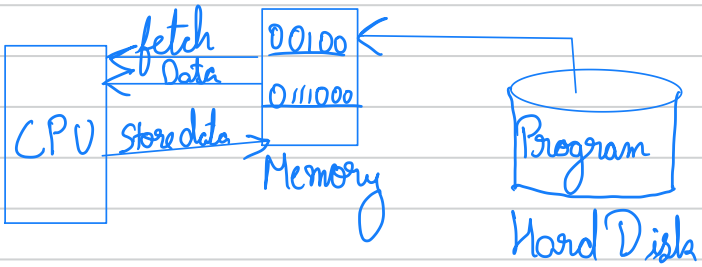
Java

CPU's fetch and execute cycle (Not Imp but should know)



How Java code is executed

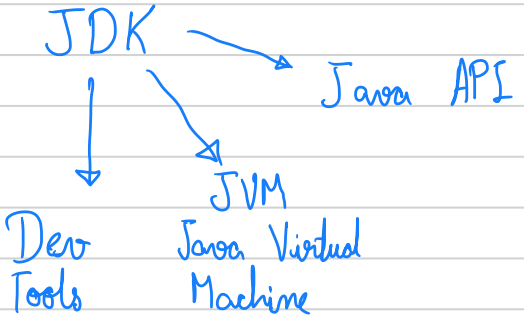
Source Code
↓
Java Compiler
↓
Java Byte Code
↓
Java Interpreter (JVM → Java Virtual Machine)
↓
Results

Eg:-

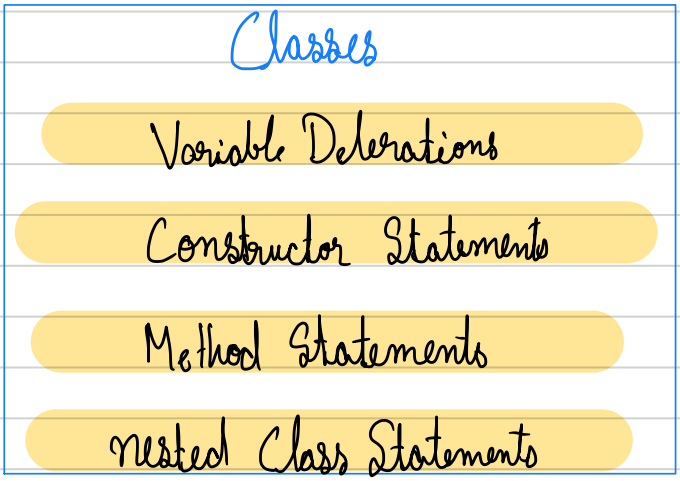
Hello.java
javac Hello.java
↓
compilation
↓
Java Bytecode Hello.class

java Hello
JVM windows
JVM linux
JVM macOS

What Does JDK include

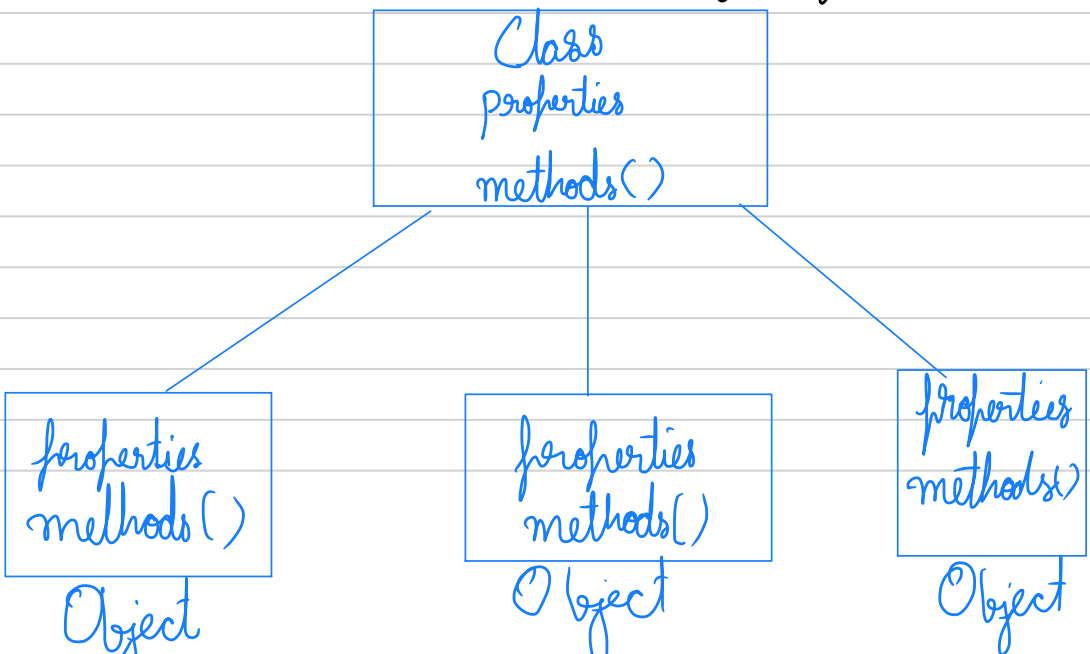


Java Program Structure



Classes and Objects

→ A Class is a blueprint for creating objects



→ An Objects can get methods and properties from a class while at the same time having their own properties and methods.

→
```
class Student {
    int id;
    String name;
    String gender;

    boolean updateProfile (String newName){
        name = newName;
        return true;
    }
}
```

Variable declaration

method defenation

Creating object using class

→
```
class StudentTest {

    public static void main (String [ ] args)
        Student s = new Student ( );

        s.id = 1000;
        s.name = "Aryan";
        s.gender = "male";

        s.updateProfile ("John");
    }
}
```

Data type is class name for creating object from class

keyword

Class name

Creating new student object

Assigning Values

Calling methods

→ Naming methods / variables / classes

→ Names given to different parts of program are known as Identifier
→ Identifiers can include letters, underscore, digits and $.
→ Identifiers cannot begin with a number.
→ It cannot be a reserved keyword of java.
→ Identifiers are case sensitive

→ Printing in Java

→ System.out.println(value) → advances the cursor to next line
→ System.out.print(value) → Cursor stays at the end

System.out.println() → Prints blank line
System.out.print("") → Prints blank

→ Java is a Statically Typed Language language

→ Declaring Variable

<type> <name> [= literal or expression];

Types can be both

Primitive
int etc
boolean

Non-Primitive
Class name etc
String

Passing value or expression is known as initialization and it is optional in a class

→ Reinitializing Variable

<name> = literal or expression;

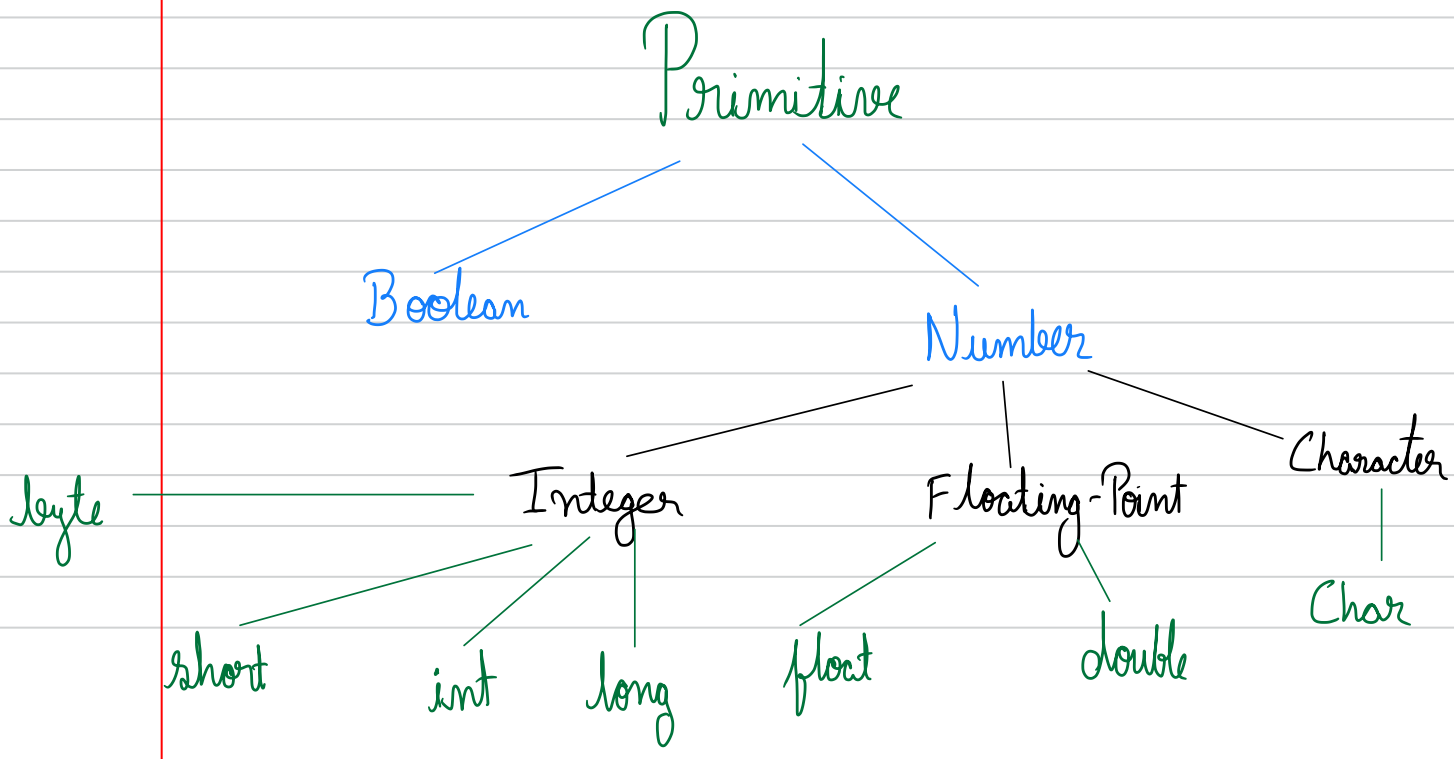
Reinitialization cannot take place at class-level.

It can only take place within methods in a class

Local Variables that are initialized within a method do not get Default Values.

Only Global Variables that are initialized within a class get Default Values

→ Primitive Data types



→ Integers

→ Whole or fixed-point numbers, eg :- 65, -100

→ byte, short, int, long → prefer int

Type	Bit depth	Value range	Default
byte	8 bits (8-1=7)	-2^7 to $2^7 - 1$	0
short	16 bits (16-1=15)	-2^{15} to $2^{15} - 1$	0
int	32 bits (32-1=31)	-2^{31} to $2^{31} - 1$	0
long	64 bits (64-1=63)	-2^{63} to $2^{63} - 1$	0

1 bit in (left most bit) each is known as sign bit and it tells weather a number is -ve or +ve it is also known as Most Significant Bit (MSB)

→ if sign bit is 0 → +ve number
if sign bit is 1 → -ve number

→ Floating Point Numbers

→ float, double → prefer double

Type	Bit depth	Value range	Default	Precision
float	32 bits	-3.4E38 to 3.4E38	0.0f	6-7 decimal digits
double	64 bits	-1.7E308 to 1.7E308	0.0d	15-16 decimal digits

→ Character

→ Single letter characters, eg :- 'A', '0', '$' } Single Quotes only

→ Internally character is stored as an Integer ~ 16-bit unsigned Integer

↓

UTF-16 ⇒ hexadecimal

→ Boolean

→ default value is false

→ Local, Static & Instance Variables

→ Instance & Static Variables

- Declared within Class
- Scope : Entire Class
- Gets Default Value
- Cannot be redefined directly within block (can be redefined through methods)

→ Local Variables

- Declared within a method
- Scope : Local
- Doesnot get default value
- Can be redefined directly within the method

→ Instance : Represents object state → Belongs to each object

→ Values are unique to object

→ From outside class can be accessed via object reference

→ Unique to each object

→ Static :

→ Values are unique to class ~ shared across objects → Belongs to Class

→ From outside class can be accessed via Class name and also within method

→ Some copy shared among all objects

→ Type Casting

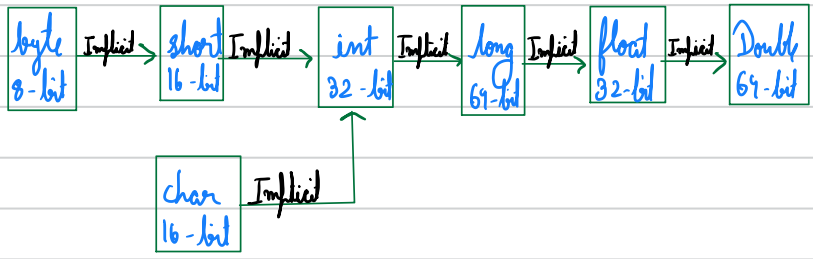
→ Assigning a variable of one type to variable of another type

→ Only numeric-to-numeric casting is possible

→ Implicit Casting ⇒ Done by Compiler

→ Possible is smaller datatype is assigned to larger

Eg:- `int x = 65;`
`float y = x;` Implicit casting by compiler



→ Explicit Casting

→ Larger datatype to smaller datatype

→ Order is opposite of Implicit

→ Conversion to and from char to short and char to byte is Explicit.

Eg:- `long y = 42L;`
`int x = (int)y;`

→ Object References ⇒ Un-Primitive data-type

Assigns student object's address

`Student s = new Student();`

Allocates space for reference

Allocates space for new Student object

→ All objects are stored on heap

↑

area of memory

→ default value is null

→ Statement types

→ Declaration Statements

Eg :- int count = 25;

→ Expression statements

Eg :- → count = 35; // asignment

→ getCount(); // method invocation

→ count++; // increment

→ Control flow statements

Eg :- if (x<50) {

}

these statements cannot be executed at class level

→ Arrays

→ It is an object

→ In Java the number of elements an array can store is fixed

→ It stores values of a single type

→ There are 3 ways to initialize an Array

1. `int[] scores = new int[4];`

   - `int[]` → Data type of value that will be stored
   - `new int[4]` → number of elements in array

   now to add values (cannot be added at class level)

```
scores[0] = 90;
scores[1] = 100;
scores[2] = 80;
scores[3] = 70;
```

   can be used for long arrays

2. `int[] scores = new int[] {90, 100, 80, 70}`

   - `int[]` → data type of elements
   - `{90, 100, 80, 70}` → elements

3. `int[] scores = {90, 100, 80, 70}`

   - `int[]` → data type of elements
   - `{90, 100, 80, 70}` → elements

   used for short arrays

→ How to reinitialize an array created within a class from a method.

1.
```
public class Main {

    static int[] scores;

    public static void main(String[] args) {

        scores = new int[4];
        now to add values

        scores[0] = 90;
        scores[1] = 100;
        scores[2] = 80;
        scores[3] = 70;
}
```

2.
```
public class Main {
    static int[] scores;
    public static void main(String[] args) {
        scores = new int[4]{90,100,80,70};
    }
```

→ To get length of array

array.length

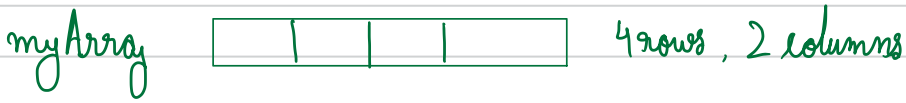
→ You can also create an array of a class and assign object references to it

```
class Student {
    int id;
    String name;
    String gender;
    boolean updateProfile(String newName){
        name = newName;
        return true;
    }
}

class array {
    public static void main(String[] args){
        Student[] students = new student[3];

        students[0] = new Student();
        students[1] = new Student();
        students[2] = new Student();
}
```

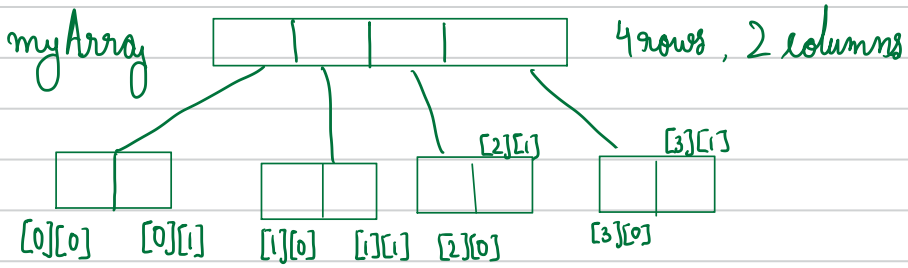
A Class in different file but same directory can be accessed from another file within the directory

→ 2D Arrays

→ JVM creates an array for each row



→ Each element then references to an array with 2 values of the columns.



→ Creating 2D arrays

1. `int[][] myArray = new int [4][2];`

row column

```
myArray[0][0] = 9;
myArray[0][1] = 11;
```

2. `int [][] myArray = new int[][] {`

```
{9,11},
{2,5},
{4,4},
{6,13}
};
```

Rows

3. `int[][] myArray = {`

```
{9,11},
{2,5},
{4,4},
{6,13}
};
```

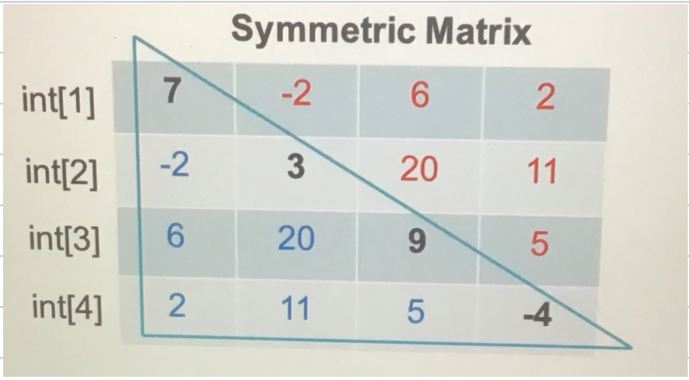
→ 2D Array with Irregular Rows ⇒ column not fixed

Eg:-

```
int[][] myArray = new int[2][];
myArray[0] = new int[5];
myArray[1] = new int[2];
```

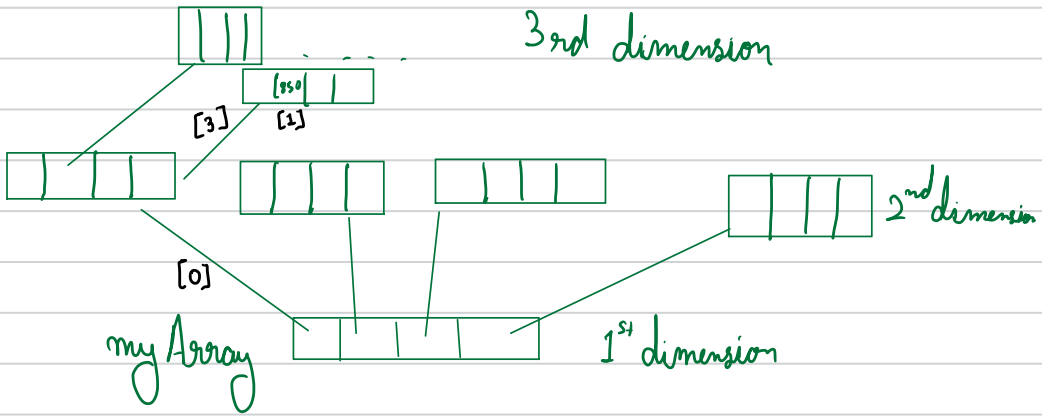
→ columns not fixed

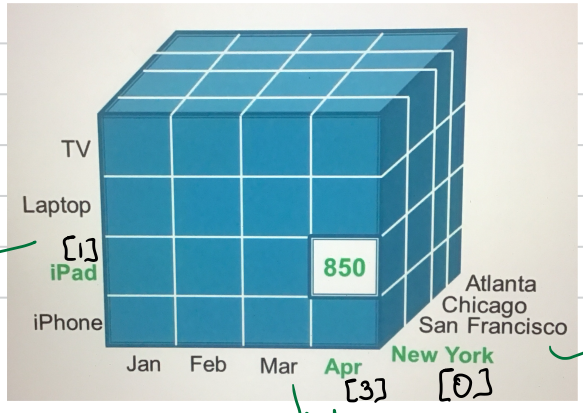
⇒ Useful for Symmetric matrix



→ 3D Arrays ⇒ rarely used

→ Syntax is same just add 1 more box for 3rd dimension

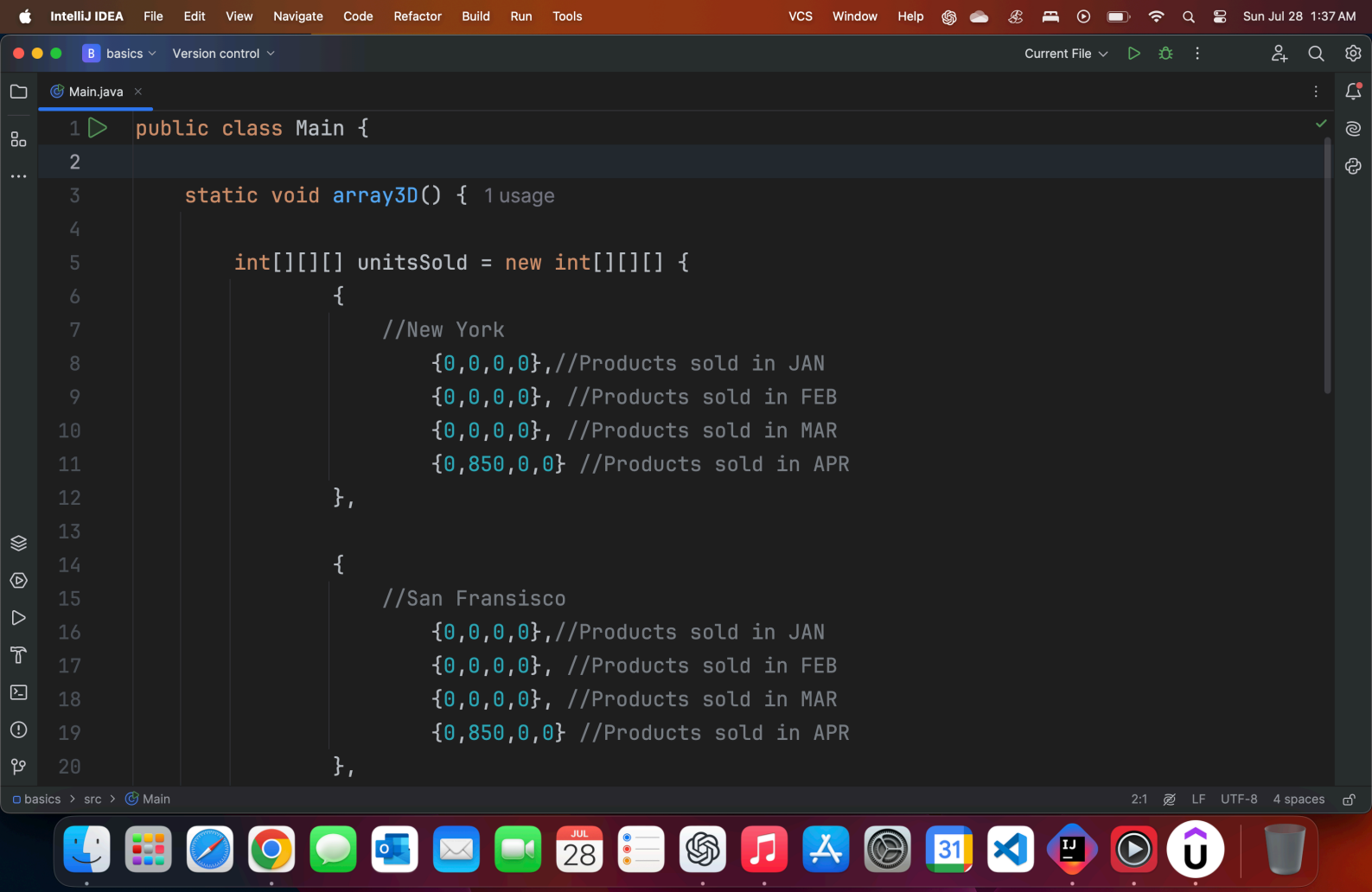




3rd dimension

1st dimension

2nd dimension

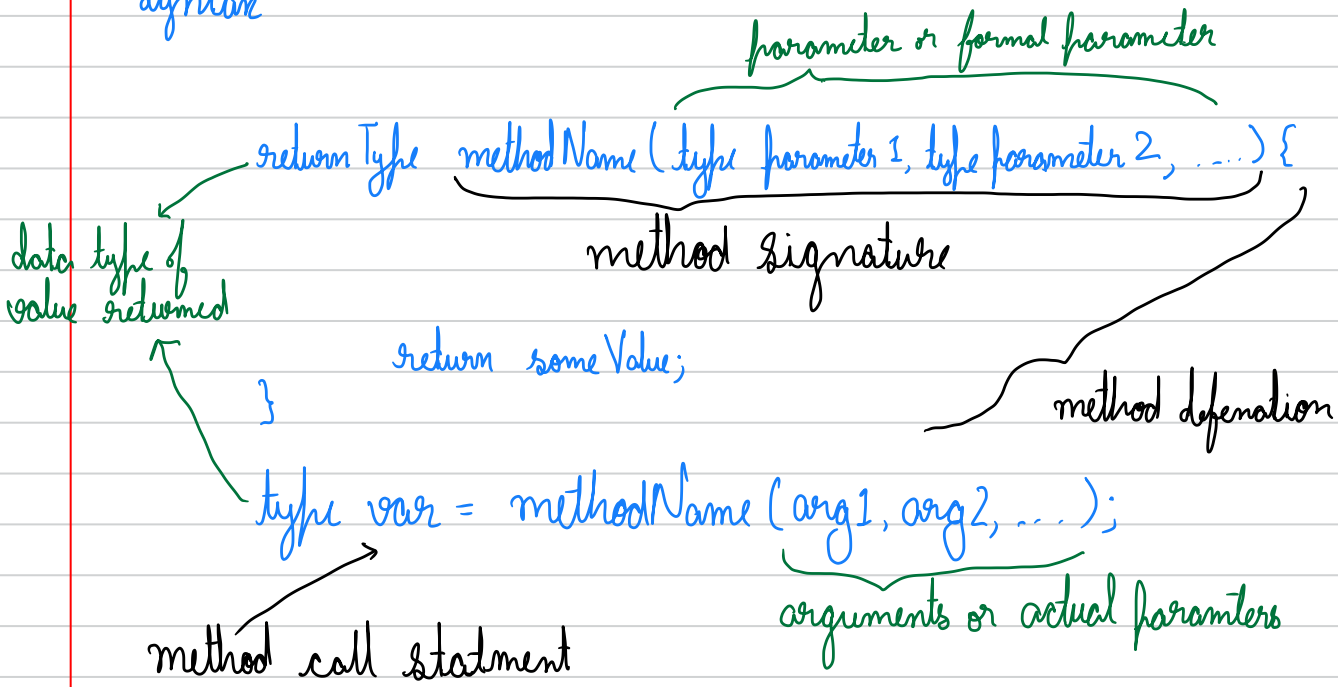

IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools VCS Window Help Sun Jul 28 1:37 AM
basics Version control Current File
Main.java
public class Main {
static void array3D() { 1 usage
int[][][] unitsSold = new int[][][] {
{
//New York
{0,0,0,0},//Products sold in JAN
{0,0,0,0}, //Products sold in FEB
{0,0,0,0}, //Products sold in MAR
{0,850,0,0} //Products sold in APR
},
{
//San Fransisco
{0,0,0,0},//Products sold in JAN
{0,0,0,0}, //Products sold in FEB
{0,0,0,0}, //Products sold in MAR
{0,850,0,0} //Products sold in APR
},
basics > src > Main
2:1 LF UTF-8 4 spaces

```
                {

                        {0,0,0,0},//Products sold in JAN
                        {0,0,0,0}, //Products sold in FEB
                        {0,0,0,0}, //Products sold in MAR
                        {0,850,0,0} //Products sold in APR
                },

                {
                        {0,0,0,0},//Products sold in JAN
                        {0,0,0,0}, //Products sold in FEB
                        {0,0,0,0}, //Products sold in MAR
                        {0,850,0,0} //Products sold in APR
                }
        };

        System.out.println(unitsSold[0][3][1]);
    }
```

```
40
41      public static void main(String[] args) {
42          array3D();
43      }
44
45  }
```

→ Methods

→ Syntax



⇒ For void method call statement

```
methodName ( arg1, arg2, ... );
```

→ type and order of method parameters and arguments should be same

→ Methods cannot be called at class-level

→ Data type of method and method call statement should be same

⇒ returnType

→ must be a primative type or array, or class or interface or void

→ void

→ nothing to return

→ Other than void all methods must return

→ Type of Methods

→ Instance Method

→ They are object level methods

→ Invocation: objectreference.methodName();

→ Can access anything from an instance method. (static and instance variables or methods)

→ Static Methods

→ keyword static in declaration

→ They are class level methods

→ Invocation: className.methodName()

→ main method is a static method.

→ Cannot directly access instance variables/methods. Can access via object reference.

→ Can directly access static variables/methods.

→ Data changed within methods

→ Updating primatives within methods

```
void updateValue(int newId) {
        newId = 1001;
}
int id = 1000;
updateValue(id);
System.out.println(id); ⇒ prints 1000
```

→ Updating value using object reference

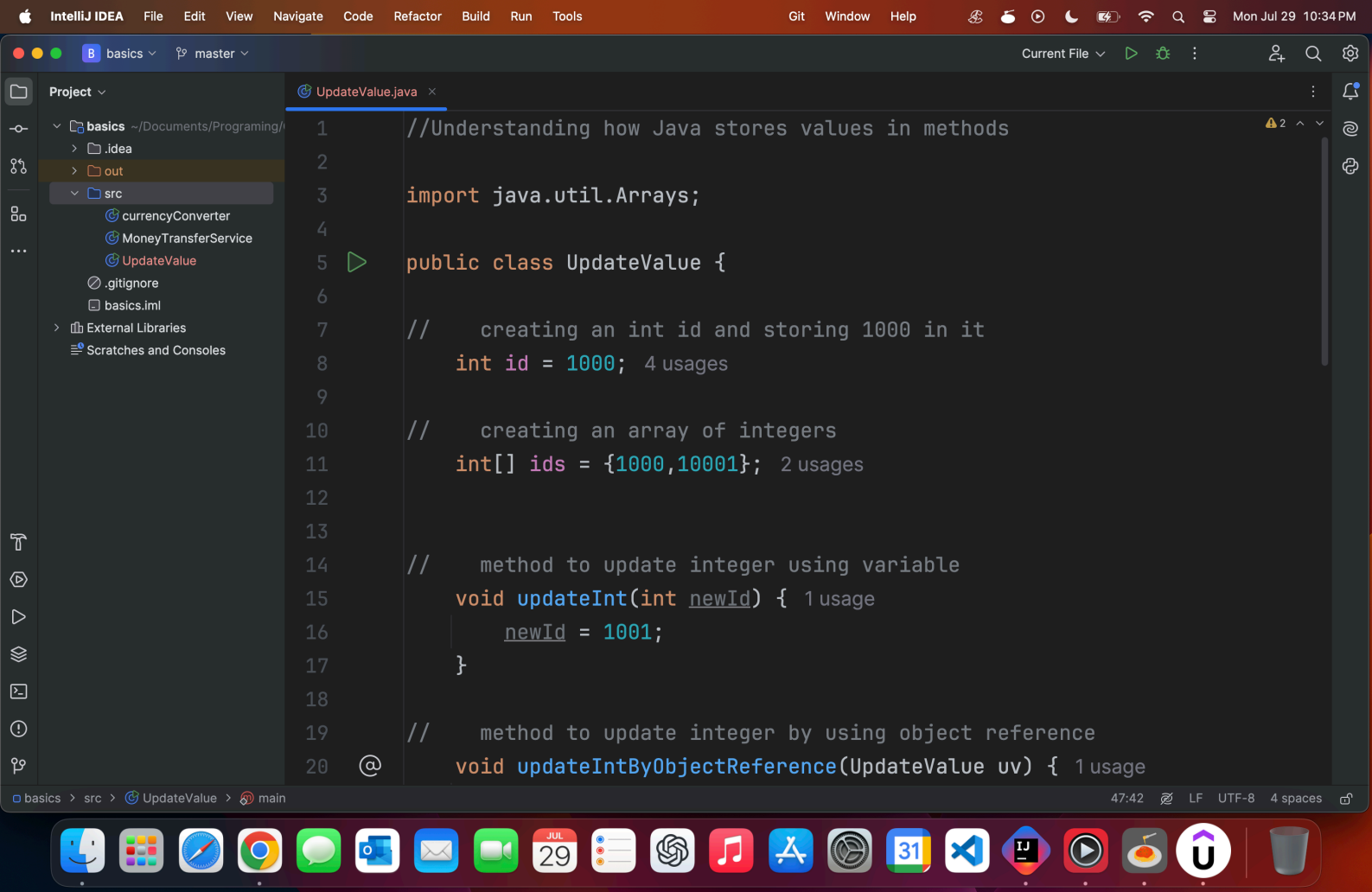
```
class Student {

    int id;

    void updateValue(Student s1){
        s1.id = 1001;
    }

Student s = new Student();
s.id = 1000
System.out.println(s.id); ⇒ prints 1001
```

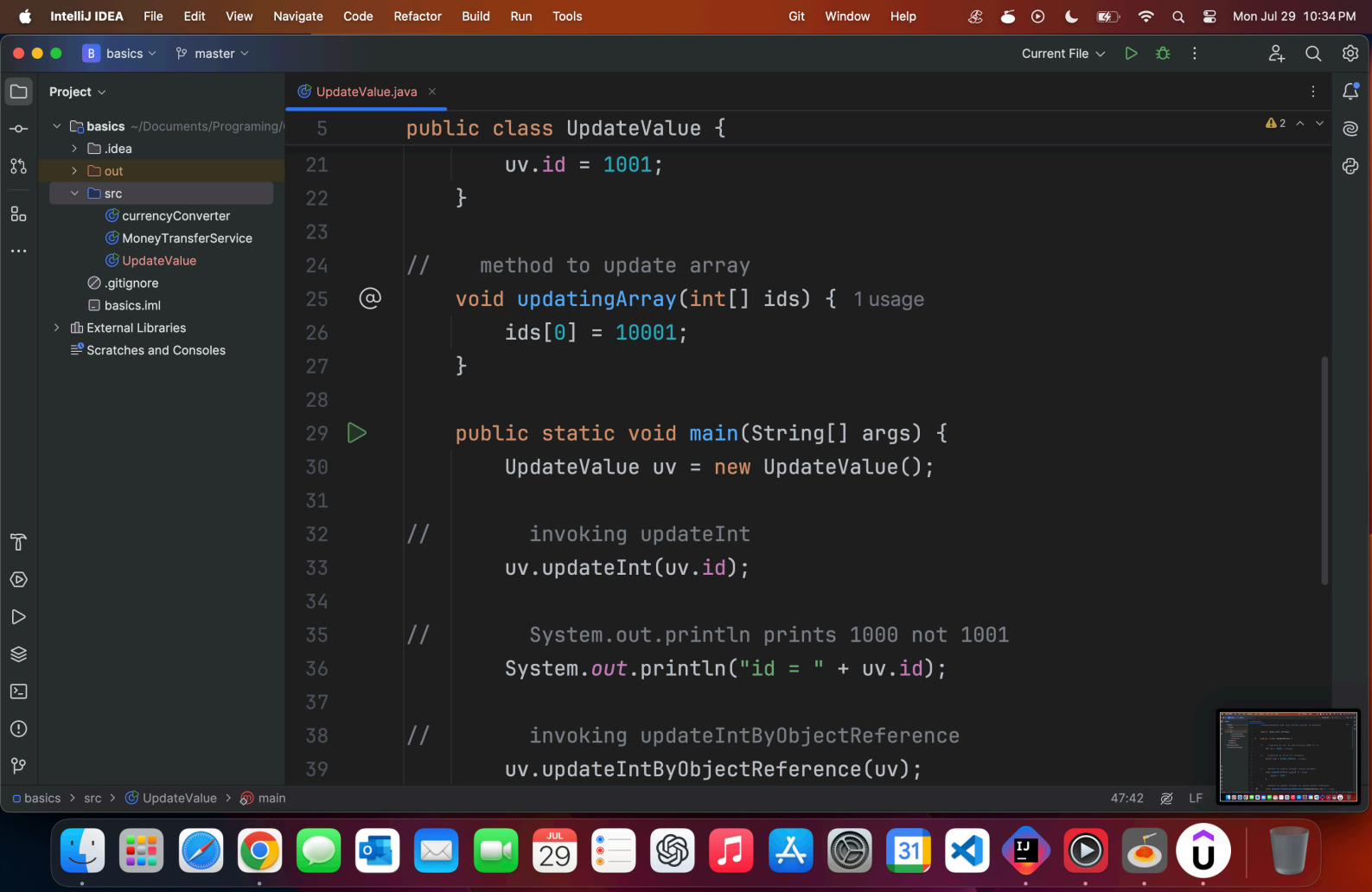
→ when a primitive value is passed in a method java creates a new local variable with the copy of the value of primitive passed as argument, the local variable is then discarded. Thus, the value of primitive does not update

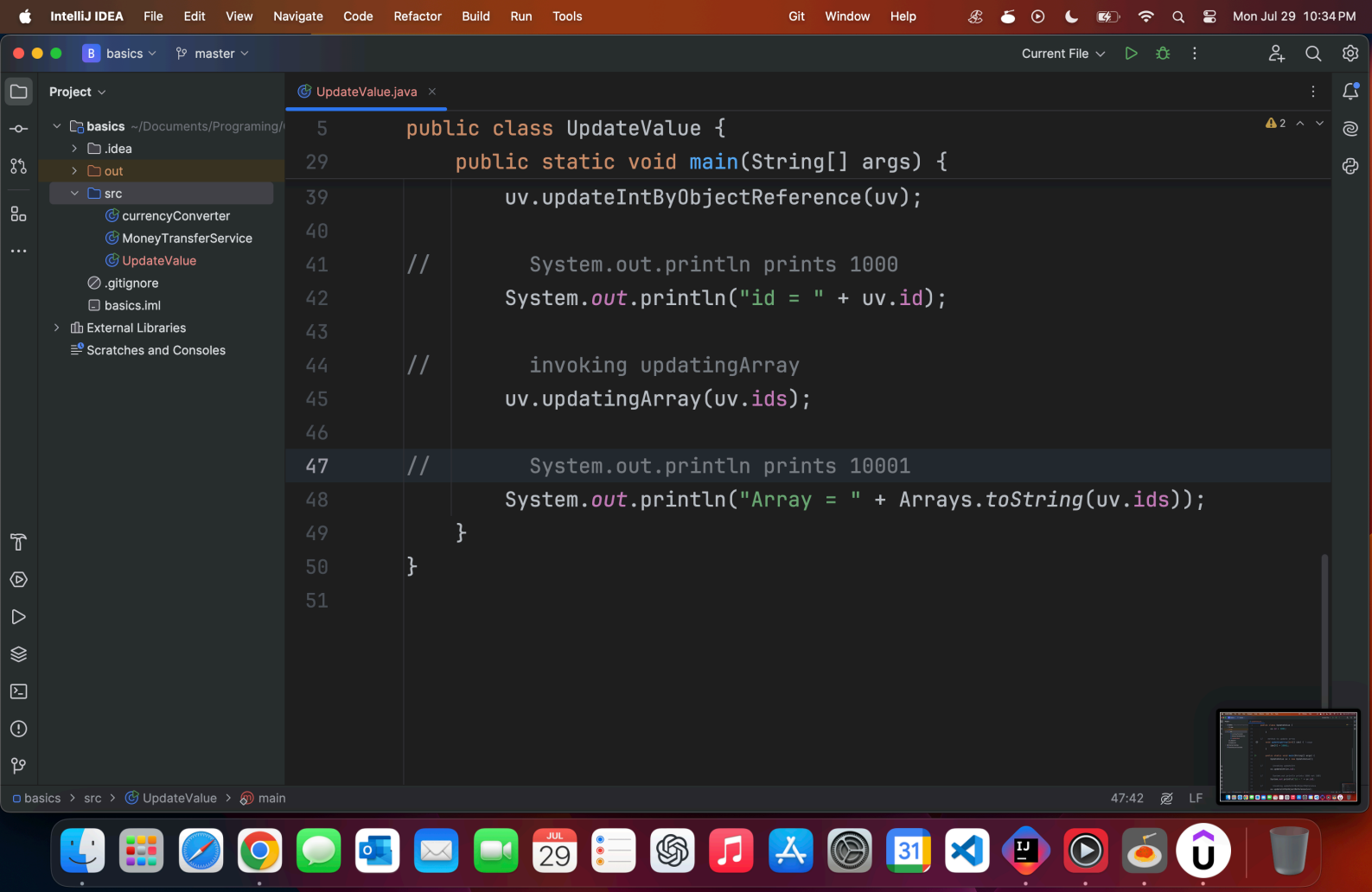
→ when an object reference is passed in a method it stores the memory address of the original object passed as argument. Thus, the value is updated.

→ Java is always pass by value i.e. method parameter stores a copy of the value passed in as argument.

Primitives always pass copy of the value
Object reference pass the copy of memory address


IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help Mon Jul 29 10:34 PM
basics master Current File
Project
basics ~/Documents/Programing/
.idea
out
src
currencyConverter
MoneyTransferService
UpdateValue
.gitignore
basics.iml
External Libraries
Scratches and Consoles
UpdateValue.java
//Understanding how Java stores values in methods
import java.util.Arrays;
public class UpdateValue {
// creating an int id and storing 1000 in it
int id = 1000; 4 usages
// creating an array of integers
int[] ids = {1000,10001}; 2 usages
// method to update integer using variable
void updateInt(int newId) { 1 usage
newId = 1001;
}
// method to update integer by using object reference
void updateIntByObjectReference(UpdateValue uv) { 1 usage
basics src UpdateValue main
47:42 LF UTF-8 4 spaces


IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help Mon Jul 29 10:34 PM
basics master Current File
Project
basics ~/Documents/Programing/
.idea
out
src
currencyConverter
MoneyTransferService
UpdateValue
.gitignore
basics.iml
External Libraries
Scratches and Consoles
UpdateValue.java
public class UpdateValue {
uv.id = 1001;
}
// method to update array
void updatingArray(int[] ids) { 1 usage
ids[0] = 10001;
}
public static void main(String[] args) {
UpdateValue uv = new UpdateValue();
// invoking updateInt
uv.updateInt(uv.id);
// System.out.println prints 1000 not 1001
System.out.println("id = " + uv.id);
// invoking updateIntByObjectReference
uv.updateIntByObjectReference(uv);
basics > src > UpdateValue > main
47:42 LF


IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help Mon Jul 29 10:34 PM
basics master Current File
Project
basics ~/Documents/Programing/
.idea
out
src
currencyConverter
MoneyTransferService
UpdateValue
.gitignore
basics.iml
External Libraries
Scratches and Consoles
UpdateValue.java
public class UpdateValue {
public static void main(String[] args) {
uv.updateIntByObjectReference(uv);
// System.out.println prints 1000
System.out.println("id = " + uv.id);
// invoking updatingArray
uv.updatingArray(uv.ids);
// System.out.println prints 10001
System.out.println("Array = " + Arrays.toString(uv.ids));
}
}
basics src UpdateValue main
47:42 LF

→ Method Overloading

→ In java multiple methods within the same class can have same name, but parameter list is different.

→ To overload a method we must change parameter list.

→ JVM selects the most appropriate method according to the passed argument

→ parameters, parameter types or order of parameters must change.

→ Changing only return type or making a method static doesn't work or vice versa making static instance.

→ Method overloading applies to both static and instance methods

→ Valid Examples:-

1) `void updateProfile(int newId){}`

2) `void updateProfile(int newId, char gender){}`

3) `void updateProfile(char gender, int newId){}`

4) `void updateProfile(short newId){}`

→ Invalid Examples:-

1) `boolean void updateProfile(int newId){}`

2) `void updateProfile(int Id){}`

3) `static void updateProfile(int newId){}`

→ Varargs Parameter in methods

→ varargs means variable length argument

→ It can take variable number of arguments

→ If the method has more than one parameter It has to the last parameter

→ It can be the only parameter too

→ Syntax

→ Three dots following data type.

Eg :- 1) go(int num, String name, int... items) {}

2) go(int... nums) {}

→ Invocation

→ An array or Comma sperated value can be passed to varargs.

Eg :- 1) go (new int[] {1, 2, 3})

2) go (1, 2, 3)

→ Compiler automatically converts varargs to array

→ varargs can be ommitted too.

Eg :- go ()

→ A method cannot have more than one varargs

→ A varargs parameter is simply an array but provides simpler and flexible invocation

→ Invalid varargs overload :-

```
foo (boolean, int... items)
foo (boolean, int[] items)
```

Both are same as varargs is an array too

→ In case of method overloading the method with varargs will be matched at last

→ Constructors

→ Constructors are used to initialize object state

→ Syntax

```
className (type parameter 1, type param2 ....) {
}
```

can have varargs

→ Example :-

```
class Student {
        int id;
        Student (int newId) {
          id = newId;
        }
}

public static void main (String[] args) {
        Student s = new Student (1001);
}
```

→ If class does not include constructor compiler inserts one with no parameters. This constructor is also known as no-args constructor

→ Constructors cannot return a value

→ You can invoke methods from within constructors

→ There can be multiple constructors ⇒ Constructor Overloading

→ Constructor Overloading & this invocation (this())

→ It is same as method overloading

→ methods can be created using any constructor

→ To reduce hassle of changing each constructor for small tasks we can create a primary constructor with all basic requirements and create overloaded constructor for specific task and invoke primary constructor using this();

→ this() has to be the first statement

↓

calls the constructor in case of overloaded matches the arguments and that constructor is called

→ a constructor can have only one this()

→ a this statement within a constructor cannot call the constructor in which it causes recursive invocation

↓

repeated invocation

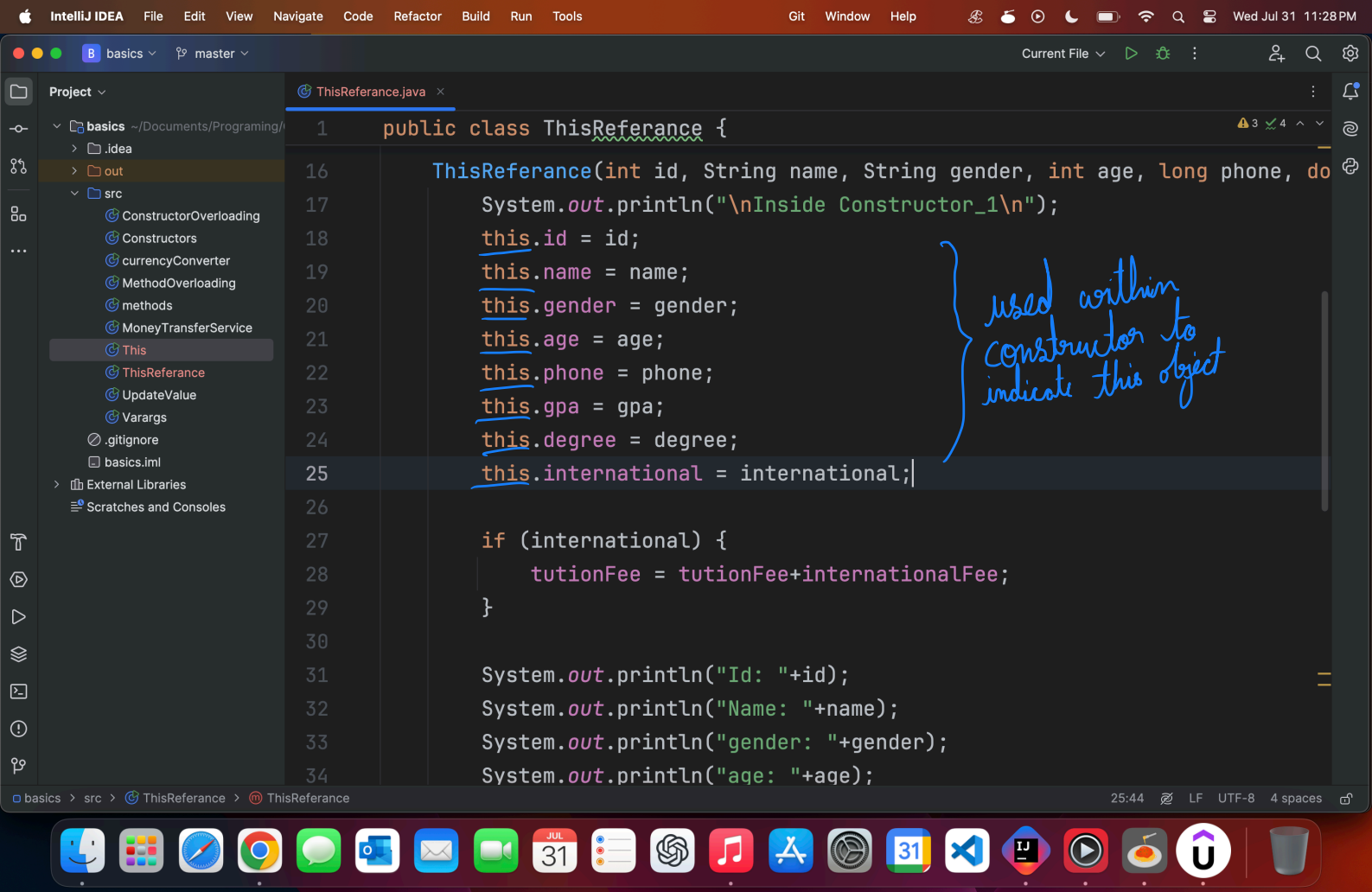
→ this statement cannot cause recursive invocation

→ this statement cannot have instance variables

→ this reference (this.something)

→ A this reference can be used inside constructors or an method to access a shadowed variable in this object.

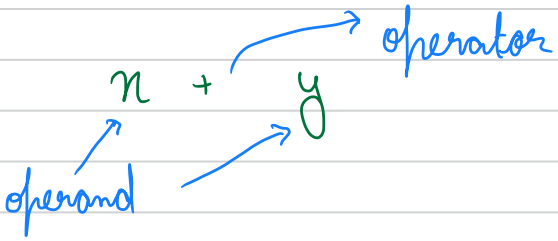
In a method or constructor when the parameter and variable have the same name the parameter shadows or hides the instance variable as the parameter is a local variable


IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools Git Window Help Wed Jul 31 11:28 PM
basics master Current File
Project
basics ~/Documents/Programing/
.idea
out
src
ConstructorOverloading
Constructors
currencyConverter
MethodOverloading
methods
MoneyTransferService
This
ThisReferance
UpdateValue
Varargs
.gitignore
basics.iml
External Libraries
Scratches and Consoles
ThisReferance.java
1 public class ThisReferance {
16 ThisReferance(int id, String name, String gender, int age, long phone, do
17 System.out.println("\nInside Constructor_1\n");
18 this.id = id;
19 this.name = name;
20 this.gender = gender;
21 this.age = age;
22 this.phone = phone;
23 this.gpa = gpa;
24 this.degree = degree;
25 this.international = international;
26
27 if (international) {
28 tutionFee = tutionFee+internationalFee;
29 }
30
31 System.out.println("Id: "+id);
32 System.out.println("Name: "+name);
33 System.out.println("gender: "+gender);
34 System.out.println("age: "+age);
used within constructor to indicate this object
basics > src > ThisReferance > ThisReferance
25:44 LF UTF-8 4 spaces

→ You cannot use this inside static methods as we cannot access instance variable inside static methods

→ Operators

→ Operators are symbols that perform operation on its operand and produce a result



→ There are three types of operators based on the number of operands.

1) Unary operator
   - → One operand
   - → Eg :- x++ ⇒ postincrement

2) Binary Operator
   - → Two operands
   - → Eg :- x + 3

3) Ternary operator (?:)
   - → Three operator
   - → Eg :- x > 3 ? x : 0 → executed if condition is false

Condition — Executed if condition is true

→ Arithmetic Operators

→ Addition (+)

→ int i = 5 + 2;

→ Unary plus: int i = +5;

→ String Concatination

→ Subtraction (-)

→ int i = 5 - 2;

→ Unary minus: int i = -5;

→ Multiplication (*)

→ int i = 5 * 2;

→ Division (/)

→ int i = 5/2;

→ Modulus (%)

→ int i = 5 % 2;

→ Arithmatic Operators apply only to primative numeric types

→ Pre and Post increment or decrement

→ applies to + and - only

→ ++ or --

Post: x ++

Pre: ++ x

→ Compound Arithmetic Assignment Operators

→ +=, -=, %=, /=, *=

→ Order of Precedence of Arithmetic Operators

1) Parantheses are evaluated first

2) Unary operators and increment and decrement

3) Multiplication

4) Division

5) Modulus

6) Addition

7) Substraction

→ Arithmetic Operation Rules

→ Types smaller than int are promoted to int before operation is carried.

Eg:- 127(byte) + 1(byte) → 127(int) + 1(int) → 128(int)

→ If both operands are int, float or double, then operations are carried in the same type and result will also be of that type.

Eg:- 5(int) + 6(int) → 11(int)

1/2 = 0 (not 0.5 as both 1 and 2 are int)

→ If operands belong to diff types, the smaller type is promoted to larger type.

Order of Promotion: int → long → float → double

Eg:- 1/2.0 or 2.0/1 → 1.0/2.0 → 0.5

→ Comparison Operators

→ They return true or false

- → Equal to (==)
- → not Equal to (!=)
- → Greater than (>)
- → less than (<)
- → Greater than Equal to (>=)
- → less than Equal to (<=)

→ Logical Operators

- → AND (&&)
- → OR (||)
- → NOT (!)

Short-Circuit Operators

→ AND (&&) → if left operand is false, returns false

→ OR (||) → if left is true, returns true

↓

If either of these case are true java doesn't check the other condition

→ due to && being a Short-Circuit Operator it helps in preventing NullPointerException

↓

When we try to access a variable of a null object reference. It stops program execution

→ Logical Operators Precedence Order

1) NOT (!)

2) AND (&&)

3) OR (||)

NOT (!) > Arithmetic Operators > Comparison Operators > AND (&&) > OR (||)

5000, 2, F, T, T, F, F, F

→ Bitwise Operator ⇒ Imp in DSA (Hash tables)

→ Works with individual bits of its operands

(1) Among bitwise operators, only &, |, and ^ can be used with boolean operands. **Bitwise NOT (~) will not even compile with boolean.** If we need such a behavior, then we would use **logical NOT (!)** operator, which we discussed in Logical Operators lecture.

(2) Also, strictly speaking *to be consistent with the Java Language Specification (JLS)*, the operators &, | and ^ when applied on ***boolean*** operands are referred to as **logical operators** and not bitwise. In other words, the operators &, |, ^, ~ are referred to as *bitwise* **ONLY** when they are applied on **integer operands** and this is the common scenario as we discussed. Also keep in mind that the logical operators && and || have the short-circuit property due to which JLS refers to them as **conditional AND** and **condition OR** respectively. With & and |, as discussed in the lecture, they do not have the short-circuit property, i.e., they always force JVM to evaluate both operands.

→ Bitwise AND (&)

→ Returns 1 if both input bits are 1

→ Bitwise OR (|)

→ Returns 1 if one of the input bits are 1

→ Bitwise XOR (^)

→ Returns 1 only one of the input bits is 1

→ Bitwise NOT (~)

→ Inverts bits

→ Bitwise assignment

→ &=

Eg:- boolean b = true;
b&= false;

→ |=

→ ^=

→ Bitshift Operators ⇒ Imp in DSA (Hash tables)

→ Works with individual bits of Operands

→ Shifts bits

→ Left-shift (<<)

→ Left shifts bits by # bits specified on right

▸ Example

6 → 00000000 00000000 00000000 00000110
6 << 1 → 00000000 00000000 00000000 00001100 → 12

→ Inserts zeroes at lower-order bits

→ Same as multiplying by powers of 2

Eg:- $6 << 1 \rightarrow 6 * 2^1 \rightarrow 12$
$6 << 2 \rightarrow 6 * 2^2 \rightarrow 24$

→ Unsigned Right-Shift (>>>)

→ right shifts bits by # bits specified on right

→ Inserts zeroes at higer order bits

▸ Example

12 → 00000000 00000000 00000000 00001100
12 >>> 1 → 00000000 00000000 00000000 00000110 → 6

→ Same as division by powers of 2

Eg:- $12 >>> 1 \rightarrow 12/2^1 \rightarrow 6$
$12 >>> 2 \rightarrow 12/2^2 \rightarrow 3$

→ Signed Right Shift (>>)

→ Same as >>>, but padded with MSB

due to which sign is preserved

▸ Example

-2,147,483,552 → 10000000 00000000 00000000 01100000
-2,147,483,552 >> 4 → 11111000 00000000 00000000 00000110
(-134,217,722)

→ Control Flow Statements

→ They do not require ; at the end of the block.

→ The statements inside control flow block require ;

→ If the statement is of one line {} is optional

↓

but always use {}

→ If there are statements after return in a block they become unreachable

→ Code in Github with Explaination

→ Switch block restrictions

Eg:-

```
switch (month) {                    → selector expression
    case 1 :
        ....
        break;
    case 2 :
        ....
        break;
    case 3 :                        → case labels
        ....
        break;
    :
}
```

→ The selector expression can be an int, byte, short, char, string, enum or any object reference.

→ Value of Case label should be known at compile time

→ The case label should be in range of data type of selector expression

→ Case label should be unique

→ Arrow Labels