

JAVA

How Java Changed the Internet

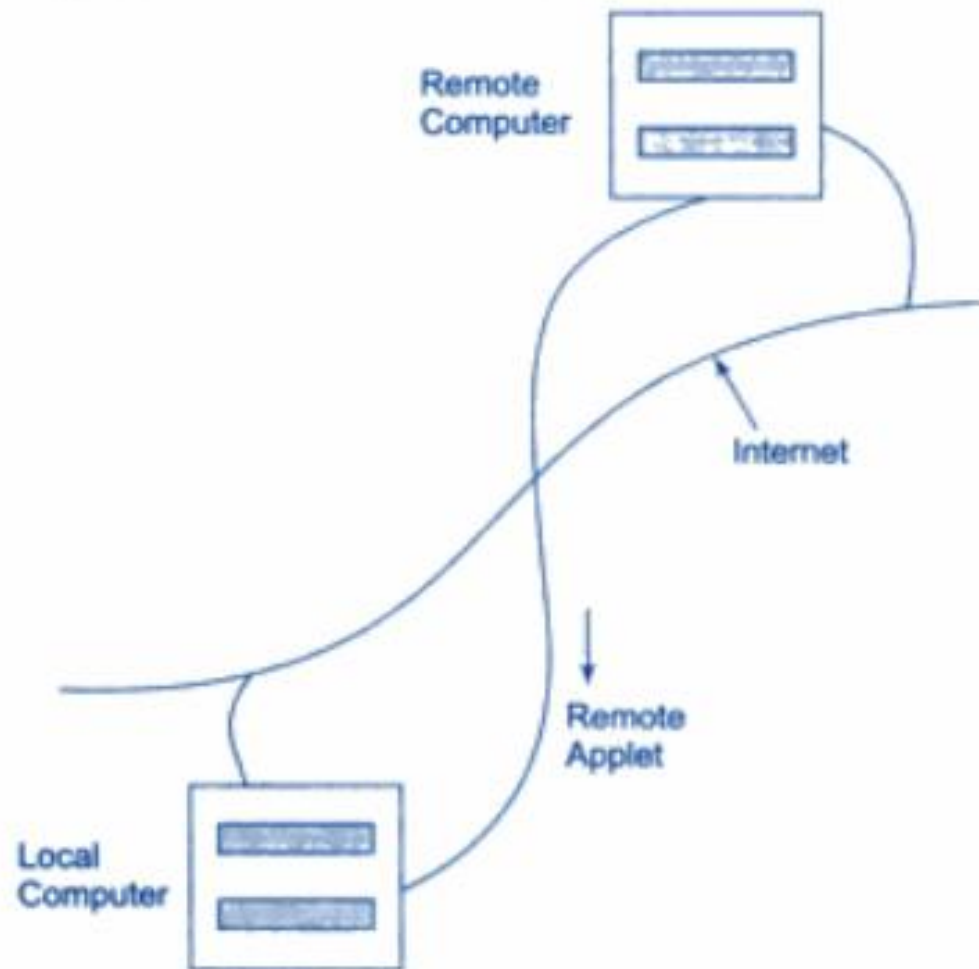
- Java had a profound effect on the Internet.
- Besides applet, Java also addressed some of the thorniest issues associated with the Internet: portability and security.
- Let's look more closely at each of these-

How Java Changed the Internet-Applets (cont...)

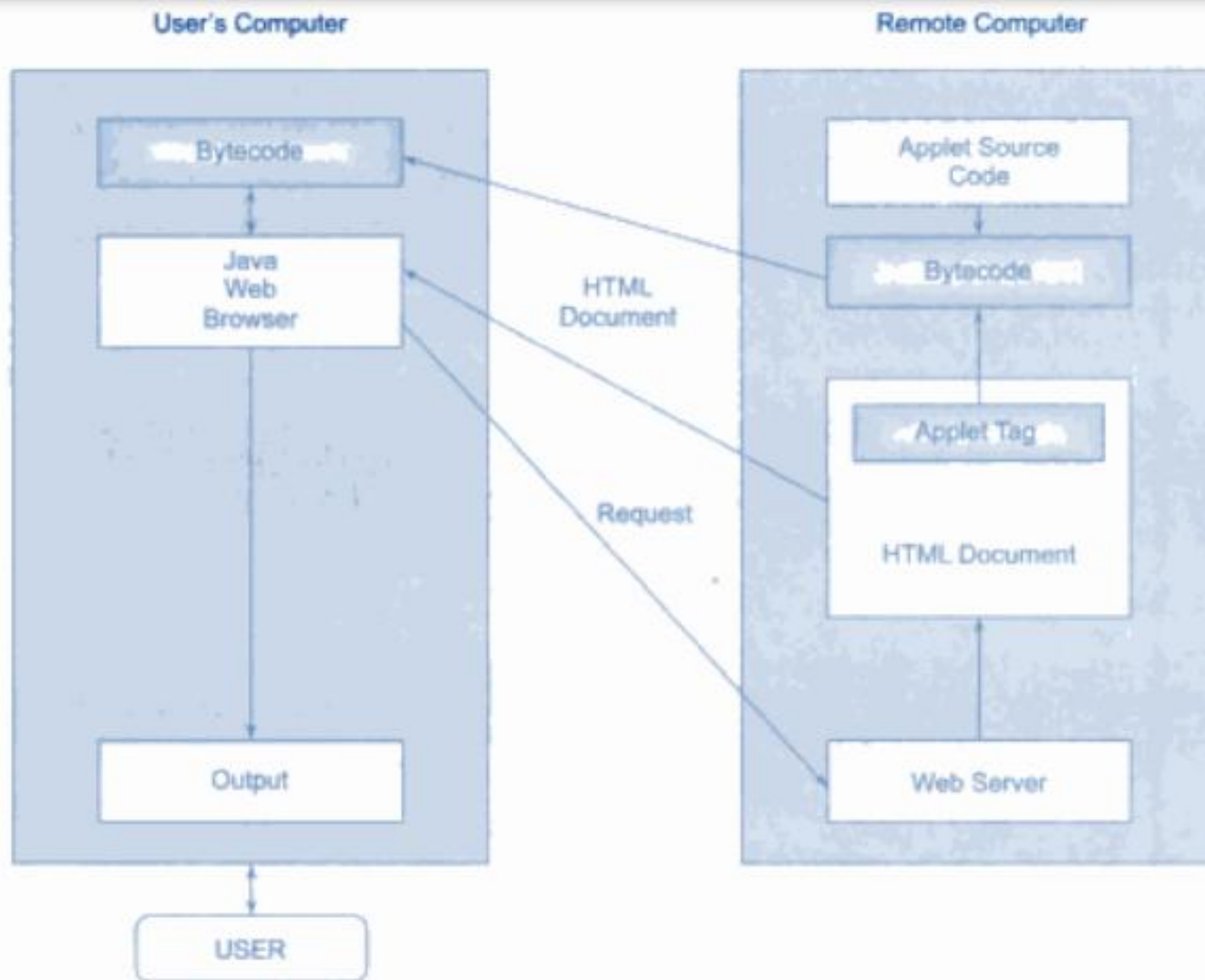
- An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser.
- Furthermore, an applet is downloaded on demand, without further interaction with the user.
- If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.
- Applets are intended to be small programs.
- They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server.
- In essence, the applet allows some functionality to be moved from the server to the client.

How Java Changed the Internet-Applets (cont...)

- The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace.
- The applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.
- As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability.
- Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm.
- It must also be able to run in a variety of different environments and under different operating systems.
- As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each.



Downloading of applets via Internet



Java's interaction with the web

How Java Changed the Internet-Security

- Every time we download a “normal” program, we are taking a risk, because the code we are downloading might contain a virus, Trojan horse, or other harmful code.
- At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.
- In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
- Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.
- The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

How Java Changed the Internet-Portability

- Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it.
- If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems.
- It is not practical to have different versions of the applet for different computers.
- The same code must work on all computers.
- Therefore, some means of generating portable executable code was needed.
- Java has the same mechanism that helps ensure security also helps create portability.

Java's Magic: The Bytecode

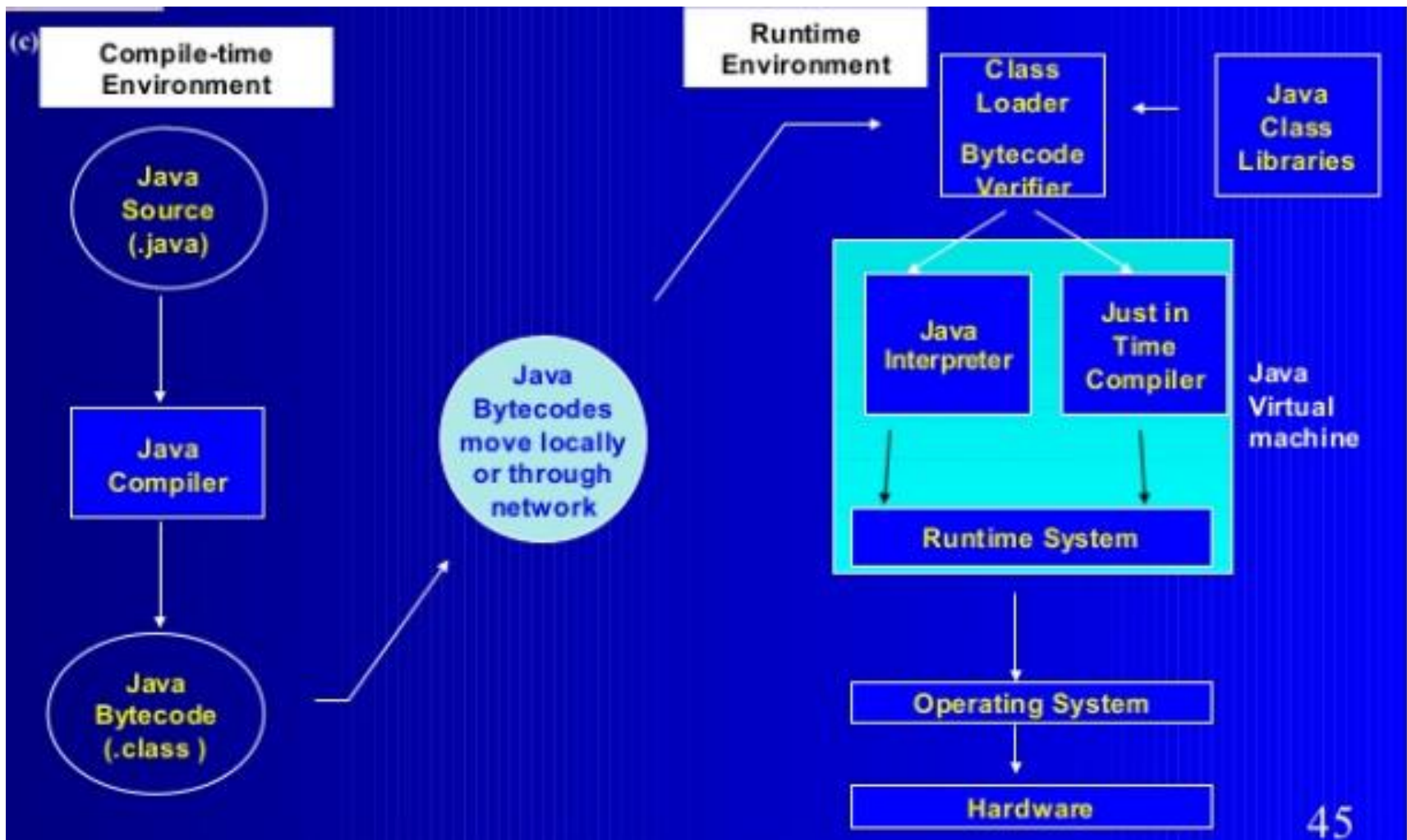
- The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode.
- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- In essence, the original JVM was designed as an interpreter for bytecode.
- Many modern languages are designed to be compiled into executable code because of performance concerns.
- However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs.

Java's Magic: The Bytecode (cont...)

- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- The fact that a Java program is executed by the JVM also helps to make it secure.
- Because the JVM is in control, it can contain the program and prevent it from generating.
- Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.
- HotSpot technology provides a Just-In-Time (JIT) compiler for bytecode.

Java's Magic: The Bytecode (cont...)

- A JIT compiler compiles code as it is needed, during execution.
- Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation.
- The remaining code is simply interpreted.
- However, the just-in-time approach still yields a significant performance boost.
- Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.



a1.c

```
main(){  
    f1();  
    f2();  
}
```

a1.c

```
main(){  
    f1();  
    f2();  
}
```

a2.c

```
f1(){  
}
```

a1.c

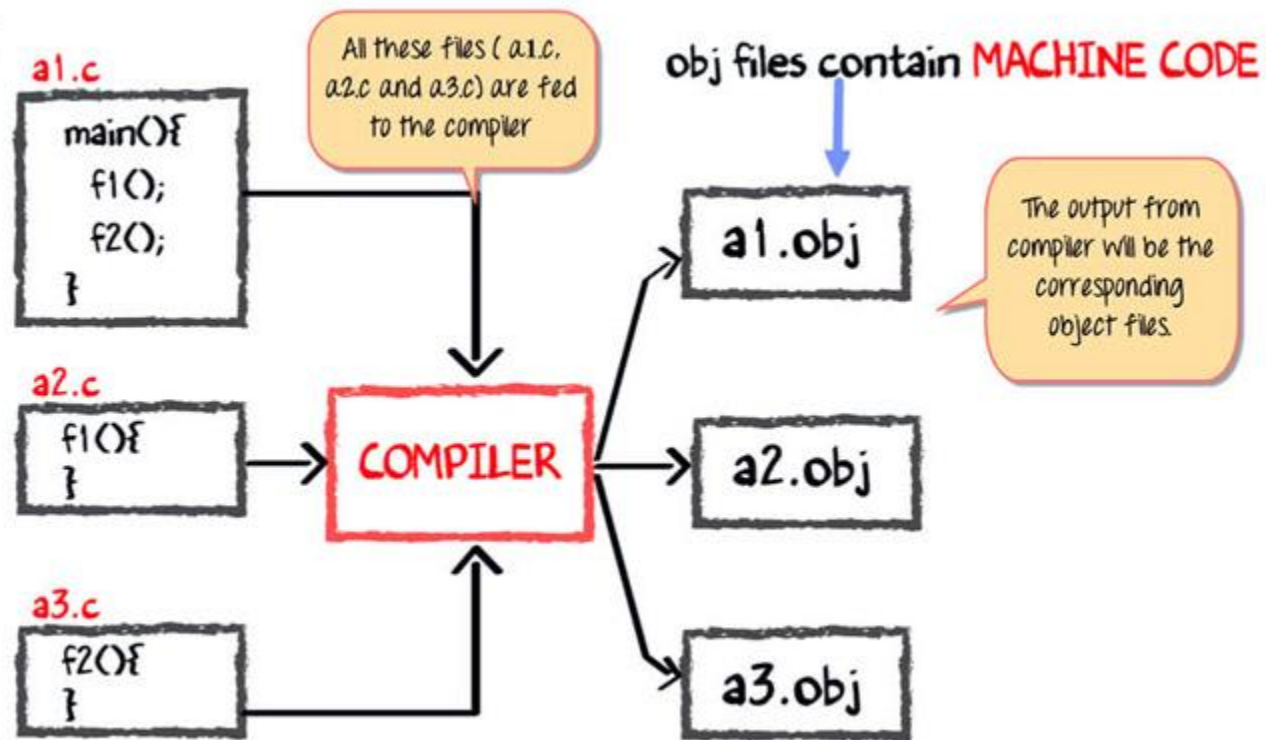
```
main(){  
    f1();  
    f2();  
}
```

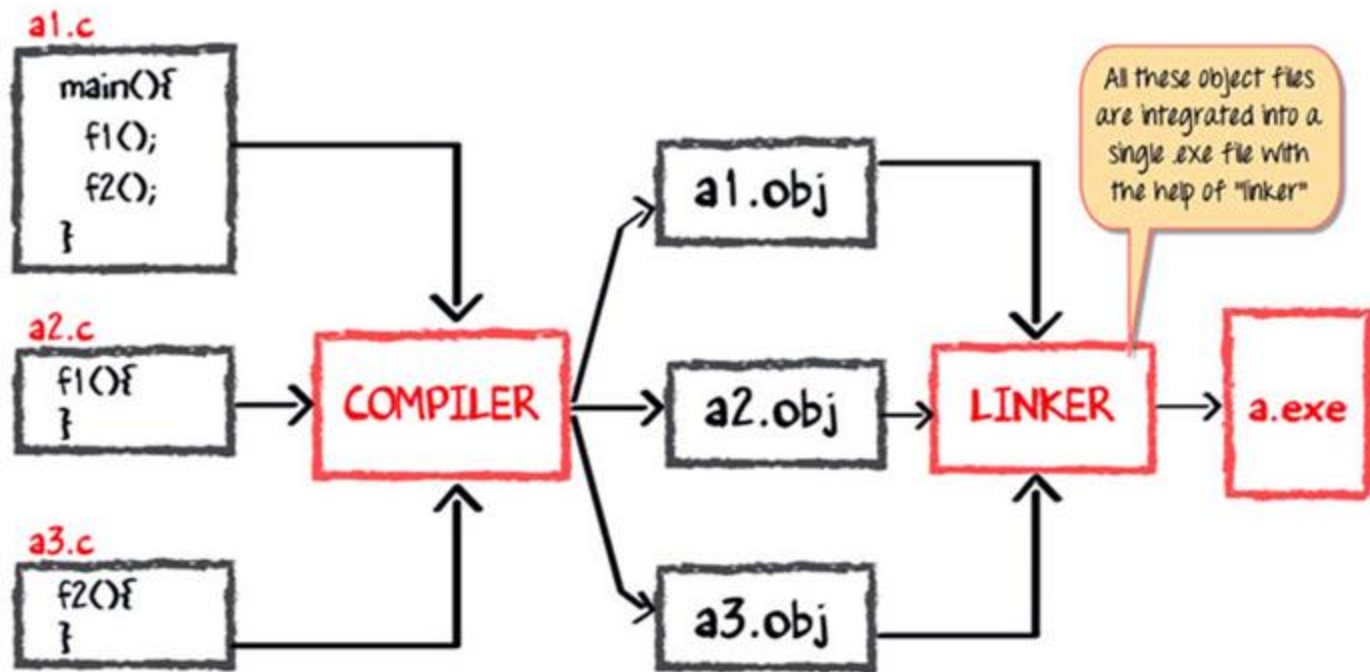
a2.c

```
f1(){  
}
```

a3.c

```
f2(){  
}
```

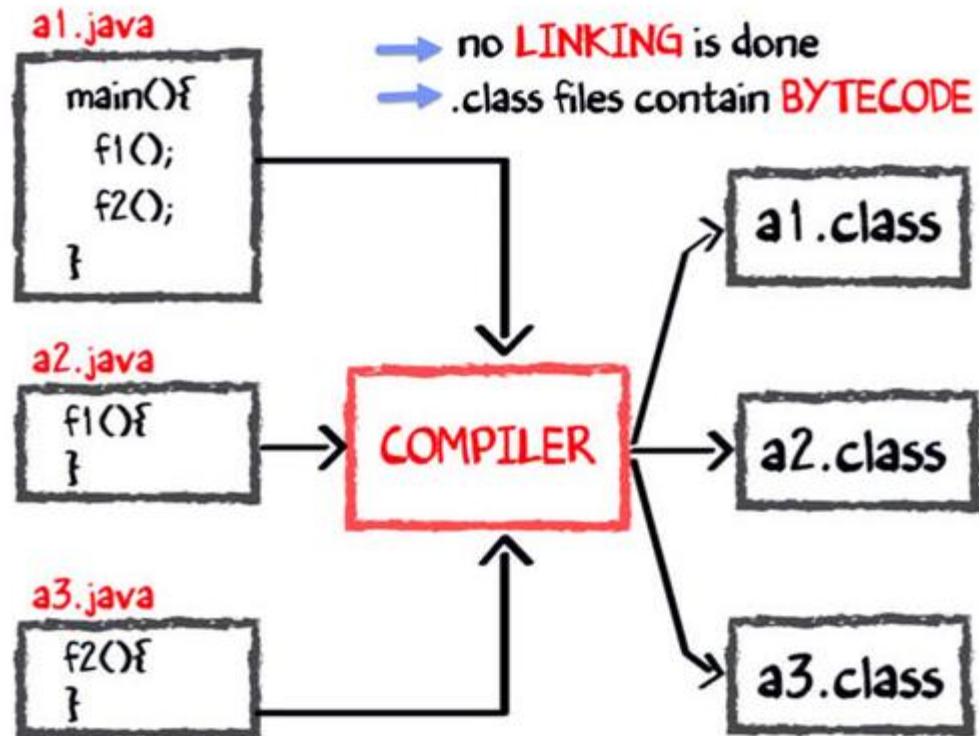




RAM

a.exe

.exe file
gets
executed in
RAM



JAVA ARCHITECTURE

The .class files will be brought on to JVM using class loader

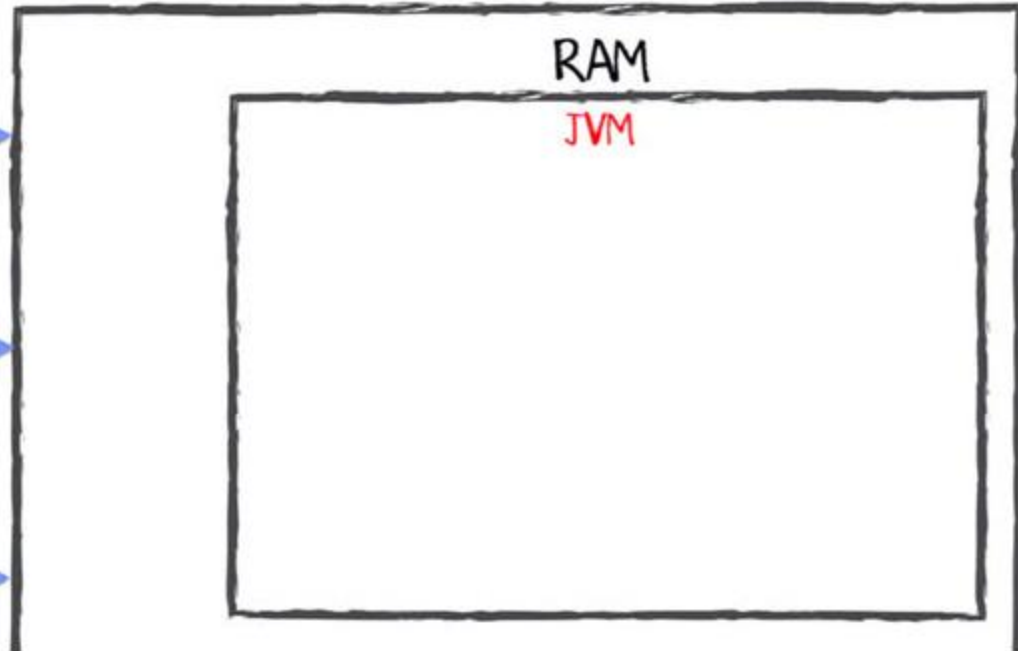
a1.class

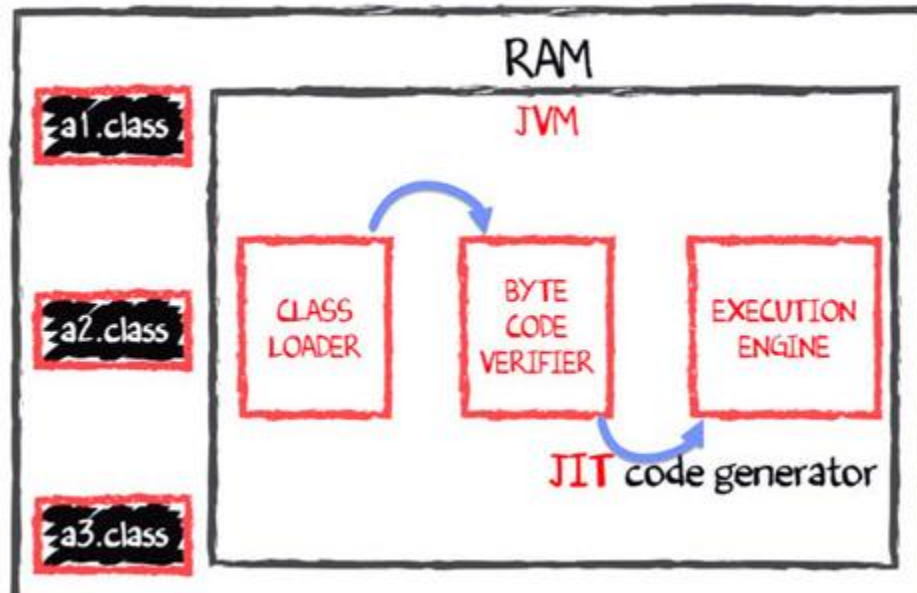


a2.class

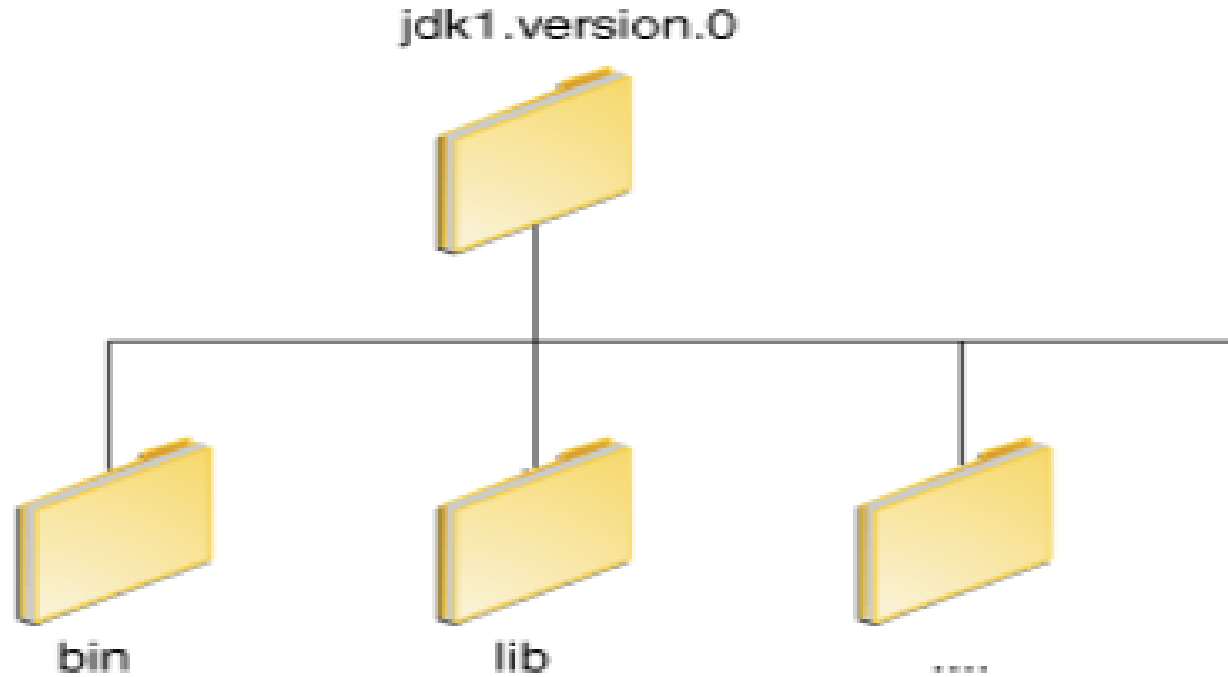


a3.class





Path and Classpath



Path

- You can run Java applications just fine without setting the PATH environment variable.
- Set the PATH environment variable if you want to be able to conveniently run the executables (javac.exe, java.exe, javadoc.exe, and so on) from any directory without having to type the full path of the command.

Example

- C:/> C:\Java\jdk1.7.0\bin\javac
MyClass.java

OR

- C:/> javac MyClass.java

Classpath

- The CLASSPATH variable is one way to tell applications, including the JDK tools, where to look for user classes.
 - -cp command switch
 - e.g. C:/> javac -cp . MyClass.java

Setting the path and classpath temporarily

- SET PATH=C:\Program Files\Java\jdk_version\bin
- SET CLASSPATH=C:\Program Files\Java\jdk_version\bin

Documentation Section	←	Suggested
Package Statement	←	Optional
Import Statements	←	Optional
Interface Statements	←	Optional
Class Definitions	←	Optional
<pre> Main Method Class { Main Method Definition } </pre>	←	Essential

General structure of a Java program

JAVA

Lexical Issues

- White space
- Identifiers
- Literals
- Comments
- Separators
- Keywords

White space

Whitespace in java may be

- A space
- A new line
- A tab

Identifiers

- Used for naming variable names, method names and class names
- Should **not start with a number** (as it confuse with a number) **or there should not be any hyphen – in any identifiers.**
- Can start with
 - _ (underscore)
 - \$ (dollar symbol)
 - Uppercase or lowercase alphabets

Examples

Audi	count	a7	\$good	this_is_ok
------	-------	----	--------	------------

Literals

It is a constant created in Java.

88	76.5	'A'	"I am a string"
----	------	-----	-----------------

88 is an integer

76.5 is a float

'A' is a character

"I am a string" is a string

Comments

- Three types
 - Single line comment (Similar like C++)
 - `//`
 - Multiline comment (similar like C++)
 - `/*`
 - `---`
 - `---`
 - `*/`
 - Documentation comment (useful for producing HTML documents)
 - `/**`
 - `-----`
 - `-----`
 - `*/`

Separators

Symbol	Name	Purpose
()	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Keywords (totally there are 50)

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

DATA TYPES

Primitive/Simple types

- Integers
- Floating point numbers
- Characters
- Booleans

Integers

Name	Width
long	64
int	32
short	16
byte	8

Range	
-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	long
-2,147,483,648 to 2,147,483,647	int
-32,768 to 32,767	short
-128 to 127	byte

Floating point type

Name	Width in Bits	Approximate Range
double	64	4.9e−324 to 1.8e+308
float	32	1.4e−045 to 3.4e+038

Characters

Name	Width in Bits	Range
Char	16	0 to 65,535

Booleans

Name	Width in Bits	Range
boolean	1(Not Fixed)	true or false

boolean: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

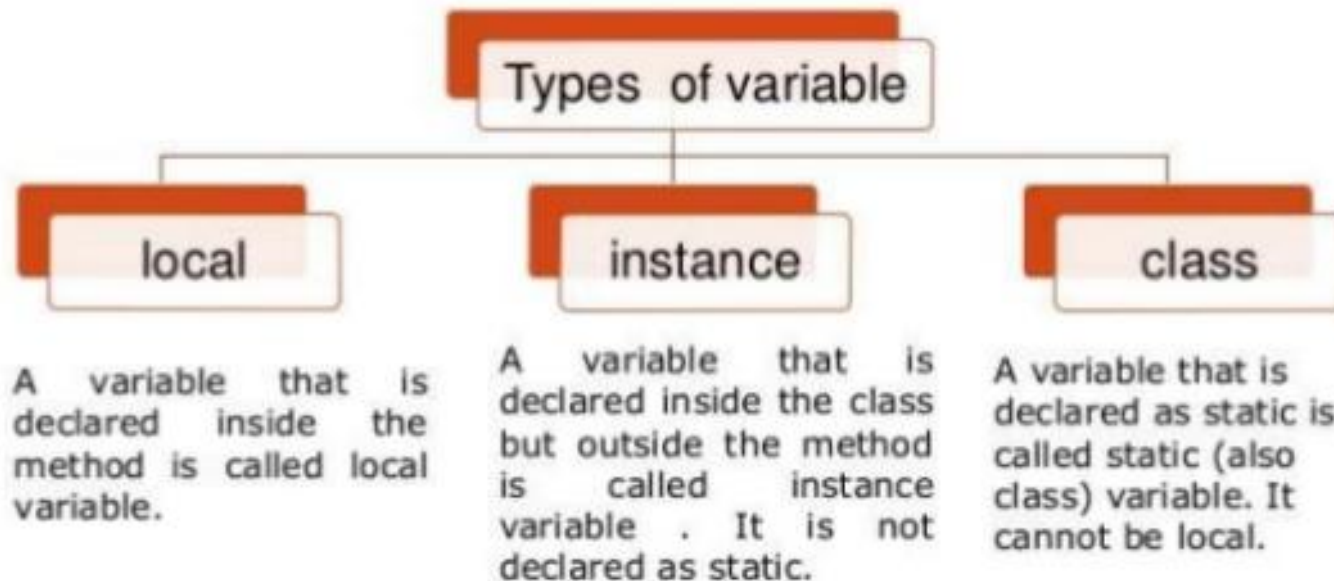
Default Values

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Character escape sequences

Escape Sequence	Description
<code>\ddd</code>	Octal character (ddd)
<code>\uxxxx</code>	Hexadecimal Unicode character (xxxx)
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\\</code>	Backslash
<code>\r</code>	Carriage return
<code>\n</code>	New line (also known as line feed)
<code>\f</code>	Form feed
<code>\t</code>	Tab
<code>\b</code>	Backspace

Variables



• Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Variable Types

- Here's a method that declares a local variable named `i`, and then initializes the variable before using it:

```
public static void main(String[] args)
{
    int i; //declare local variable
    i = 0; //initializing the local variable
    System.out.println("i is " + i);
}
```

OR

```
public static void main(String[] args)
{
    int i=0; //declare and initializing local variable in single line
    System.out.println("i is " + i);
}
```

Variable Types

- When you declare more than one variable in a single statement, each variable can have its own initializer:

```
int x = 5, y = 10;
```

- When you declare two class or instance variables in a single statement but use only one initializer, the initializer applies only to the last variable in the list.
- **For Example:**

```
int x, y = 5; //Here, only y is initialized.
```

Variable Types

- Declare local variables within blocks of code marked by braces. For example:

```
if (taxRate > 0) {  
    double taxAmount; //Declare local variable  
    taxAmount = subTotal * taxRate;  
    total = subTotal + total;  
}
```

- Local variables are not given initial default values. Thus, you must assign a value before you use a local variable.


```
if (taxRate > 0) {  
    double taxAmount=0; //Declare and initializing local variable  
    taxAmount = subTotal * taxRate;  
    total = subTotal + total;  
}
```

Local Variables Example

```
public class Test{  
    public void age() {  
        int age = 0 ; //initializing with 0  
        age = age + 7;  
        System.out.println("Age is : " + age);  
    }  
    public static void main(String[] args) {  
        Test test = new Test(); //Creating an object  
        test.age();  
    }  
}
```

Here, *age* is a local variable. This is defined inside *age()* method and its scope is limited to only this method.

Calling age Method
with Using the Object
of Class Test



Local Variables Example

```
public class Test{  
    public void age() {  
        int age = 0 ; //initializing with 0  
        age = age + 7;  
        System.out.println("Age is : " + age);  
    }  
    public static void main(String[] args) {  
        Test test = new Test();  
        test.age();  
    }  
}
```

Output is:
Age is : 7

Local Variables Example(Some Changes)

```
public class Test{  
    public void age() {  
        int age ;  
        age = age + 7;  
        System.out.println("Age is : " + age);  
    }  
    public static void main(String[] args) {  
        Test test = new Test();  
        test.age();  
    }  
}
```

Same Program as Previous but in this Program we use Local Variable *age* Without Initializing it, so it would Throw Compile time Error

Local Variables Example(Some Changes)

```
public class Test{  
    public void age() {  
        int age ;  
        age = age + 7;  
        System.out.println("Age is : " + age);  
    }  
    public static void main(String[] args) {  
        Test test = new Test();  
        test.age();  
    }  
}
```

Same Program as Previous but in this Program we use Local Variable *age* Without Initializing it, so it would Throw Compile time Error

Compiler Error

Test.java:4:variable number might not have been initialized
age = age + 7;

Instance Variables-1

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.

Instance Variables-2

- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class.


Instance Variables

```
class Rectangle {  
    //instance variables  
    double length;  
    double breadth;  
    // This class declares an object of type Rectangle.  
    public static void main(String args[]) {  
        Example myrect = new Example();  
        double area;  
        // assign values to myrect1's instance variables  
        myrect.length = 10;  
        myrect.breadth = 10;  
        // Compute Area of Rectangle  
        area = myrect.length * myrect.breadth ;  
        System.out.println("Area of Rectangle : " + area);  
    }  
}
```


Instance Variables

```
class Rectangle {  
    //instance variables  
    double length;  
    double breadth;  
    // This class declares an object of type Rectangle.  
    public static void main(String args[]) {  
        Example myrect = new Example();  
        double area;  
        // assign values to myrect1's instance variables  
        myrect.length = 10;  
        myrect.breadth = 10;  
        // Compute Area of Rectangle  
        area = myrect.length * myrect.breadth ;  
        System.out.println("Area of Rectangle : " + area);  
    }  
}
```

Instance
variables



Instance Variables

```
class Rectangle {  
    //instance variables  
    double length;  
    double breadth;  
  
    // This class declares an object of type Rectangle.  
    public static void main(String args[]) {  
        Example myrect = new Example();  
        double area;  
        // assign values to myrect1's instance variables  
        myrect.length = 10;  
        myrect.breadth = 10;  
        // Compute Area of Rectangle  
        area = myrect.length * myrect.breadth ;  
        System.out.println("Area of Rectangle : " + area);  
    }  
}
```

Instance
variables

Using dot Operator we
can Access Instance
Variable of class.

Output is:
Area of Rectangle : 100.0s:

Class/Static Variables-1

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.
- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.

Class/Static Variables-2

- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name *ClassName.VariableName*.

Static Variables Example

```
public class Employee {  
    // salary variable is a private static variable  
    private static double salary;  
    // DEPARTMENT is a constant  
    public static final String DEPARTMENT = "Development";  
    public static void main(String args[]) {  
        salary = 1000;  
        System.out.println(DEPARTMENT + "average salary:" + salary);  
    }  
}
```

Note: If the Variables are Accessed from an Outside class, the Constant should be Accessed as Employee.DEPARTMENT

Static Variables Example

```
public class Employee {  
    // salary variable is a private static variable  
    private static double salary;  
    // DEPARTMENT is a constant  
    public static final String DEPARTMENT = "Development";  
    public static void main(String args[]) {  
        salary = 1000;  
        System.out.println(DEPARTMENT + "average salary:" + salary);  
    }  
}
```

"DEPARTMENT" is
Constant Because By
Using "Final" Keyword

Note: If the Variables are Accessed from an Outside class, the
Constant should be Accessed as Employee.DEPARTMENT

Static Variables Example

```
public class Employee {  
    // salary variable is a private static variable  
    private static double salary;  
    // DEPARTMENT is a constant  
    public static final String DEPARTMENT = "Development";  
    public static void main(String args[]) {  
        salary = 1000;  
        System.out.println(DEPARTMENT + "average salary:" + salary);  
    }  
}
```

Output is:

Development average salary:1000.0

Characteristic	Local variable	Instance variable	Class variable
<i>Where declared</i>	In a method, constructor, or block.	In a class, but outside a method. Typically private.	In a class, but outside a method. Must be declared static. Typically also final.
<i>Use</i>	Local variables hold values used in computations in a method.	Instance variables hold values that must be referenced by more than one method	Class variables are mostly used for <i>constants</i> , variables that never change from their initial value
<i>Lifetime</i>	Created when method or constructor is entered. Destroyed on exit.	Created when instance of class is created with new. Destroyed when there are no more references to enclosing object (made available for garbage collection).	Created when the program starts. Destroyed when the program stops.

Characteristic	Local variable	Instance variable	Class variable
<i>Declaration</i>	Declare before use anywhere in a method or block.	Declare anywhere at class level (before or after use).	Declare anywhere at class level with static.
<i>Initial value</i>	None. Must be assigned a value before the first use.	Zero for numbers, false for booleans, or null for object references. May be assigned value at declaration or in constructor.	Same as instance variable, and it addition can be assigned value in special static <i>initializer block</i> .
<i>Name syntax</i>	Standard rules.	Standard rules, but are often prefixed to clarify difference from local variables, eg with my, m, or m_ (for member) as in myLength, or this as in this.length.	static public final variables (constants) are all uppercase, otherwise normal naming conventions.

Characteristic	Local variable	Instance variable	Class variable
<i>Access from outside</i>	Impossible. Local variable names are known only within the method.	Instance variables should be declared private to promote information hiding, so should not be accessed from outside a class. However, in the few cases where there are accessed from outside the class, they must be qualified by an object (eg, myPoint.x).	Class variables are qualified with the class name (e.g. Color.BLUE). They can also be qualified with an object, but this is a deceptive style.

Example to Understand the Types of Variables in Java

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


Scope of Variables in Java

- The scope of a variable is the part of the program over which the variable name can be referenced. You cannot refer to a variable before its declaration.
- We can declare variables in several different places:
 - In a class body as class fields. Variables declared here are referred to as class-level variables.
 - As parameters of a method or constructor.
 - In a method's body or a constructor's body.
 - Within a statement block, such as inside a while or for block.
- Variable scope refers to the accessibility of a variable.

Scope of Variables in Java

- We can declare variables within any block.
- Block is begun with an opening curly brace and ended by a closing curly brace.
- 1 block equal to 1 new scope in Java thus each time you start a new block, you are creating a new scope.
- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Scope of Variables in Java

- **Block Level Scope**

- The variables that defined in a block are only accessible from within the block. The scope of the variable is the block in which it is defined. For example, consider the following for statement.

```
public class MainClass {  
    public static void main(String[] args) {  
        for (int x = 0; x < 5; x++) {  
            System.out.println(x);  
        }  
        //x is not accessible here  
        //System.out.println(x);  
    }  
}
```

Here, the Scope
of Variable x is
Block Level

Scope of Variables in Java

- A nested block can access variables declared in the outer block. Consider this code.

```
public class MainClass {  
    public static void main(String[] args) {  
        for (int x = 0; x < 5; x++) {  
            for (int y = 0; y < 3; y++) {  
                System.out.println(x);  
                System.out.println(y);  
            }  
            // Here Compiler Error y is not accessible in this block  
            //System.out.println(y);  
        }  
    }  
}
```

Scope of Variables in Java

- Variables declared as method parameters can be accessed from within the method body. Class-level variables are accessible from anywhere in the class.
- If a method declares a local variable that has the same name as a class-level variable, the former will 'shadow' the latter. To access the class-level variable from inside the method body, use the `this` keyword.

Scope of Variables in Java

```
// Demonstrate block scope
class Scope {
    public static void main(String args[]){
        int n1=10; // Visible in main
        if(n1 == 10)
        {
            // start new scope
            int n2 = 20; // visible only to this block
            // num1 and num2 both visible here.
            System.out.println("n1 and n2 : "+ n1 + " "+ n2);
        }
        // n2 = 100; // Error! n2 not known here
        // n1 is still visible here.
        System.out.println("n1 is " + n1);
    }
}
```

Output is:

```
n1 and n2 : 10 20
n1 is 10
```


Scope of Variables in Java

Explanation of Previous Program

- n1 is declared in main block thus it is **accessible in main block**.
- n2 is declared in if block thus it is only **accessible inside if block**.
- Any attempt to access it outside block will **cause compiler time error**.
- Nested Block can have access to its outermost block. [if block is written inside main block thus all the variables declared inside main block are accessible in if block]

Scope of Variables in Java

```
class ScopeInvalid {  
    public static void main(String args[]) {  
        int num = 1;  
        {           // creates a new scope  
            int num = 2; // Compile-time error  
            // num already defined  
        }  
    }  
}
```

Here Compile Error
Because Variable "num" is
Declared in main Scope
and thus it is Accessible to
all the Innermost Blocks.

```
class ScopeValid {  
    public static void main(String args[]) {  
        {           // creates a new scope  
            int num = 1;  
        }  
        {           // creates a new scope  
            int num = 2;  
        }  
    }  
}
```


Scope of Variables in Java

• Re-Initializing Same Variable Again and Again

```
class Scope {  
    public static void main(String args[]) {  
        int i;  
        for(i = 0; i < 3; i++) {  
            int y = -1;  
            System.out.println("y is : " + y);  
        }  
    }  
}
```

Output is:

```
y is : -1  
y is : -1  
y is : -1
```

- Variable "y" is declared inside for loop block.
- Each time when control goes inside "For loop Block", Variable is declared and used in loop.
- When control goes out of the for loop then the variable becomes inaccessible.