Java Development Environment

Java Devlopment Environment

- **Software Development Kits (SDKs)** include the tools and documentation developers use to program applications. For example, you'll use the **Java Development Kit (JDK)** to build and run Java applications
- The Java compiler translates Java source code into **bytecodes** that represent the tasks to execute in the execution phase
- The **Java Virtual Machine (JVM)**—a part of the JDK and the foundation of the Java platform—executes bytecodes. A **Virtual Machine (VM)** is a software application that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications written for that type of VM can be used on all those platforms. The JVM is one of the most widely used virtual machines.

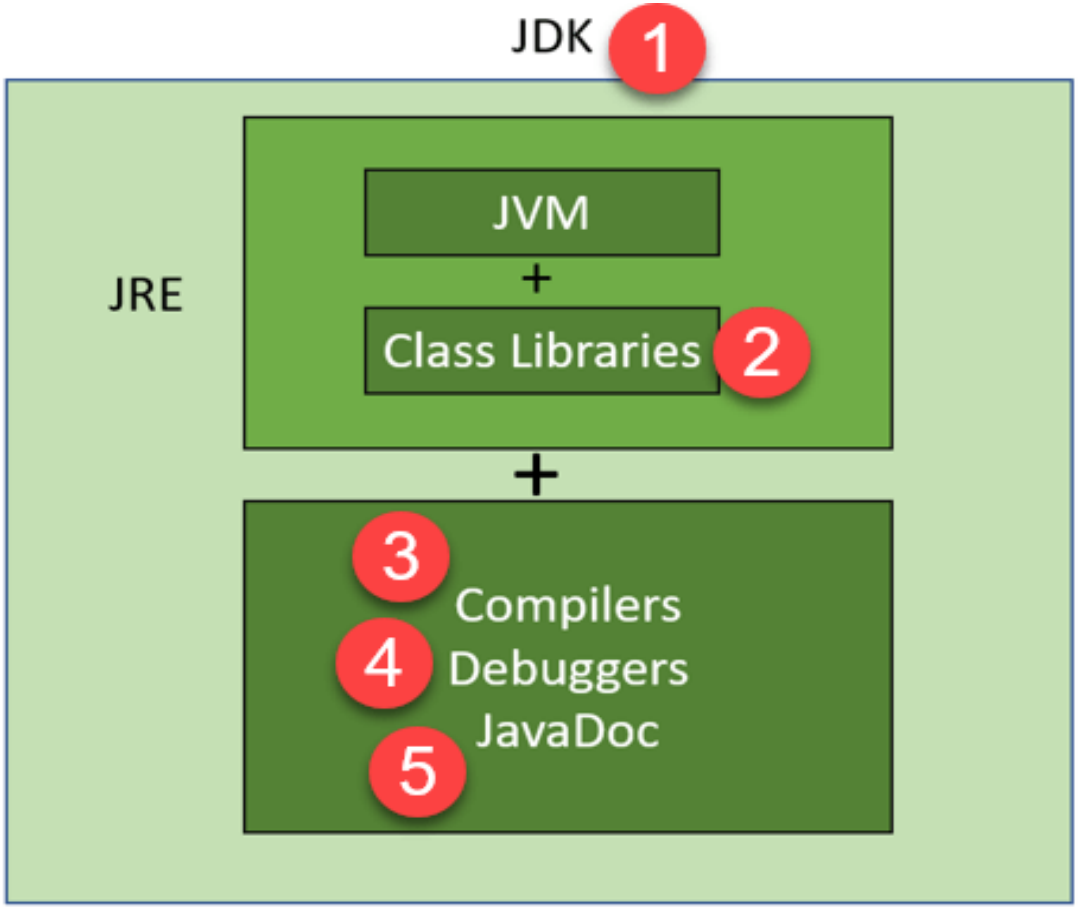
Java Devlopment Environment

- Unlike machine-language instructions, which are platform dependent (that is, dependent on specific computer hardware), bytecode instructions are platform independent.
- So, Java's bytecodes are **portable**—without recompiling the source code, the same bytecode instructions can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled.

JDK-JRE-JVM

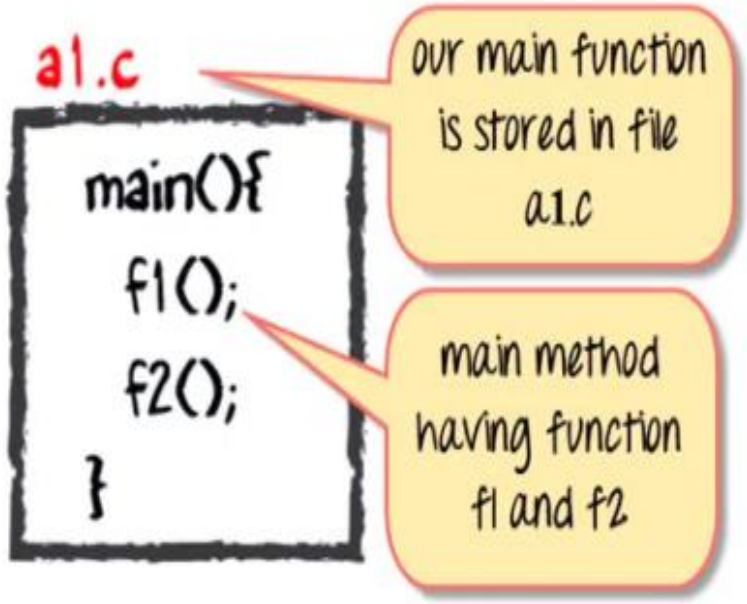
JDK	JRE	JVM
The full form of JDK is Java Development Kit.	The full form of JRE is Java Runtime Environment.	The full form of JVM is Java Virtual Machine.
JDK is a software development kit to develop applications in Java.	It is a software bundle which provides Java class libraries with necessary components to run Java code.	JVM executes Java byte code and provides an environment for executing it.
JDK is platform dependent.	JRE is also platform dependent.	JVM is highly platform dependent.
It contains tools for developing, debugging, and monitoring java code.	It contains class libraries and other supporting files that JVM requires to execute the program.	Software development tools are not included in JVM.
It is the superset of JRE	It is the subset of JDK.	JVM is a subset of JRE.
The JDK enables developers to create Java programs that can be executed and run by the JRE and JVM.	The JRE is the part of Java that creates the JVM.	It is the Java platform component that executes source code.
JDK comes with the installer.	JRE only contain environment to execute source code.	JVM bundled in both software JDK and JRE.

JDK-JRE-JVM

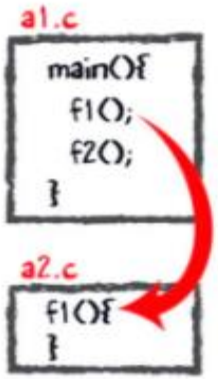

JDK
1
JRE
JVM
+
Class Libraries
2
+
3
Compilers
4
Debuggers
JavaDoc
5

Example

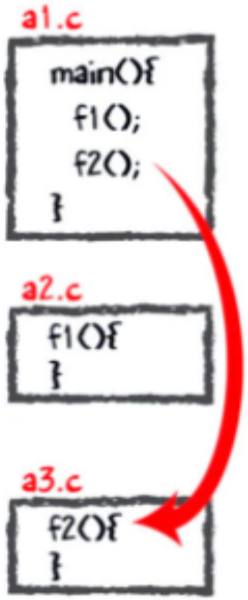
Suppose in the main, you have called two function f1 and f2. The main function is stored in file a1.c.



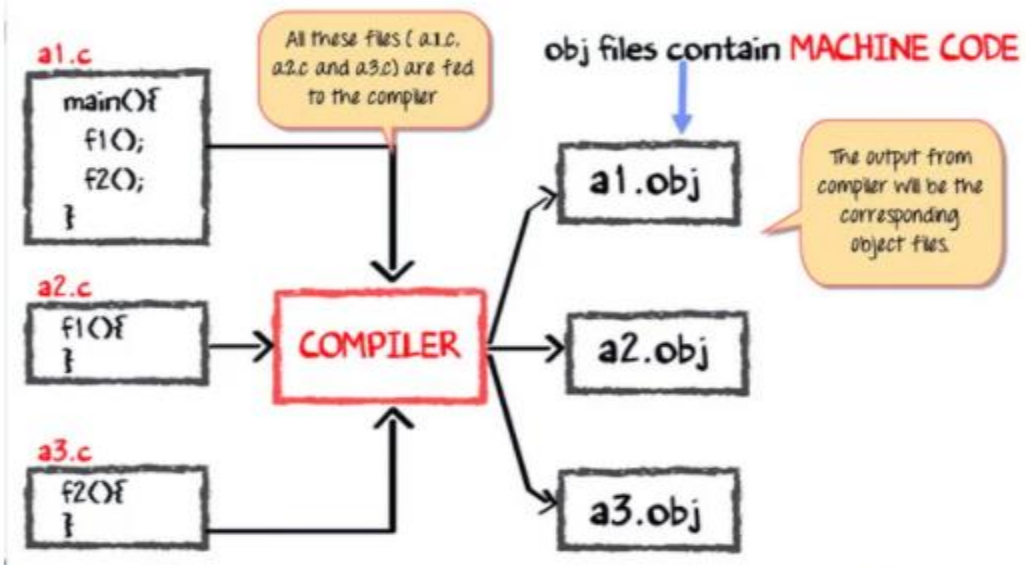
Function f1 is stored in a file a2.c



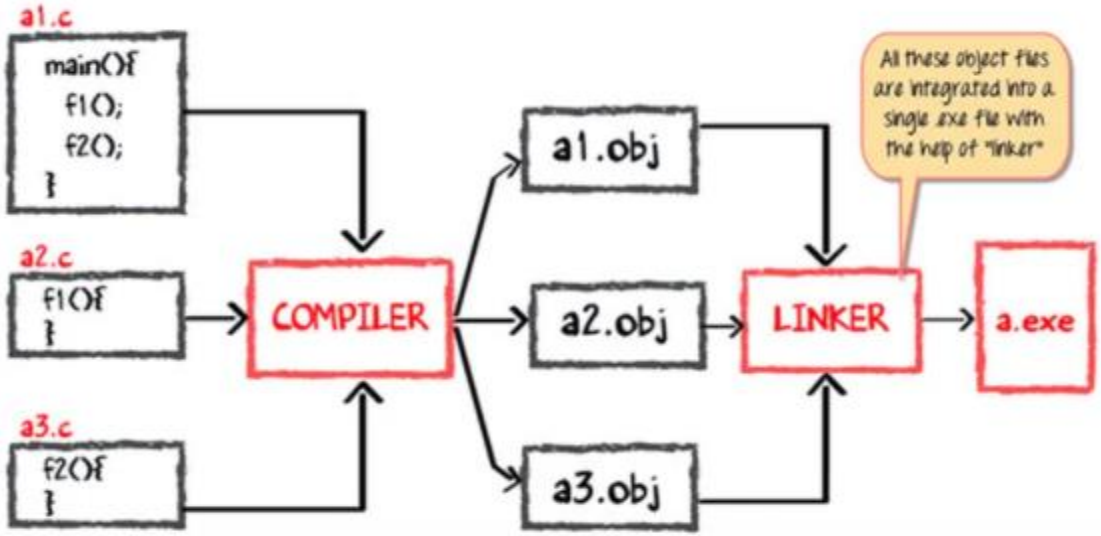
Function f2 is stored in a file a3.c



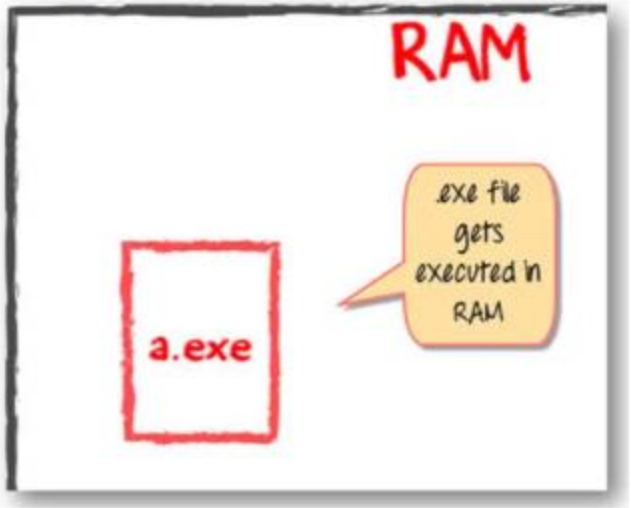
All these files, i.e., a1.c, a2.c, and a3.c, is fed to the compiler. Whose output is the corresponding object files which are the machine code.



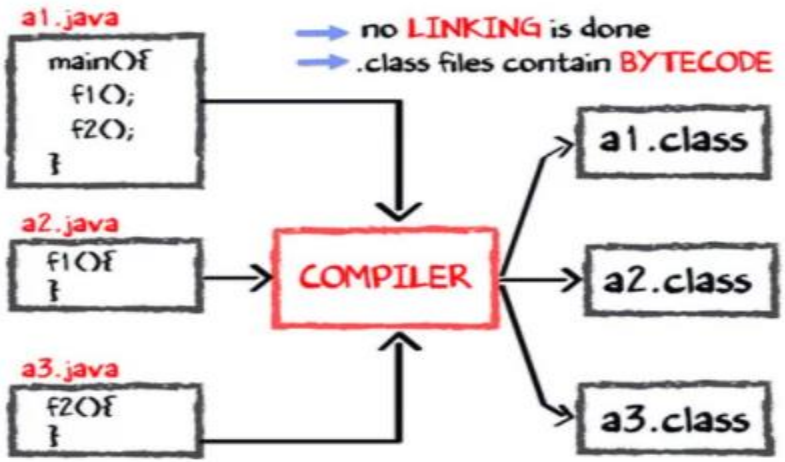
The next step is integrating all these object files into a single .exe file with the help of linker. The linker will club all these files together and produces the .exe file.



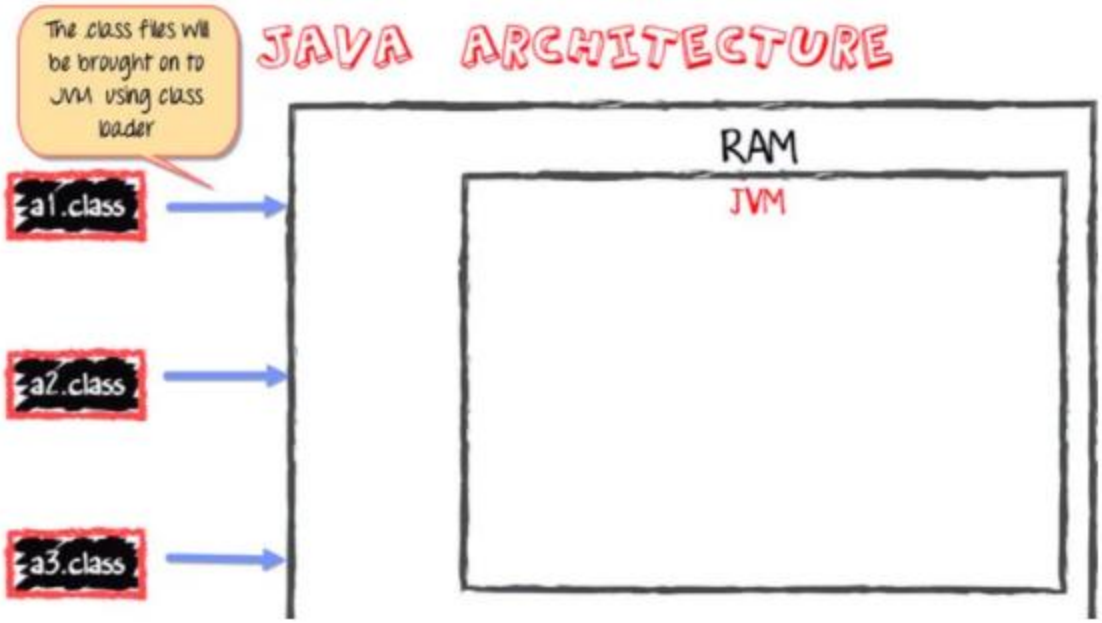
During program run, a loader program will load a.exe into the RAM for the execution.



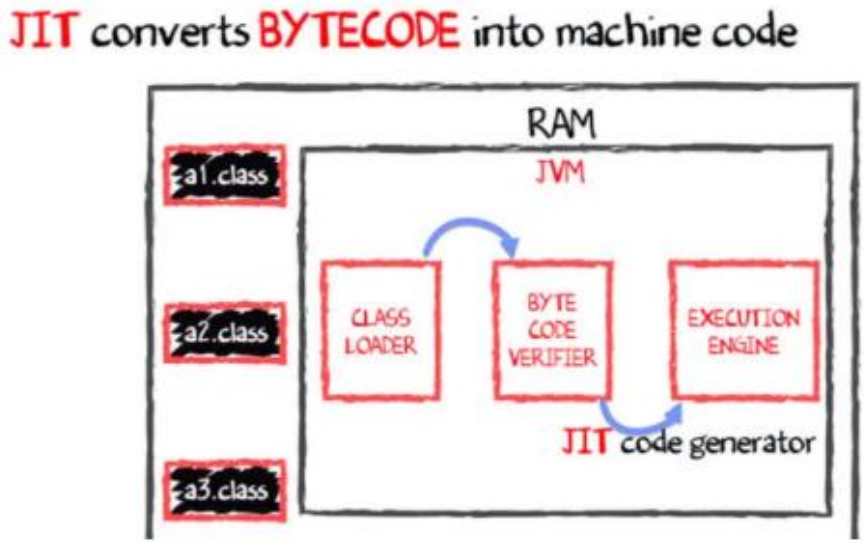
- The main method is stored in file a1.java
- f1 is stored in a file as a2.java
- f2 is stored in a file as a3.java



The Java VM or Java Virtual Machine resides on the RAM. During execution, using the class loader the class files are brought on the RAM. The BYTE code is verified for any security breaches.

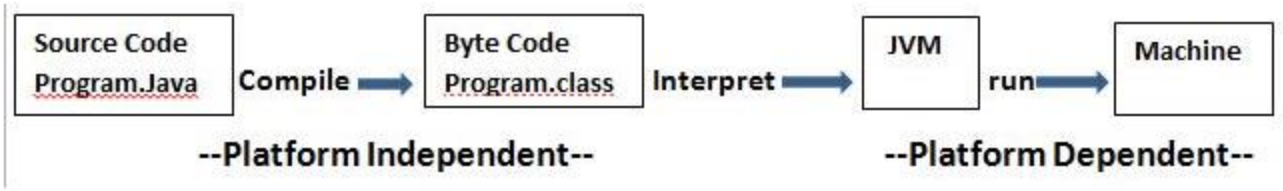


Next, the execution engine will convert the Bytecode into Native machine code. This is just in time compiling. It is one of the main reason why Java is comparatively slow.



- Java is platform-independent and JVM is platform-dependent

Platform-independent and Platform Dependent



Sample Java Program

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.println("Welcome to Java Programming!");
10    } // end method main
11 } // end class Welcome1
```

Comments

- Commenting single line can be done with double slash (//)
- Commenting multiple lines can be done using delimiters, /* and */

```
// Fig. 2.1: Welcome1.java
```

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

Class

- Every Java program consists of **at least one class** that you (the programmer) define.
- The **class keyword** introduces a class declaration and is immediately followed by the **class name (Welcome1**).
- every class we define begins **with the public keyword** (there are exceptions also will be discussed later).
- By convention, **class names begin with a capital letter and capitalize the first letter of each word they include (e.g., SampleClassName)**. A class name is an identifier—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces.
- **Class Body:** A left brace { begins the body of every class declaration. A corresponding right brace } must end each class declaration.

The Main Method

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

- is the starting point of every Java application. The parentheses after the identifier main indicate that it's a program building block called **a method**.
- Java class declarations normally contain one or more methods. For a Java application, **one of the methods must be called main and must be defined** otherwise, the Java Virtual Machine (JVM) will not execute the application.
- Keyword **static** means the method can be called without creating objects of class.
- Keyword **void** indicates that this method will not return any information.
- the **String[] args** in parentheses is a required part of the method main's declaration
- A method declaration must also start and end with left and right braces respectively { } .

Printing Lines

```
1  // Fig. 2.6: Welcome4.java
2  // Displaying multiple lines with method System.out.printf.
3
4  public class Welcome4
5  {
6     // main method begins execution of Java application
7     public static void main(String[] args)
8     {
9        System.out.printf("%s%n%s%n",
10          "Welcome to", "Java Programming!");
11    } // end method main
12 } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Adding Integers Example

```
1  // Fig. 2.7: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition
6  {
7     // main method begins execution of Java application
8     public static void main(String[] args)
9     {
10       // create a Scanner to obtain input from the command window
11       Scanner input = new Scanner(System.in);
12
13       int number1; // first number to add
14       int number2; // second number to add
15       int sum; // sum of number1 and number2
16
17       System.out.print("Enter first integer: "); // prompt
18       number1 = input.nextInt(); // read first number from user
19
20       System.out.print("Enter second integer: "); // prompt
21       number2 = input.nextInt(); // read second number from user
22
23       sum = number1 + number2; // add numbers, then store total in sum
24
25       System.out.printf("Sum is %d%n", sum); // display sum
26    } // end method main
27 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Import Declarations

- A great strength of Java is its rich set of predefined classes that you can reuse rather than **"reinventing the wheel."**
- These classes are grouped into **packages**—named groups of related classes—and are collectively referred to as the Java class library.

```
import java.util.Scanner;
```

- An import declaration that helps the compiler locate a class that's used in this program. It indicates that the program uses the predefined Scanner class (discussed shortly) from the package named java.util. The compiler then ensures that you use the class correctly.

System input and output

- A variable is a location in the computer's memory where a value can be stored for use later in a program. All Java variables must be declared with a name and a type before they can be used. A variable's name enables the program to access the value of the variable in memory.

```
Scanner input = new Scanner(System.in);
```

- Scanner is a variable declaration statement that specifies the name (input) and type (Scanner) of a variable that's used in this program. A Scanner enables a program to read data (e.g., numbers and strings) for use in a program. The data can come from many sources, such as the user at the keyboard or a file on disk. Before using a Scanner, you must create it and specify the source of the data.
- **System.out** is used for printing values.

Variable

- There are three types of variables: local, instance and static.
- **Local Variable:**
- A variable that is declared inside the method is called local variable.
- **Instance Variable:**
- A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.
- **Static variable:**
- A variable that is declared as static is called static variable. It cannot be local.

- Examp

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

Control Statements

Java's Selection Statements

- Java supports two selection statements: **if and switch.**
- **These statements allow you to** control the flow of your program's execution based upon conditions known only during run time.
- There are various types of if statement in java.
  - if statement
  - if-else statement
  - nested if statement
  - if-else-if ladder

Java If statement:

- if (*condition*) *statement1*;

```
public class IfExample {
public static void main(String[] args) {
    int age=20;
    if(age>18){
        System.out.print("Age is greater than 18");
    }
}
}
```

Java IF-else Statement

- if (*condition*) *statement1*;
  else *statement2*;

```
public class IfElseExample {
public static void main(String[] args) {
    int number=13;
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Java IF-else-if ladder Statement

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Java Nested ifs:

```
if(i == 10) {
  if(j < 20) a = b;
  if(k > 100) c = d; // this if is
  else a = c;        // associated with this else
}
else a = d;          // this else refers to if(i == 10)
```

Switch Case

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......

default:
 code to be executed if all cases are not matched;
}
```

expression must be of type ***byte***, ***short***, ***int***, ***char***, or an enumeration.
Expression can also be of type ***String***.
Each value specified in the case statements must be a unique constant expression (such as a literal value). Duplicate **case values are not** allowed. The type of each value must be compatible with the type of *expression*.

Switch Case

- Example:

```
public class SwitchExample {
public static void main(String[] args) {
    int number=20;
    switch(number){
    case 10: System.out.println("10");break;
    case 20: System.out.println("20");break;
    case 30: System.out.println("30");break;
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Switch Case

- The break statement is optional. If you omit the break, execution will continue on into the next case.
- It means it executes all statement after first match if break statement is not used with switch cases.
- It is sometimes desirable to have multiple cases without break statements between them.

```
public class SwitchExample2 {
public static void main(String[] args) {
    int number=20;
    switch(number){
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Output:

20
30
Not in 10, 20 or 30

- For example:

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

- Output:

```
I is less than 5
I is less than 5
I is less than 5
I is less than 5
I is less than 5
I is 10 or more
I is less than 10
I is less than 10
I is less than 10
I is less than 10
I is 10 or more
I is 10 or more
```

Nested switch Statements:

```
switch(count) {
  case 1:
    switch(target) { // nested switch
      case 0:
        System.out.println("target is zero");
        break;
      case 1: // no conflicts with outer switch
        System.out.println("target is one");
        break;
    }
    break;
  case 2: // …
```

Iteration Statements

- Java's iteration statements are **for, while, and do-while. These statements create what** we commonly call *loops*.

For loop

```
for(initialization;condition;incr/decr){
//code to be executed
}
```

```
public class ForExample {
public static void main(String[] args) {
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
}
}
```

Using the Comma in for loop:

```
class Sample {
  public static void main(String args[]) {
    int a, b;

    b = 4;
    for(a=1; a<b; a++) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      b--;
    }
  }
}
```

```
// Using the comma.
class Comma {
  public static void main(String args[]) {
    int a, b;

    for(a=1, b=4; a<b; a++, b--) {
      System.out.println("a = " + a);
      System.out.println("b = " + b);
    }
  }
}
```

Output

```
a= 1
b= 4
a= 2
b= 3
```

```
a= 1
b= 4
a= 2
b= 3
```

Some for Loop Variations

```
boolean done = false;

for(int i=1; !done; i++) {
  // …
  if(interrupted()) done = true;
}
```

In this example, the **for loop continues to run until the boolean variable done is set to true. It does not test the value of i.**

```
// Parts of the for loop can be empty.
class ForVar {
  public static void main(String args[]) {
    int i;
    boolean done = false;

    i = 0;
    for( ; !done; ) {
      System.out.println("i is " + i);
      if(i == 10) done = true;
      i++;
    }
  }
}
```

Here, the initialization and iteration expressions have been moved out of the **for. Thus,** parts of the **for are empty.**

done done==true

!done done==false

Java For-each Loop:

- The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

- It works on elements basis not index. It returns element one by one in the defined variable.

Java For-each Loop:

```
for(Type var:array){
//code to be executed
}
```

```
public class ForEachExample {
public static void main(String[] args) {
    int arr[]={12,23,44,56,78};
    for(int i:arr){
        System.out.println(i);
    }
}
}
```

Output:
12
23
44
56
78

```
for(String i:names){
 print(i);
}
```

Java Labeled For Loop:

- We can have name of each for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.
- Normally, break and continue keywords breaks/continues the inner most for loop only.

```
labelname:

for(initialization;condition;incr/decr){

//code to be executed

}
```

```
public class LabeledForExample {
public static void main(String[] args) {
    aa:
        for(int i=1;i<=3;i++){
            bb:
                for(int j=1;j<=3;j++){
                    if(i==2&&j==2){
                        break aa;
                    }
                    System.out.println(i+" "+j);
                }
        }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
```

If you use **break bb;**, it will break inner loop only which is the default behavior of any loop.

Using break to Exit a Loop:

- By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.

- Example:

```
// Using break to exit a loop.
class BreakLoop {
  public static void main(String args[]) {
    for(int i=0; i<100; i++) {
      if(i == 10) break; // terminate loop if i is 10
      System.out.println("i: " + i);
    }
    System.out.println("Loop complete.");
  }
}
```

Output

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
loop complete
```

- When used inside a set of nested loops, the break statement will only break out of the innermost

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i< 3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.
```

- **Using continue:**
- Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.
- The continue statement performs such an action.

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {

    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

Output

```
0 1
2 3
4 5
6 7
8 9
```

- As with the break statement, continue may specify a label to describe which enclosing loop to continue.

```
// Using continue with a label.
class ContinueLabel {
  public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
         for(int j=0; j<10; j++) {
           if(j > i) {
             System.out.println();
             continue outer;
           }
           System.out.print(" " + (i * j));
         }
       }
       System.out.println();
  }
}
```

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

return statement:

- The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- As you can see, the **final println()** statement is not executed. As soon as return is executed, control passes back to the caller.

```
// Demonstrate return.
class Return {
  public static void main(String args[]) {
    boolean t = true;

    System.out.println("Before the return.");

    if(t) return; // return to caller

    System.out.println("This won't execute.");
  }
}
```

Output:
Before the return.

Thank you