ET0736 Object Oriented Programming and Data Structure

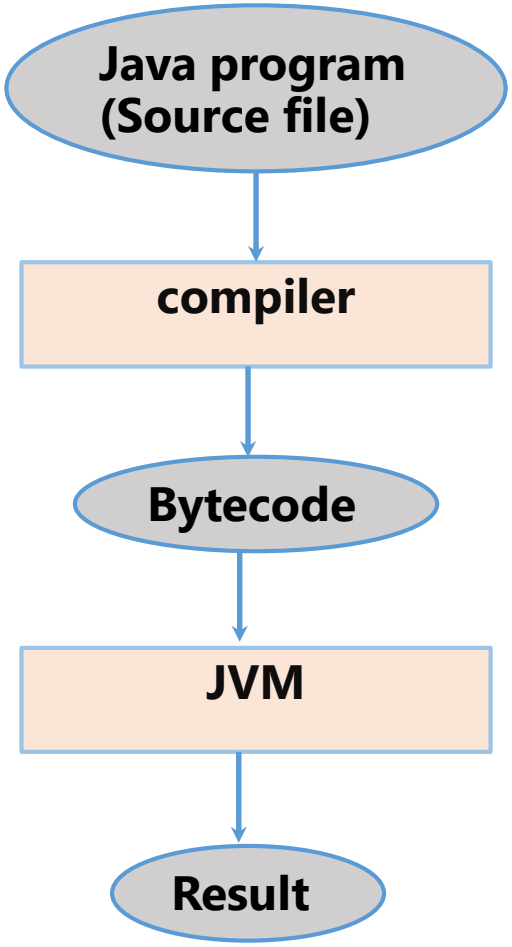
Lesson 1

Introduction to Java programming, object Oriented Programming, Data Structure and Algorithm and Elementary programming in Java

Topics

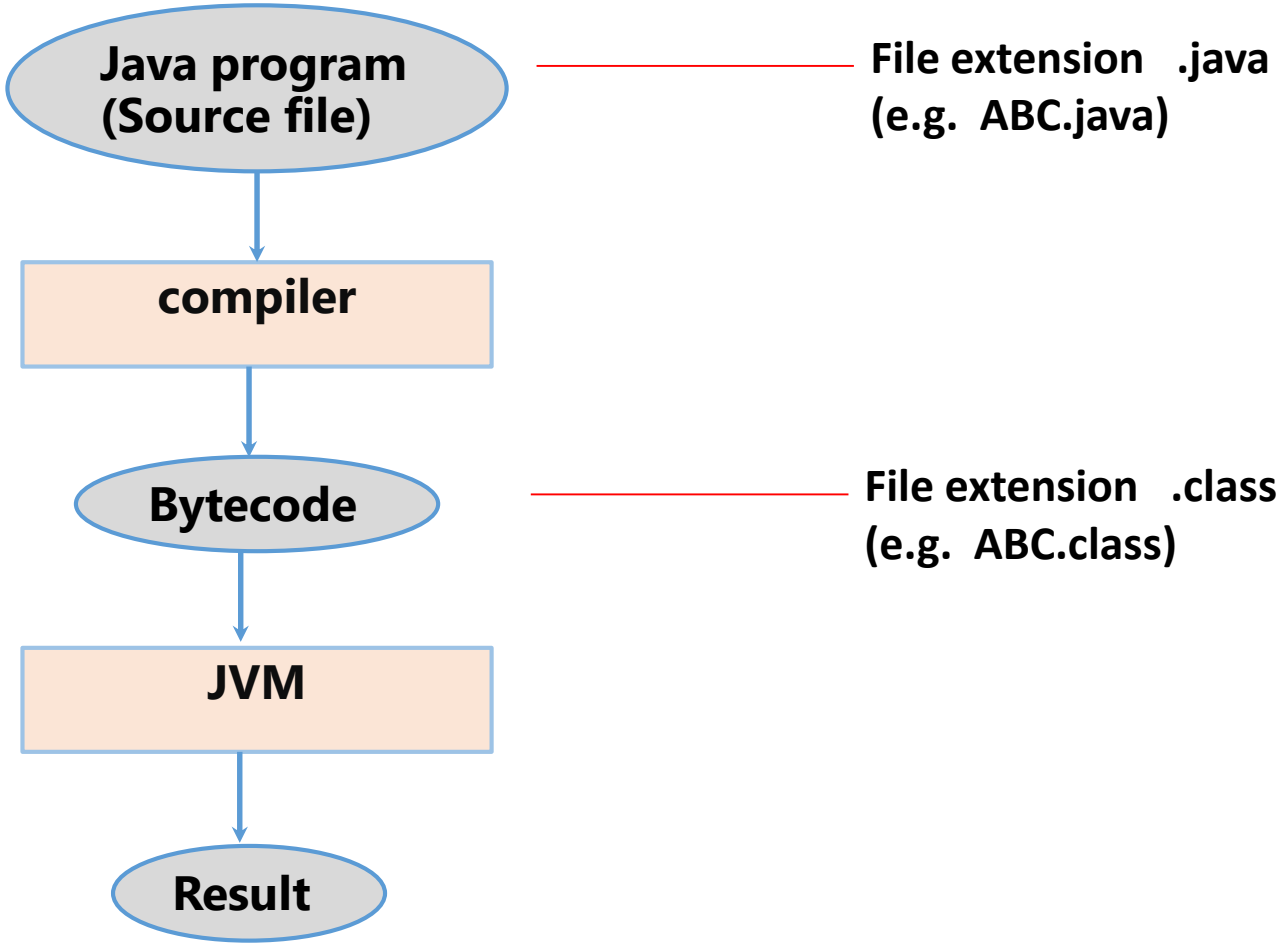
- Java Terminology
- JDK, JRE and JVM
- What is OOP
- Why Data Structure
  What is algorithm
- Identifiers, variables and constants
- Primitive data types
- Variable and constant
- Display mixed text and variable
- Operators
- Character type and String
- Selection (if-else/switch-case)
- Repetition (loop)
- Methods

Terminology

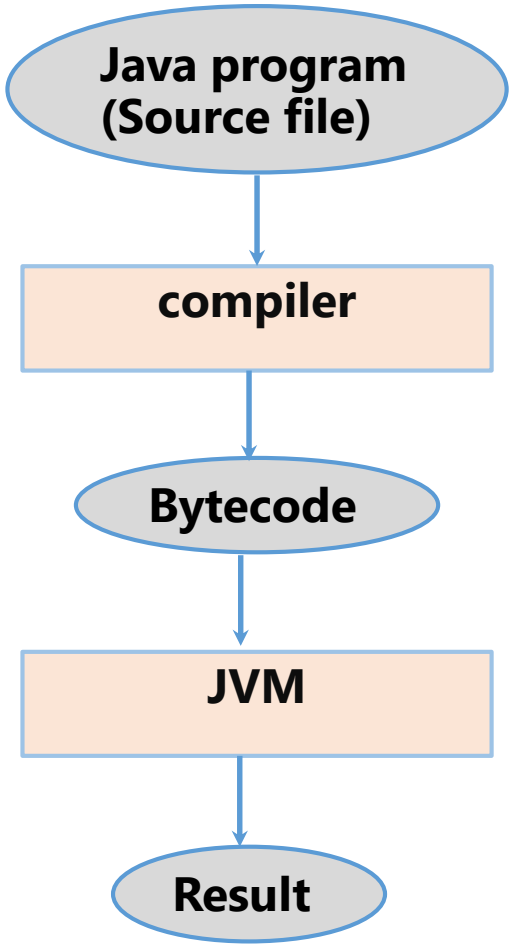


- **Java program can be written using text editor (e.g. NotePad) or Integrated Development Environment (IDE) (e.g. IntelliJ, Eclipse)**
- **Compiler compiles the source code program into standard Bytecode (independent of platform)**
- **Java Virtual Machine (JVM) runs the Bytecode (JVM is platform-dependent)**

Java Terminology



Java Terminology



Need to download IDE (e.g. Eclipse)

Need to download Java Development Kits (JDK) which includes compiler and JRE

JDK or JRE?

- JRE (Java Runtime) is a physically present installation that provides the environment to only run the Java programs.
- JDK (Java Development Kit) is also physically present, which includes JRE plus the development tools (such as compiler and debugger), is needed for *writing* Java programs.
- Java Virtual Machine (JVM) is part of JRE, which is an abstract machine that is responsible for
  - Loading codes
  - Verifies Codes
  - Linking codes
  - Executing codes
- To compile and debug Java Programs, you will need to install JDK

Which IDE?

- ***IDE*** (Integrated Development Environment) is a program environment that allows developers to edit, debug, compile, run and manage their programs.
- Popular IDEs for Java developers include ***NetBeans***, ***Eclipse, IntelliJ, BlueJ*** etc.
- The IDE used in this module is: ***IntelliJ***

Installations

- Please refer to “Installations.pptx” for steps to install
  - JDK
  - IntelliJ
  - JavaFX

Java Programming Language

- Class based. Basic unit is class except for the 8 primitives data types
- Mostly object-oriented
- Case sensitive
- Statements housed in blocks of { }
- Statements end with ;
- Desktop applications begin execution from main()
- Better memory management than C++

Java Programming Language

Program template:

```
public class HelloWorld {
  public static void main(String args[]) {
       System.out.println("Hello, world!");
  }
}
```

Output:
```
Hello, world!
```

- The filename storing this class must be "***HelloWorld.java***"
- The ***main*** method is from where the execution actually begins
- **String args[]** is an array where each element is an input parameter when the program is run from CLI (can also be set in run configuration in any IDE)
- **System.out.println()** print the string to console output

Object-Oriented-Programming

- OOP attempts to model all problem situations as objects
- An object (or instance) belongs to a class
- A class is a blueprint definition containing attributes (or properties, or data members) and behaviours (or methods)

In non-OOP, storing these name, ID and GPA of 3 students requires 3 separate arrays

ゆりこ Yuriko
1923232
3.97

あおやま Aoyama
1919191
3.88

なおき Naoki
1908080
4.0

name[3]

Aoyama	Yuriko	Naoki

ID [3]

1919191	1923232	1908080

GPA [3]

3.88	3.97	4.0

In non-OOP, storing these name, ID and GPA of 3 students requires 3 separate arrays

ゆりこ Yuriko
1923232
3.97

あおやま Aoyama
1919191
3.88

```
int main()
{
    String name[3];
    String ID[3];
    double GPA[3];

    name[0] = "Aoyama";
    ID[0] = "1919191";
    GPA[0] = 3.88;

    name[1] = "Yuriko";
    //. . . .
    name[2] = "Naoki“
    //. . . .

}
```

なおき Naoki
1908080
4.0

In OOP, name, ID and GPA of individual student are clustered in 1 unit, which is an object of a class (in this case, SpStudent)

an instance of class SpStudent

ゆりこ Yuriko
1923232
3.97

an instance of class SpStudent

あおやま Aoyama
1919191
3.88

```
class SpStudent
{
    String name;
    String ID;
    double GPA;
}
```

an instance of class SpStudent

なおき Naoki
1908080
4.0

Create first SpStudent object and push it into an array.

(2nd and 3rd SpStudent objects can be created in similar way)

```
class SpStudent
{
    String name;
    String ID;
    double GPA;
}
```

an instance of class SpStudent

あおやま Aoyama

1919191

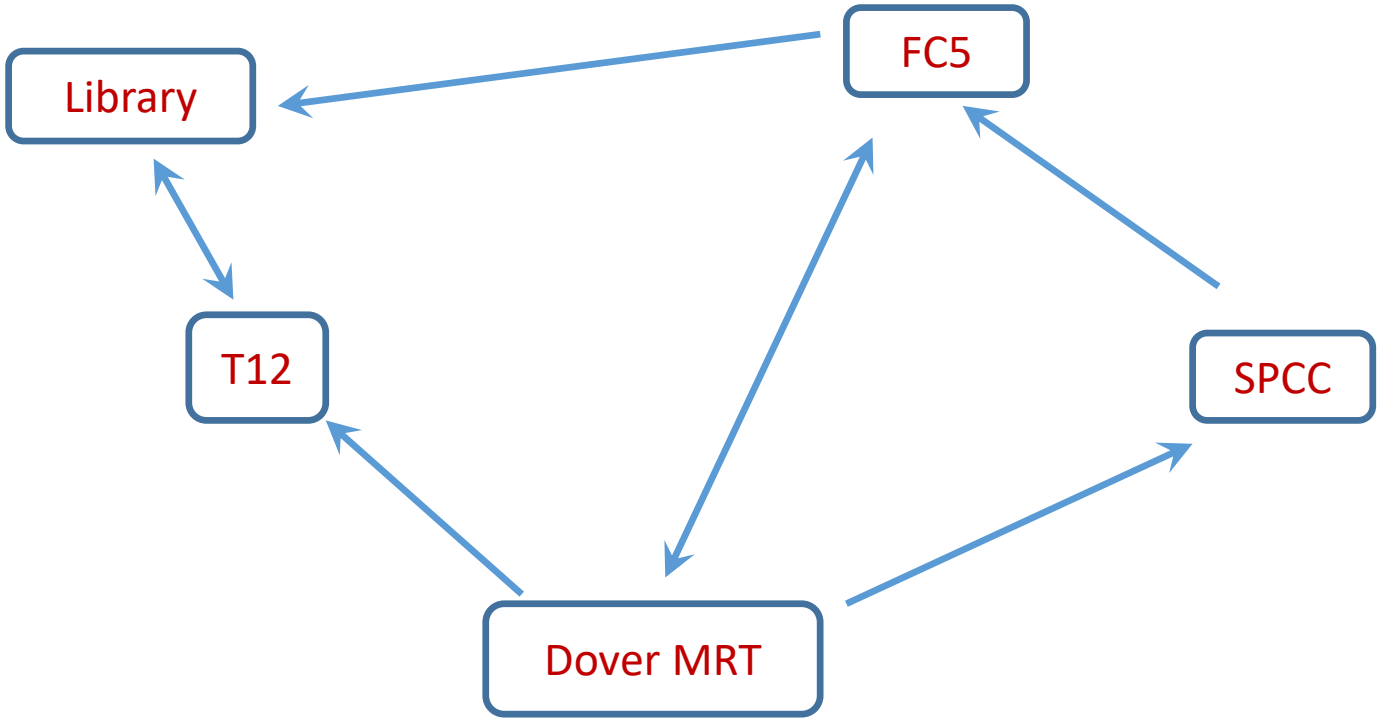
3.88

```
int main()
{
    SpStudent a[3];

    SpStudent s1 = new SpStudent();
    s1.name = "Aoyama";
    s1.adm = "1919191";
    s1.GPA = 3.88;

    // push the entire object into the array
    a[0] = s1;
    // reference name of the object
    System.out.println(a[0].name);
}
```

What is data structure?

Structure of how useful data is being organised in a computer program for better accessibility and scalability

For instance, if we have these places in SP and they are connected by either 1-way or 2-way traffics



If we are interested in their locations, one way is to store the longitude and latitude information.

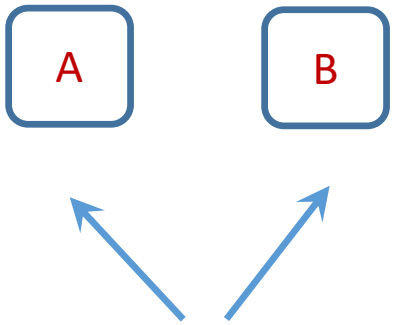
Dover MRT	1.311334, 103.778590
SPCC	1.3113028, 103.77949
FC5	1.309550, 103.776996
Library	1.308382, 103.779892
T12	1.310708, 103.778803

However, if are interested in finding the way from 1 place to another, we will store very different information and use different data structure in our program.

What is algorithm?

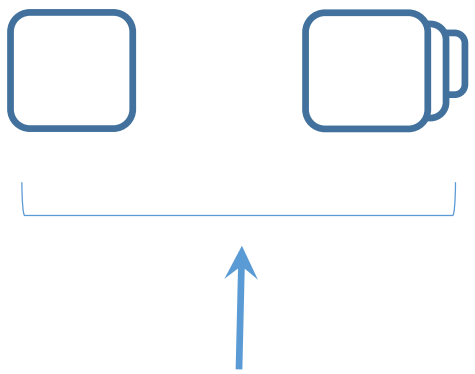
It is a series of steps to be taken to arrive at a solution using computer program

For instance, if we have 2 numbers, A and B.
We want the program to find out which number is bigger.
It is pretty easy to solve but still it takes a few simple steps.



- Start with one of the number, say A
- Compare it with the other number, B
- If (A > B) then A is bigger
- If (A < B) then B is bigger

What if we want to find the biggest among 1000 numbers?



- A more structured way of programming is needed
- A proper algorithm is needed
- The algorithm used must be scalable, regardless of how many numbers there are to be compared

Installation of JDK and IDE

- Please refer to installation guide

Elementary Programming in Java

Identifier, Variable and Constant

- Identifier is basically a name given to a class, method or variable, - a place where data is stored. An identifier
  - cannot start with a digit
  - cannot contain space
  - cannot be a Java keyword (e.g. class, if, while etc)
  - can only contains special characters of dollar sign and underscore
  - can start with special character but limited to only the underscore

- Variable name, is the identifier for a computer memory space to store data. The data value stored in the variable, can be changed.

- Constant, represents data whose value cannot be changed (e.g. the mathematical constant PI). Constants are declared with the keyword final.

Primitive data types

Data Type			Description	Size	Min	Max	Default
numeric	integers	byte	Byte-length integer	8-bit	-128 (-2^7)	127 (2^7-1)	(byte) 0
		short	Short integer	16-bit	-32768 (-2^{15})	32767 ($2^{15}-1$)	(short) 0
		int	Integer	32-bit	-2^{31}	$2^{31}-1$	0
		long	Long integer	64-bit	-2^{63}	$2^{63}-1$	0L
	floating-point numbers	float	Single-precision floating point	32-bit IEEE 754	±3.4E38 (6 to 7 significant digit of accuracy)		0.0F
		double	Double-precision floating point	64-bit IEEE 754	±1.7E308 (14 to 15 significant digit of accuracy)		0.0
Character char			A single character	16-bit	Single quotation		'\u0000' (null)
boolean			A boolean value	8-bit			false

Declaring Variable

A variable

- has to be declared before it can be used to store data (or assigned with a value)
- can only be declared once, within the same scope.
- can be declared again under a different scope of code but it will be treated as a different variable

```
int a;
int b;
int c;
// the above 3 lines can be combined as
// int a, b, c;

float d;
String e;
char f;
boolean g;
```

Initialising Variable

```
float h = 12.34f;
double m = 12.34;
int x;
x=55;
```

Declaring and Initialising Constant

- Constant is a value that cannot be changed after assigning it
- The naming convention of constant is capitalised letters and underscore
- Is defined with ***final***
- The ***final*** modifier makes the value fixed

```
final double MAX_WIDTH = 432.78;
```

- If it is defined with ***static*** and ***final***, the static modifier makes it available without loading any instance of the class in which it is defined

```
static final int MAX_CLASS_SIZE = 20;
```

Display output to console

```
System.out.println("I am a student from SP");
```

Output:

```
I am a student from SP
```

Display mixed text and variable

```
int acad_year;

acad_year = 3;

System.out.println("I am a year " + acad_year + " student from SP");
```

Output:

```
I am a year 3 student from SP
```

Example: **ComputeArea.java**

```
public class ComputeArea {
  // compute surface area of a cylinder

  public static void main(String args[]) {
    double radius; //declare a variable, data type is double
                   // variable is radius
    double surfaceArea;
    final double PI = 3.14159; //declare a constant PI
                               //  its value cannot be changed later
    double height;

    // Assign a radius
    radius = 10.0;//assign a value of 10 to a variable
    //radius =10.0;
    //the variable value can be changed by
    // assigning a different value to it

    Height = 2.0;

    // expression for computing surface area
    surfaceArea = (radius * radius * PI) + (height *(2*PI*radius));

    // Display results
    System.out.println("The surface area for the cylinder of radius "+ radius
      + " and height " + height + " is " + surfaceArea);
  }
}
```

Output:

```
The surface area for the cylinder of radius 10.0
and height 2.0 is 439.82259999999997
```

Literals

Literals are fixed values found in a program.

```
int a=0;
boolean found;
a = a + 500;                // 500 is an int literal
System.out.println('Z');    // Z is a char literal
found = false;              // false is a Boolean literal
long distance = 42.2;       // 42.2 is a long literal
int inputBinValue = 0b1111; // 1111 is a binary literal
```

Operators

There are 5 categories of Operators

- Assignment: =, +=, -=, *=, /=, %=
- Arithmetic: +, -, *, /, ++, --
- Comparison/Relational: >=, <=, ==, >, <, !=
- Boolean: !, &&, ||, ^
- Integer bitwise: >>, <<, |, &, >>>

Character Type and String Type

Character value is enclosed by single-quote ‘ ‘.
String value is enclosed by double-quote “ “.
String is NOT a primitive type. It is a class.

```
char a='A';
char b = 'B';
char c = 'b';

String d = "A";
String e = "A BCD ef";
```

Character type and Operations

- A character is stored as a sequence of 0 and 1 (binary)
- Mapping a character to binary is called *encoding*
- Java supports *Unicode*, a 16-bit encoding scheme (hex 0000 – FFFF)
- Hence, it is valid to code with \u (for 2-byte):

```
char a='\u1C2F';
```

- Unicode also include ASCII, an 8-bit encoding scheme for commonly used characters (\u0000 to \u007F for first 128 values)
- Hence, for the following statements, all character variables are assigned with character capital 'B':

```
char c1 ='\u0042';
char c2 = 66;
char c3 = 0x42';
```

Character	value in decimal	Unicode value
'0' to '9'	48 to 57	\u0030 to \u0039
'A' to 'Z'	65 to 90	\u0041 to \u005A
'a' to 'z'	97 to 122	\u0061 to \u007A

Character type and Operations

```
char c1 = 'B';
c1 += 5;                    // G

char c2 = 'C' + '5';        // }

System.out.println('Q' > 'q'); // false
```

Character	ASCII value in decimal
'0' to '9'	48 to 57
'A' to 'Z'	65 to 90
'a' to 'z'	97 to 122
:	:
{	123
|	124
}	125
:	:

String class

String is not a primitive type. String is a class.
A ***string object*** can be created by ***new*** keyword or from a ***literal***.

```
String m1 = new String("Welcome to Java!");
char[] c = new char[] {'c','o','v','i','d'};
String m2 = new String(c);
String m3 = new String(m3);
String m4 = "Welcome to Java!";
```

A ***string literal*** – a series of alphanumeric enclosed in double quotes.

Example:

```
"I love Java"
```

Whenever Java encounters a string literal for the first time, it creates an ***String object*** with the string literal, i.e. "`I love Java`". Subsequently, if the same value of this stored literal is required again, Java will not create any new copy but will reuse the stored copy.

Comparing Strings

```
String s1 = new String("I love Java");  // 1st encounter
"I love Java"
String s2 = new String("I love Java");
String s3 = " I love Java";
String s4 = " I love Java";
String s5 = new String (s4);
String s6 = " I love Java";
```

The keyword ***new*** always create a brand new object.

String ***s3, s4*** and ***s6*** are referencing the same String object that was created when the String literal was first encountered when creating ***s1***. This can be tested easily by:

```
System.out.println(s1==s2); // false
System.out.println(s1==s3); // false
System.out.println(s1==s4); // false
System.out.println(s2==s3); // false
System.out.println(s3==s4); // true
System.out.println(s4==s5); // false
System.out.println(s4==s6); // true
```

== compares references, not content of strings.

Comparing Strings

equals() compares content of strings

```
String s1 = new String("I love Java");  // 1st
encounter “I love Java”
String s2 = new String("I love Java");
String s3 = " I love Java";
String s4 = " I love Java";
String s5 = new String (s4);
String s6 = " I love Java";
System.out.println(s1.equals(s2)); // true
System.out.println(s3.equals(s1)); // true
System.out.println(s1.equals(s4)); // true
System.out.println(s2.equals(s3)); // true
System.out.println(s3.equals(s4)); // true
System.out.println(s5.equals(s4)); // true
System.out.println(s4.equals(s6)); // true
```

== compares references, not content of strings.

Comparing Strings

compareTo() compares two strings content lexicographically

```
String w = "Go Love SP";
String x = "I Love SG";
String y = "I Love SP";
String z = "Go Love SP";

System.out.println (x.compareTo(y)<=0);
System.out.println (w.compareTo(z)==0);

System.out.println (z.compareTo(y)>0);
System.out.println (y.compareTo(z)<0);
```

Output:

```
true
true
false
false
```

Some methods of String class

```
String s = new String("Welcome to Java!");   // declare and create a String object
```

```
s.length();            // no of characters in the string
s.charAt(5);           // character at index 5 – 'm'
s.indexOf("Java");     // 11
s.substring(3,9);      // come t
s.toUpperCase();       // WELCOME TO JAVA
s.contains("Java");    // true
s.contains("Well");    // false
```

More…

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html

Conversion of String

Other variable types can be converted to a ***String*** :

```
String.valueOf(23);        // convert 23 to "23"
String.valueOf(56.78)      // convert 56.78 to “56.78"
```

In reverse, a ***String*** can also be converted to ***numeric***, provided the original string contains only numeric and no alphabets, by the appropriate class and method:

```
String s1 = "356";
int a = Integer.parseInt(s1);        // a=356
String s2 = "3.789";
double b = Double.parseDouble(s2);  // b=3.789
```

```
String s3 = "I am 88";
double d = Double.parseDouble(s3);   // Error!!
```

String is immutable

Strings are ***immutable***, meaning that their values cannot be changed after they are created.

Refer to the code segment below.

At first, a new object is created with the ***string literal*** "some text".
Next, object ***s*** is created to reference the ***string literal***.
All the above happens in Line 1

```
String s = "some text";          // Line 1
System.out.println (s);
s = s + " here";                 // Line 3
System.out.println (s);
```

At Line 3, object ***s*** is updated to reference a new string literal `"some text here"`.

The old string literal is still in the Java String Pool but there is no way for this program to access it anymore. The Java garbage collector (from Java 7 onwards) performs this task by periodically identifying and reclaiming memory that is no longer in use.

Getting input from console

Use Scanner class.

System.in refers to console input from keyboard.

```
int age;
String name;
double GPA1,GPA2;

Scanner sc = new Scanner(System.in);
System.out.print ("enter name: ");
name = sc.nextLine();
System.out.print ("enter age: ");
age = sc.nextInt();
System.out.print ("enter GPA1: ");
GPA1= sc.nextDouble();
System.out.print ("enter GPA2: ");
GPA2= sc.nextDouble();
System.out.println ("You need GPA3 of " +
    (3.9*3-GPA1-GPA2) + " to get overall GPA 3.9");
```

Implicit Casting

- Implicit casting – data type is automatically cast (or converted) and generally applies when a data type with a smaller range is assigned into a bigger data type.

```
double a = 5;  // a is automatically assigned with 5.0 and not 5
byte m = 3;   // occupies 1 byte  storing the value 3
int  n = m;   // n occupies 4 bytes  storing the value 3
```

- byte, char, and short values are promoted to int before the operation.

```
byte b1 = 1, b2 = 2, b3 ;
float f;
f = b1 + b2;   //b1 and b2 are promoted to int for addition
               //the sum is converted to float. f=3.0

b3 = b1 + b2; //error: result is an int, b3 is byte
```

Explicit Casting

- Explicit casting (safer) – Specific indication of data type within parentheses () and is needed when assign a value to a variable of a type with smaller range (which may result in a loss of precision).

```
int k = (int) (6.0/4.7);
```

```
float t = 100.99f;
int p = t; // error!
int p = (int) t; // OK!
```

Formatting Output

```
System.out.printf("%d\n", 1234);
System.out.printf("%f\n", 12345.678f);
System.out.printf("%e\n", 12.345);
System.out.printf("%010d\n", 12);
```

Output:

```
1234
12345.677734
1.234500e+01
0000000012
```

%c	character
%d	decimal (integer) number (base 10)
%e	exponential floating-point number
%f	floating-point number
%i	integer (base 10)
%s	a string of characters
%u	unsigned decimal (integer) number
%x	number in hexadecimal (base 16)
%%	print a percent sign

```
System.out.format( "%s is a %s with %d sides.", "Pentagon", "shape", 5);
```

Output:

```
Pentagon is a shape with 5 sides.
```

if - else

- if – without else
- if – with one else
- if – with multiple else-if
- Nested if

Operator	Description	Usage	Example (x=5, y=8)
==	Equal to	*expr1* == *expr2*	(x == y) → false
!=	Not Equal to	*expr1* != *expr2*	(x != y) → true
>	Greater than	*expr1* > *expr2*	(x > y) → false
>=	Greater than or equal to	*expr1* >= *expr2*	(x >= 5) → true
<	Less than	*expr1* < *expr2*	(y < 8) → false
<=	Less than or equal to	*expr1* >= *expr2*	(y <= 8) → true

```
public class SelectionTest {
  public static void main(String args[]) {
     Scanner s;
     s = new Scanner(System.in);
     System.out.print ("Enter age:");
     int age = s.nextInt();
     if (age >=18 )
         System.out.println ("Can watch");
  }
}
```

```
public class SelectionTest {
  public static void main(String args[]) {
   int marks = 47;
   if (marks>=50)
           System.out.println ("PASS");
   else
          System.out.println ("FAIL");
  }
}
```

```
Scanner s;
s = new Scanner(System.in);
System.out.print ("Enter age:");
int age = s.nextInt();
if (age >=18 )
    System.out.println ("Can watch");
else if (age>=12)
     System.out.println ("Can watch with parents");
else
     System.out.println ("Go sleep");
```

More examples

```
Scanner s;
s = new Scanner(System.in);
System.out.print ("Enter 1st number:");
int num1 = s.nextInt();
System.out.print ("Enter 2nd number:");
int num2 = s.nextInt();

if (num1==num2 )
    System.out.println ("The numbers are equal.");
else {
     if (num1>num2)
               System.out.println ("1st number is bigger");
     else
               System.out.println ("2nd number is bigger");
}
```

Shorthand of if-else Structure

```
public class SelectionTest {
  public static void main(String args[]) {
     int age=20;
     String msg = (age >=18 )? "Can Watch": "Cannot watch";
     System.out.println (msg);
  }
}
```

switch-case

Works with byte, short, int (then of course char), Boolean, long and String class

```
public class SelectionTest {
  public static void main(String args[]) {
     int luckyDraw=2;
     switch(luckyDraw) {
     case 8:
     case 1:
     case 5:
          System.out.println ("1 Bubble Tea");
          break;
     case 2:
     case 6:
          System.out.println ("1 box of masks");
          break;
     case 9:
     case 4:
     case 3:
          System.out.println ("$5 Grab coupon");
     default:
          System.out.println ("1 packet drink");

     }
  }
}
```

Loops

```
public class LoopTest {
  public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
           int i = -1;
           while (i<0 || i>100) {
                     Systemv.out.println ("Enter marks : ");
                     i = sc.nextInt();
           } }}
```

```
public class LoopTest {
  public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
           int i;
           do {
                     System.out.println ("Enter marks : ");
                     i = sc.nextInt();
           } while (i<0 || i>100) ;
   }}
```

```
public class LoopTest {
  public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
           int i;
           for (i=-1; i<0 || i>100; ) {
                     System.out.println ("Enter marks : ");
                     i = sc.nextInt();
           }  }}
```

break

One simple way to design a loop is to let the loop runs forever ***until*** a 'condition' or scenario occurs.

Exit the loop using ***break***.

```
public class LoopTest {
  public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
         int i;
         for (; ;) {                  // endless loop
                  System.out.println ("Enter marks : ");
                  i = sc.nextInt();
                  if (i>=0 && i<=100)  // if i is within range 0 - 100
                     break;            // exit loop
         }
   }
}
```

continue

The keyword ***continue*** allows the current loop to skip the remaining statements and proceed to the next loop.

```
public class LoopTest {
  public static void main(String args[]) {
      Scanner sc = new Scanner(System.in);
         int i, sum=0;
         for (; ;) {                  // endless loop
                  System.out.println ("Enter a number: ");
                  i = sc.nextInt();
                  if (i<0) break;         // exit loop if negative
                  if (i%2==0) continue;  // go to next loop if even
                  sum += i;               // add to sum if odd
         }
         System.out.println ("Sum = " + sum);
   }
}
```

Method

- Java, and also other programming languages, provide a more efficient structure to handle *repeatedly-used* codes.
- In Java, it is called ***method***.
- A ***method***
  - belongs to a ***class***
  - is a collection of statements performing an operation inside a class
  - has a method *name*
  - contains the code to be reused
  - can be invoked (or called) by using the method name

Putting repeatedly-used code in a method

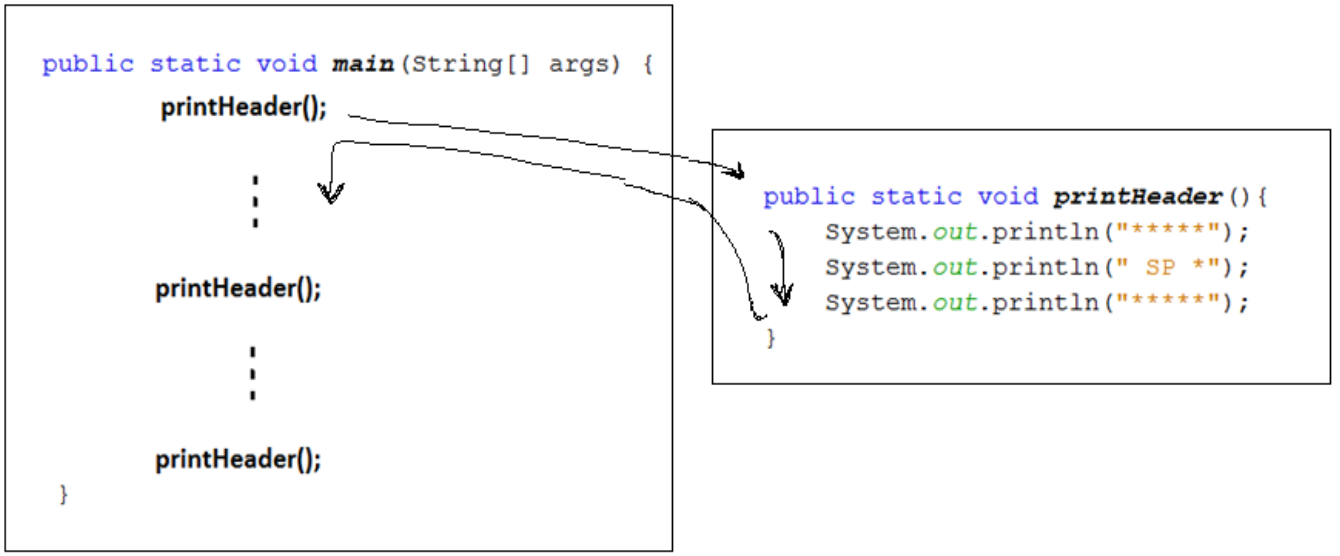
```
6   System.out.println("******");
7   System.out.println(" SP *");
8   System.out.println("******");
9
25  System.out.println("******");
26  System.out.println(" SP *");
27  System.out.println("******");
41  System.out.println("******");
42  System.out.println(" SP *");
43  System.out.println("******");
44
```

```
6   public static void main(String[] args) {
7           printHeader();
8
9
25
26          printHeader();
27
41
42          printHeader();
43  }

45  public static void printHeader(){
46      System.out.println("******");
47      System.out.println(" SP *");
48      System.out.println("******");
49  }
50
```

main() method

printHeader() method

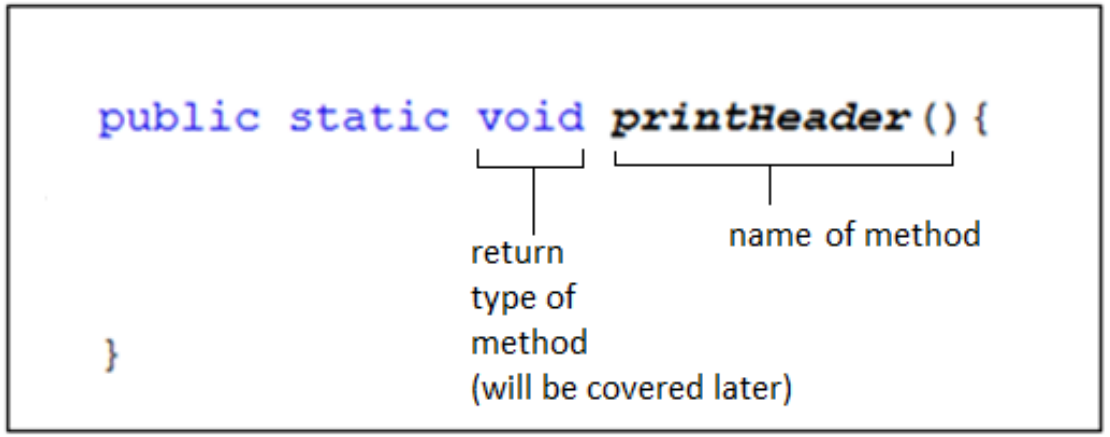
Executing sequence



Where to define method?

```
public class JavaApplication47 {

    public static void main(String[] args) {



    } // end of main

     public static void printHeader(){
          System.out.println("******");
          System.out.println(" SP *");
          System.out.println("******");
     }

} // end of proj class
```

Method - naming



Method – with arguments

Consider the following code segment with 3 clusters of *similar* codes.
The only difference among these 3 ***for loops*** is the value for the conditional check, which determines the number of loops.

```
for (int i=0; i<3; i++){
   System.out.println("******");
}

for (int i=0; i<8; i++){
   System.out.println("******");
}

for (int i=0; i<10; i++){
   System.out.println("******");
}
```

```
public class TestMethod {
   public static void main(String[] args) {
      printStar(3);
      printStar(8);
      printStar(10);
   }

   public static void printStar(int times){
      for (int i=0; i<times; i++){
         System.out.println("******");
      }
   }
}
```

The method ***printStar(int times)*** takes in a parameter.

This parameter MUST be an integer.

Hence, when the method is being called in ***main()***, an integer must be provided within the ( ).

e.g. ***printStar(3***)

Method – with return value

```
public static void computeAverage(int a, int b, int c)
{

}
```

must this always be ***void***?

```
public static returnValueType methodName ( arg-1-type arg-1, arg-2-type arg-2,... ) {
    body ;
}
```

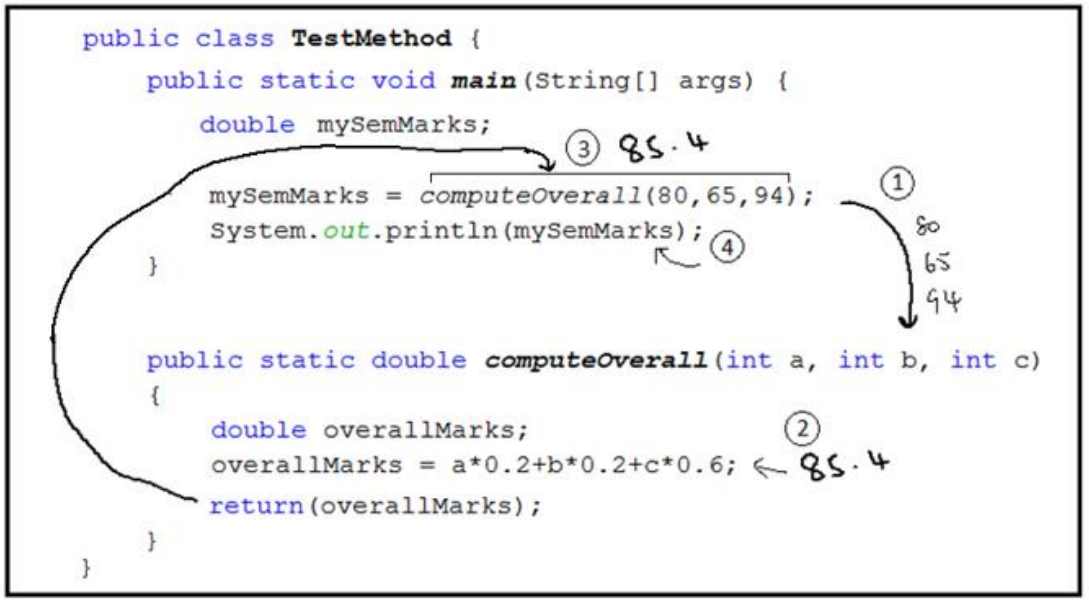
Data type (e.g.**int, double** etc) of the return value. **Void** if no return value is expected

Method – with return value

(1) - method call is being invoked in main(), with the 3 numbers.

(2) - execution continues inside method ***computeOverall()*** using the 3 numbers received as a=80, b=65 and c=94.

(3), a ***double*** value (overallMarks) is returned back to the caller, ***main***(). The returned value is being received and then assigned to the variable ***mySemMarks***, which MUST also be a matching ***double*** type



Output: **85.4**

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

    Double mySemMarks;

    mySemMarks = computeOverall(80,65,94);
                 :
}


public static double computeOverall(int a, int b, int c)
{
    double overallMarks;
                 :
    return(overallMarks);
}
```

(these 3 must match)

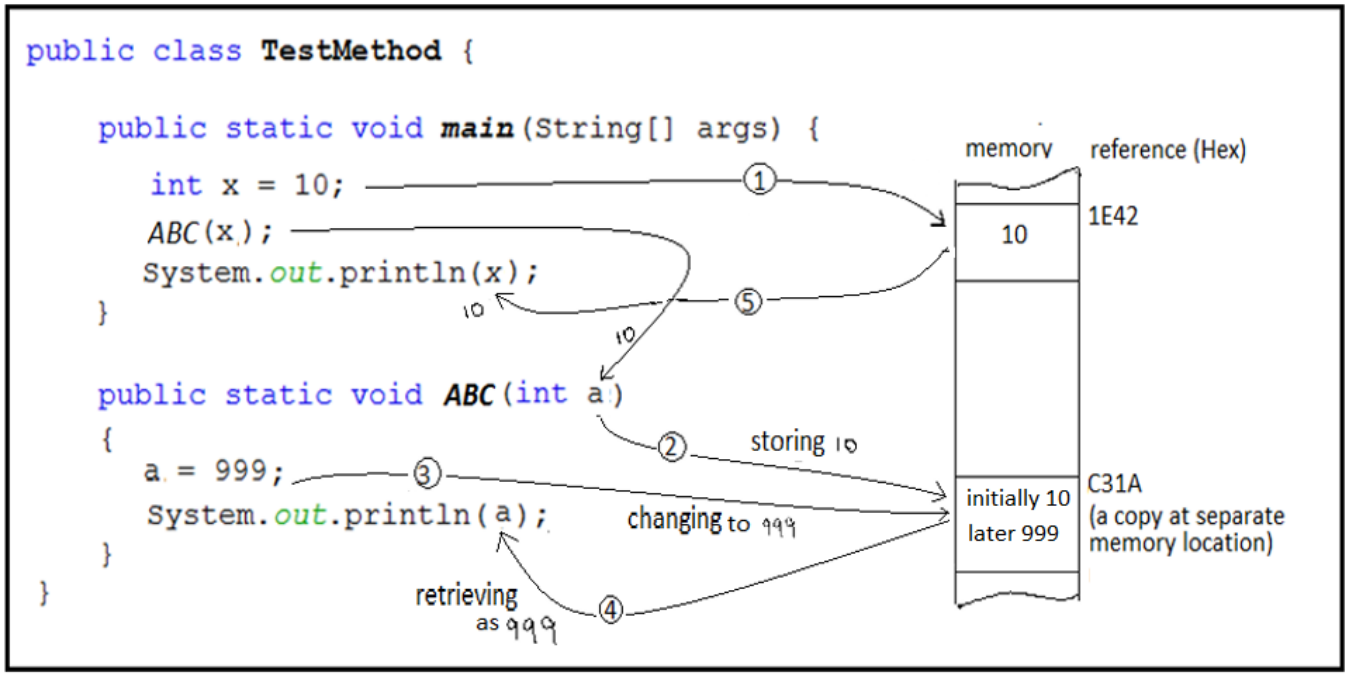
(all are double)

Argument Passed by Value

- In Java, *primitive types* (such as ***int, double, boolean*** etc) are ***passed by value***.
- The key difference between passed by value and reference : A "duplicate copy" of the data is created in a *separate memory location* when the value is passed into the method.
- The invoked method works on the "duplicate copy", and any changes made to this "duplicate copy" will not affect the data in the original memory location.

Argument Passed by Value

1. Variable ***x*** is allocated a memory location (say 1E42) and assigned with a value 10.
2. When the integer ***x*** is passed from ***main()*** to method ***ABC(),*** a duplicate copy of value 10 is stored in a *new memory* location (say C31A).
3. Code inside the method ***ABC()*** modifies the value of variable ***a*** to 999 (location (C31A)
4. Display variable ***a*** inside the method ***ABC()*** will give 999 (location C31A)
5. After the method call, displaying variable ***x*** in ***main()*** references the value stored in the original memory location (.e. 1E42), will show 10.



What will happen to the '*duplicate copy*' of the data at C314 after the method ***ABC()*** is done?

The data at this memory location is discarded and the memory is returned to the system after the ABC() has completed.

Argument Passed by Reference

- In Java, objects (e.g. ***arrays, Scanner etc***) are ***passed by reference***.
- The key difference between passed by value and reference : No "duplicate copy" of the data is created and only the reference (or address of the 1st memory location storing the object) is passed to the method.
- The invoked method uses this reference to store updates/changes of the data at the same memory location where the original data is stored.
- Hence, any changes made by the method will be continued back to the caller.

Argument Passed by Reference

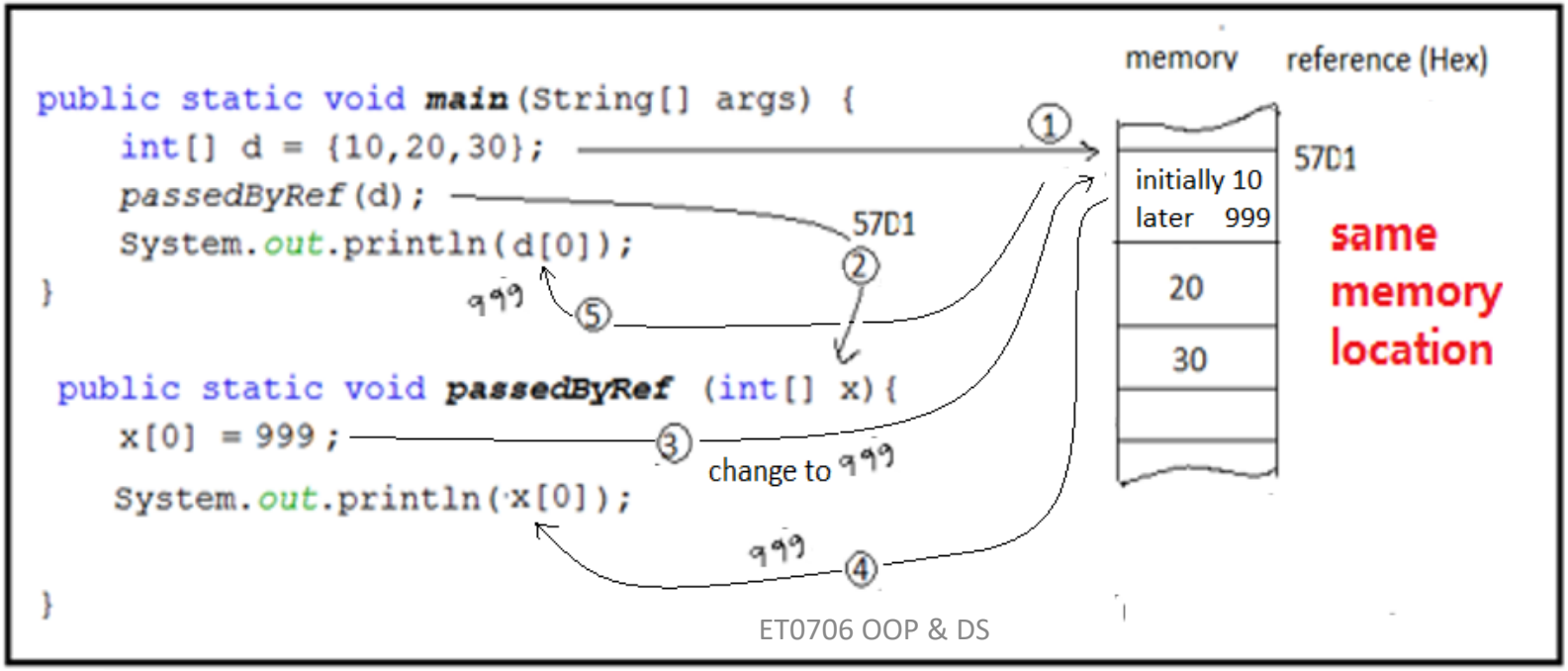
① Array ***d*** is allocated a block of memory locations (objects and array are not single value but advanced data structure with complex data) beginning say from address 57D1 and assigned with values of 10, 20 and 30.

② When the array ***d*** is passed from ***main()*** to method ***passByRef,*** it is the *reference* (or address of the 1st memory location of the entire block) i.e. 57D1 that is being passed over.

③ Base on the reference receive (i.e. 57D1), code inside the method ***passByRef()*** will be able to retrieve data for array ***x*** from the same memory locations, i.e. beginning from 57D1. Changes made within the method will overwrite the old data.

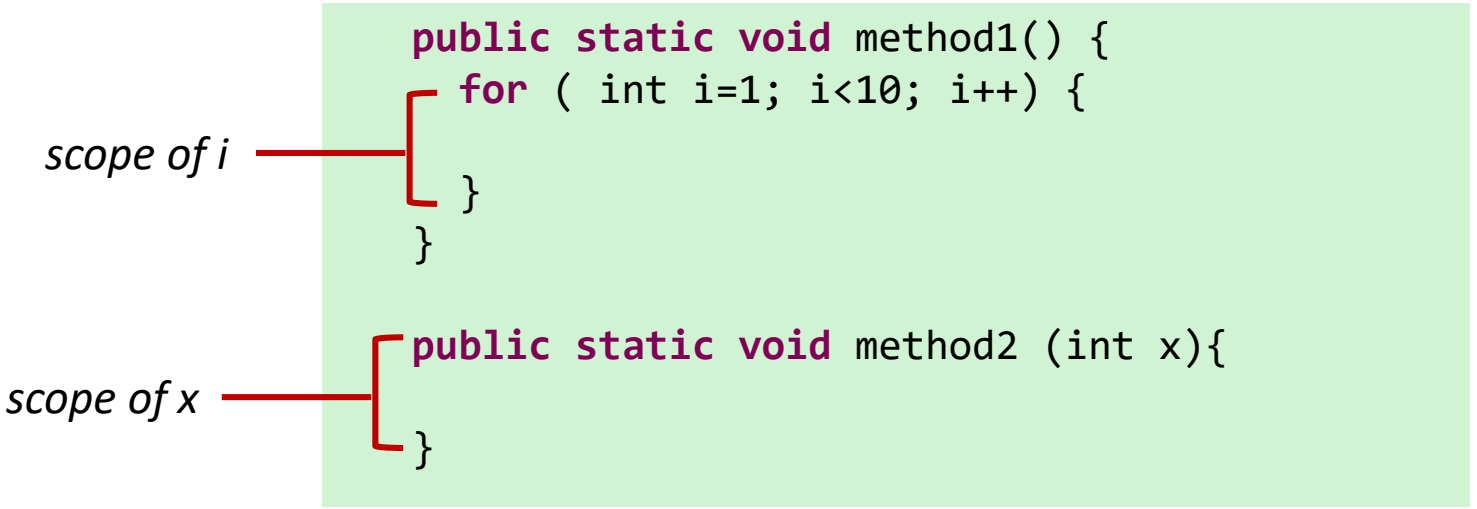
④ Display array element ***x[0]*** inside the method ***passByRef()*** will give the latest data, 999.

⑤ After the method call, displaying array element ***d[0]*** in ***main()*** will cause retrieval of the value stored in the same memory location (i.e. 57D1), and will also show 999.



Scope of Variables

- The scope of variable is determined by where the variable is being ***declared***.
- Variables declared within a method are not visible (not accessible) by codes outside the method.
- Variables declared within a loop are not visible (not accessible) by codes outside the loop.



Overloading Methods

- Methods with the same name but different signatures.
- This kind of methods are called overloading methods.
- Modifier or return value type need not be the same.
- Java compiler determines which method is used based on the best matching method signature.

Overloading Methods

```
public class TestMethodOverloading {
  public static void main(String[] args) {
    System.out.println("The minimum between 8,9 is "            + min(8, 9));
    System.out.println("The minimum between 7.7 , 3.3 is "      + min(7.7, 3.3));
    System.out.println("The minimum between 3.0, 5.4,10.14 is " + min(3.0, 5.4, 10.14));
    System.out.println("The minimum between 4 , 4.1 is "        + min(4, 4.1));
  }

  public static int min(int n1, int n2) {  //method 1
              if (n1 < n2)  return n1;
              else  return n2;
  }

  public static double min(double num1, double num2) {  //method 2
    if (num1 < num2)   return num1;
    else   return num2;
  }

  public static double min(int num1, double num2) { //method 3
              if (num1 < num2)    return num1;
              else   return num2;
 }

  public static double min(double num1, double num2, double num3) { //method 4
    return min(min(num1, num2), num3);
  }
}
```