short -> int -> long -> double

byte -> short -> int -> float

char -> int -> long -> double

byte -> short -> int -> float -> double

char -> int -> float -> double

Default Values

Variable Type	Default Value
Object reference	`null` (not referencing any object)
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

Arrays

•An array is a list of similar things

•An array has a fixed:

–name

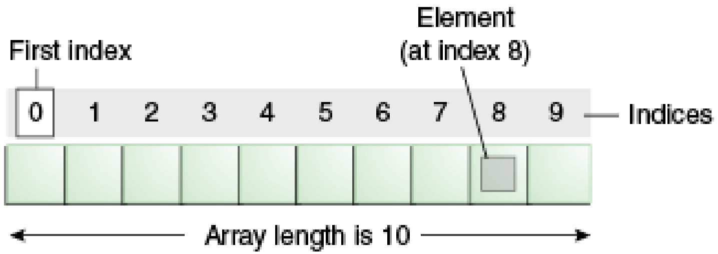
–type

–length

•These must be declared when the array is created.

•Arrays sizes cannot be changed during the execution of the code

•Arrays always be an object on the heap



Constants

In Java constants are variables whose values, once assigned, cannot be changed. You declare a constant by using the keyword final. Here are examples of constants or final variables.

```
final int ROW_COUNT = 50;
final boolean ALLOW_USER_ACCESS = true;
```

Declaring Arrays

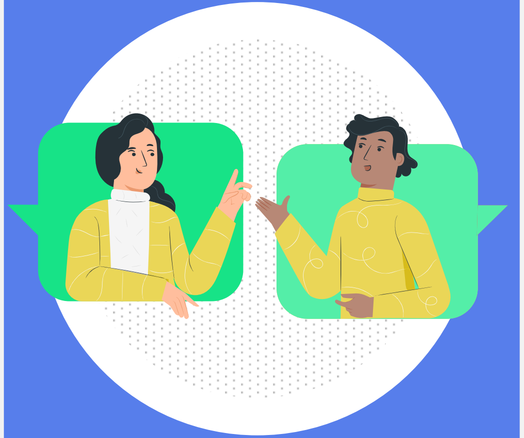
int[] key; // brackets before name (recommended)

int key []; // brackets after name (legal but less readable)

// spaces between the name and [] legal, but bad

int myArray[]; (Declaration)

int[5] scores; // not okey..



Construction: *myArray* to be an array of integers

myArray = new int[8];
(sets up 8 integer-sized spaces in memory, labelled *myArray[0]* to *myArray[7]*)

int myArray[] = new int[8];
combines the two statements in one line

Assigning Values

You can refer to the array elements by index to store values in them.

```
myArray[0] = 3;
myArray[1] = 6;
myArray[2] = 3;
```

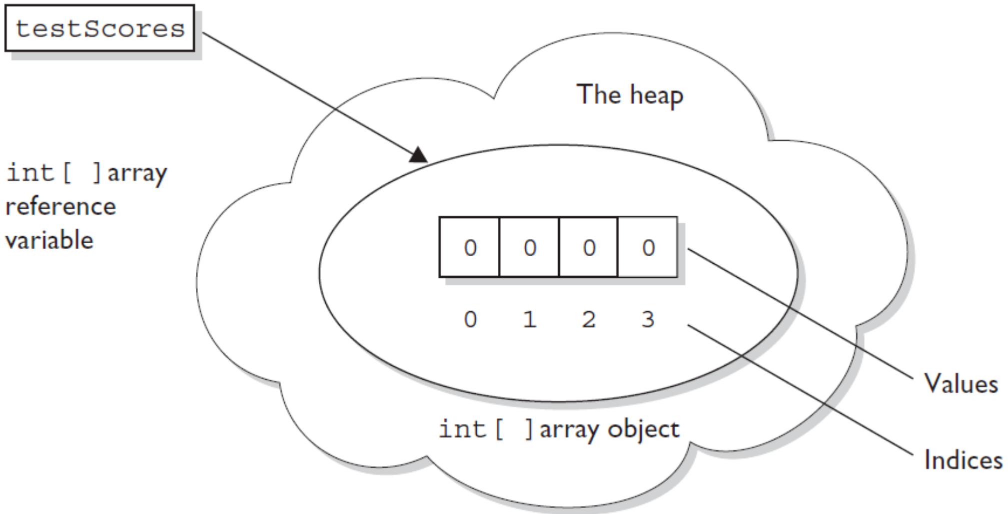
You can create and initialise in one step:

```
int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};
myArray = {3, 6, 3, 1, 6, 3, 4, 1}; // not okey
myArray = new int[] {3, 6, 3, 1, 6, 3, 4, 1}; // okey
int[] carList = new int[]; // Will not compile; needs a size
int[] carList = new int[] {3, 6, 3, 1, 6, 3, 4, 1} ; // what u say?
```

Assigning Values

int[] testScores; // Declares the array of ints

testScores = new int[4]; // constructs an array and assigns it
// to the testScores variable

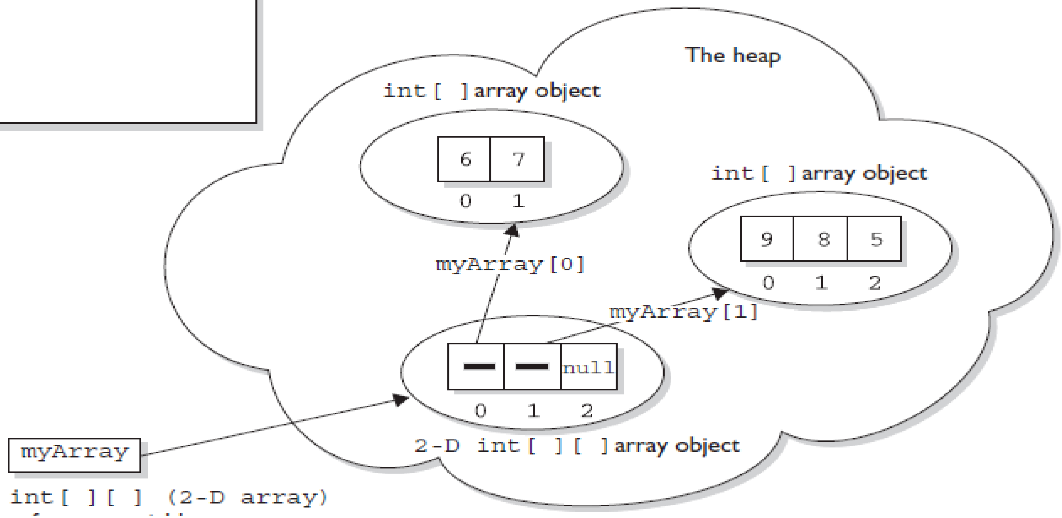


BJIT Academy EST 2014

Assigning Values

Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```



Iterating Through Arrays

for loops are useful when dealing with arrays

```
for (int i = 0; i < myArray.length; i++)
{
  myArray[i] = getsomevalue();
}
```

Statements & Blocks

A simple statement is a command terminated by a semi-colon

```
name = “Fred”;
```

A block is a compound statement enclosed in curly brackets:

```
{
name1 = “Fred”; name2 = “Bill”;
}
```

Blocks may contain other blocks

Flow of Control

- Java executes one statement after the other in the order they are written
- Many Java statements are flow control statements:

Alternation:	if, if else, switch
Looping:	for, while do while
Escapes:	break, continue, return

If – The Conditional Statement

The if statement evaluates an expression and if that evaluation is true then the specified action is taken.

```
if ( x < 10 ) x = 10;
If the value of x is less than 10, make x equal to 10
It could have been written:
if ( x < 10 )
x = 10;

Or alternatively:
if ( x < 10 ) { x = 10; }
```

Relational Operators

==

Equal (careful)

!=

Not equal

>=

Greater than or equal

<=

Less than or equal

>

Greater than

<

Less than

If… else

The if … else statement evaluates an expression and performs one action if that evaluation is true or a different action if it is false.

```
if (x != oldx) {
        System.out.print(“x was changed”);
}
else {
        System.out.print(“x is unchanged”);
}
```

Nested if … else

```
if ( myVal > 100 )

{
        if ( remainderOn == true)
        {
                myVal = mVal % 100;
        }

        else
        {
                myVal = myVal / 100.0;
        }
}
else
{
        System.out.print(“myVal is in range”);
}
```

else if

Useful for choosing between alternatives:

```
if ( n == 1 ) {
        // execute code block #1
}
else if ( j == 2 ) {
        // execute code block #2
}
else {
        // if all previous tests have failed, execute code
block #3
}
```

A Warning...

WRONG!

```
if( i == j )
if ( j == k )
System.out.print ("i equals k");
else
System.out.print ("i is not equal to j");
```

CORRECT!

```
if( i == j ) {
if ( j == k )
System.out.print ("i equals k");
}
else
System.out.print ("i is not equal to j");// Correct!
```

The switch Statement

```
switch ( n ) {
        case 1:
                // execute code block #1
                break;
        case 2:
                // execute code block #2
                break;

        default:    // if all previous tests fail then
                //execute this code block
                break;
}
```

The for loop

In this example, we have an array of integers, and we want to find the sum of all the elements in the array. We use a for loop to iterate through each element in the array and add it to a running sum variable.

```java
int[] numbers = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < numbers.length; i++) {
    sum += numbers[i];
}
System.out.println("Sum of the numbers: " + sum);
```

Foreach Loop

```java
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");
for (String name : names) {
    System.out.println("Hello, " + name + "!");
}
```

In this example, we have a list of strings, and we want to print a greeting message for each name in the list. We use a foreach loop to iterate through each element in the list and print a message using that element.

while loops

```
while(response == 1) {
        System.out.print( “ID =” + userID[n]);
        n++;
        response = readInt( “Enter “);
}
```

- What is the minimum number of times the loop is executed?
- What is the maximum number of times?

do {… } while loops

```
do {
        System.out.print( “ID =” + userID[n] );
        n++;
        response = readInt( “Enter ” );
} while (response == 1);
```

- What is the minimum number of times the loop is executed?
- What is the maximum number of times?

Break

A break statement causes an exit from the innermost containing while, do, for or switch statement.

```
for ( int i = 0; i < maxID, i++ ) {
        if ( userID[i] == targetID ) {
                index = i;
                break;
        }
}       // program jumps here after break
```

Continue

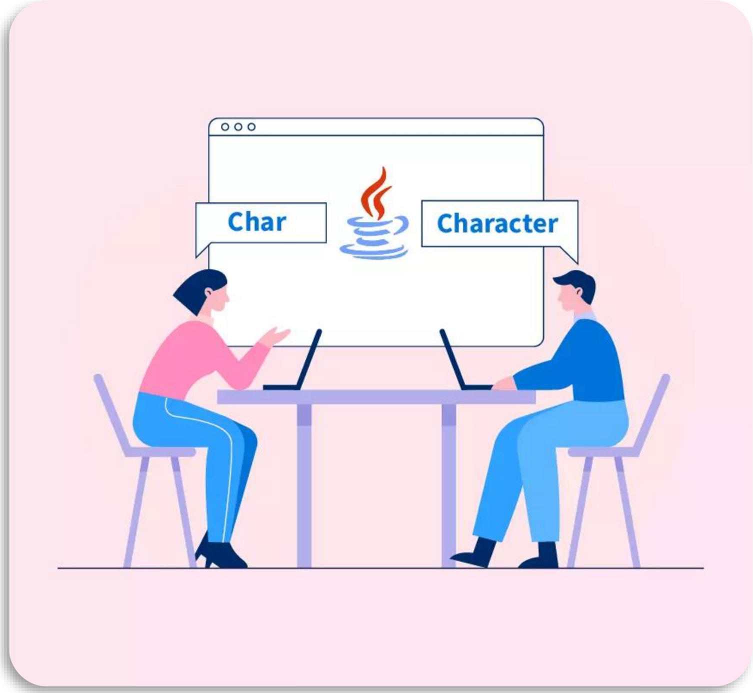
Can only be used with while, do or for. The continue statement causes the innermost loop to start the next iteration immediately

```
for ( int i = 0; i < maxID; i++ ) {
        if ( userID[i] != -1 ) continue;
        System.out.print( “UserID ” + i + “ :” +   userID);
}
```

Wrapper Classes in Java

In Java, a wrapper class is a class that provides an object representation of a primitive data type.

Wrapper classes are used to convert primitive data types into objects, which allows them to be used in situations where objects are required, such as in collections and generic classes.



Wrapper Classes in Java

Here are the wrapper classes for the eight primitive data types in Java:

1. Byte: Represents a byte value (-128 to 127)
2. Short: Represents a short value (-32,768 to 32,767)
3. Integer: Represents an integer value (-2^31 to 2^31-1)
4. Long: Represents a long value (-2^63 to 2^63-1)
5. Float: Represents a floating-point value (approximately ±1.4E-45 to ±3.4E+38)
6. Double: Represents a double value (approximately ±4.9E-324 to ±1.8E+308)
7. Boolean: Represents a Boolean value (true or false)
8. Character: Represents a character value (Unicode character set)

How to use the Integer Wrapper class

Wrapper classes are often used when working with collections, because collections can only store objects, not primitive types.

For example, if you want to store a list of integers in a collection, you can use the Integer wrapper class to convert the primitive int type into an object.

```
int i = 10;
Integer integerObject = Integer.valueOf(i); // convert int to Integer object
int j = integerObject.intValue(); // convert Integer object back to int
System.out.println(j); // prints 10
```

Here's an example of how to use the Integer wrapper class:

Example of Autoboxing and Unboxing

In Java 5 and later, autoboxing and unboxing feature was introduced which allows automatic conversion between primitive types and their corresponding wrapper classes.

This means you can write code that uses primitive types, and the Java compiler will automatically convert them to their corresponding wrapper classes as needed.

```
int i = 10;
Integer integerObject = i; // autoboxing: convert int to Integer object
int j = integerObject; // unboxing: convert Integer object back to int
System.out.println(j); // prints 10
```

Here's an example of autoboxing and unboxing:

The use of Wrapper Classes in Java: Complex Examples

Here are a few complex examples that demonstrate the use of wrapper classes in Java:

1. Converting Strings to Integers using the Integer Wrapper Class:

```
String str = "123";
int i = Integer.parseInt(str); // convert string to int
System.out.println(i); // prints 123

Integer integerObject = Integer.valueOf(str); // convert string to Integer object
System.out.println(integerObject); // prints 123
```

> *In this example, we have a string "123" and we want to convert it to an integer. We use the Integer wrapper class to do this conversion in two ways: first by using the parseInt method, which returns an int, and second by using the valueOf method, which returns an Integer object.*

The use of Wrapper Classes in Java: Complex Examples

In this example, we have a list of Integer objects, and we use them with generics.

We add some integers to the list and then use a foreach loop to iterate through each element and add them to a running sum variable.

When we access the Integer objects in the list, they are automatically unboxed to their primitive int values.

```java
List<Integer> numbers = new ArrayList<>();
numbers.add(1);
numbers.add(2);
numbers.add(3);

int sum = 0;
for (Integer num : numbers) {
    sum += num; // auto-unboxing: convert Integer object to int
}
System.out.println("Sum of the numbers: " + sum);

Collections.sort(numbers); // use Integer's compareTo method to sort the list
System.out.println("Sorted numbers: " + numbers);
```

We also use the Collections.sort method to sort the list, which works because the Integer class implements the Comparable interface and provides a compareTo method.

The use of Wrapper Classes in Java: Complex Examples

3. Converting Bytes to Strings using the Byte Wrapper Class:

In this example, we have an array of bytes, and we want to convert it to a string. We use the String constructor that takes a byte array to do this conversion.

```
byte[] bytes = {72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100};
String str = new String(bytes); // convert bytes to string
System.out.println(str); // prints "Hello World"

StringBuilder sb = new StringBuilder();
for (byte b : bytes) {
    sb.append(Byte.toString(b)).append(" "); // convert byte to string and add to 
}
String byteStr = sb.toString().trim();
System.out.println(byteStr); // prints "72 101 108 108 111 32 87 111 114 108 100"
```

We also use the Byte wrapper class to convert each byte to a string and add it to a StringBuilder. We then convert the StringBuilder to a string and print it, which gives us a space-separated list of byte values.

Generics in JAVA

Generics in Java is a feature that allows classes, interfaces, and methods to be parameterized with one or more types.

In other words, it allows you to define a class or method that can work with different types of data, without the need to create a separate version of the class or method for each data type.

The syntax for declaring a generic type is to use angle brackets (<>) and a placeholder name (usually a single uppercase letter) to represent the type parameter.

Generics in JAVA

For example, the following code declares a generic class called Box that can hold any type of object:

```
public class Box<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```

In this example, the type parameter `T` is used as a placeholder for the actual type that will be used when the class is instantiated.

Generics in JAVA

For example, to create a Box that holds strings, we can use the following code:

```java
Box<String> box = new Box<>();
box.setValue("Hello");
String value = box.getValue();
```

In this code, the <String> part after Box indicates that we are using the Box class with the String type parameter.

Using generics in Java provides several benefits, such as improved type safety, code reusability, and reduced code duplication. It also makes code easier to read and understand, as it provides a clear indication of what types are expected and returned by a given class or method.

Thank You