

Introduction to Java

Introduction

What is Java?

- ✓ Java is a widely used object-oriented programming language and software platform that runs on billions of devices, including notebook computers, mobile devices, gaming consoles, medical devices and many others.
- ✓ Its rules and syntax of Java are based on the C and C++ languages.
- ✓ One major advantage of developing software with Java is its portability.
- ✓ Once you have written code for a Java program on a notebook computer, it is very easy to move the code to a mobile device.
- ✓ When the language was invented in 1991 by James Gosling of Sun Microsystems.

Java terminology....

□ Java Virtual Machine (JVM)

- ✓ The phases of program execution as follows: we write the program, then we compile the program and at last we run the program.
 - 1) Writing of the program is of course done by java programmer like you and me.
 - 2) Compilation of program is done by javac compiler, javac is the primary java compiler included in java development kit (JDK). It takes java program as input and generates java bytecode as output.
 - 3) In third phase, JVM executes the bytecode generated by compiler. This is called program run phase.
- ✓ So, now that we understood that the primary function of JVM is to execute the bytecode produced by compiler.
- ✓ Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems.
- ✓ That is why we call java as platform independent language.

Java terminology....

□ Java Development Kit(JDK)

- JDK stands for Java Development Kit. It is a software development environment used to develop Java applications and applets.
- It is a platform-specific software i.e there are separate installers for Windows, Mac, and Unix systems.
- It contains Java Runtime Environment(JRE) and other development tools like an interpreter, compiler, archiver, and a document generator.

□ Java Runtime Environment(JRE)

- JRE stands for Java Runtime Environment.
- It is the implementation of JVM and it is specially designed to provide an environment to execute Java programs.
- It is also platform dependent like JDK. It consists of JVM, Java

rogram.



Popular Java IDEs

- ✓ IDEs typically provide a code editor, a compiler or interpreter and a debugger that the developer accesses through a unified graphical user interface (GUI).
- ✓ A Java IDE is an integrated development environment for programming in Java; many also provide functionality for other languages.
- ✓ Few popular Java IDEs:
 - NetBeans (*Support: C/C++, PHP etc.*)
 - Eclipse (*C, C++, C#, Fortran, JavaScript*)

A Simple Java Program....

✓ The requirement for Java

For executing any Java program, the following software or application must be properly installed:

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. C:\Program Files (x86)\Java\jdk1.6.0_01\bin. *(Computer -> System Properties -> Advanced system settings -> Environment Variables -> System variables -> PATH)*
- Create the Java program.
- Compile and run the Java program.

✓ Creating Welcome to Java! Example

```
class Simple{  
    public static void main(String args[]){  
        System.out.println("Welcome to java!");  
    }  
}
```

A Simple Java Program.....

✓ **To Save:**

Save the above file as Simple.java.

✓ **To Compile:**

`javac Simple.java.`

✓ **To Run:**

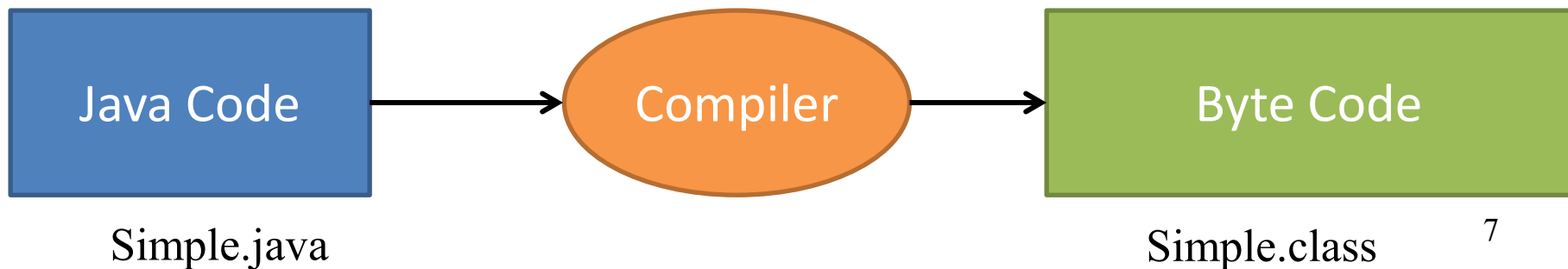
`java Simple.`

✓ **Output:**

Welcome to Java!.

✓ **Compilation Flow:**

When we compile Java program using javac tool, the Java compiler converts the source code into byte code:



A Simple Java Program....

✓ Parameters used in First Java Program

What is the meaning of class, public, static, void, main, String[], System.out.println()?

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args or String args[]** is used for command line argument. It means an array of sequence of characters (Strings) that are passed to the "main" function.
- **System.out.println()** is used to print statement.

Java Static Method

✓ Example: Static method accessed directly in static and non-static method.

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object.

```
class JavaExample{
    static int i = 100;
    static String s = "Dept. of ICE";
    //Static method
    static void display()
    {
        System.out.println("i:"+i);
        System.out.println("s:"+s);
    }
    //non-static method
    void funcn()
    {
        //Static method called in non-static
        method
        display(); }
}
```

```
//static method
public static void main(String args[])
{
    JavaExample obj = new JavaExample();
    //You need to have object to call this non-static
    method
    obj.funcn();
    //Static method called in another static method
    display();
}
}
```

Output:

```
i:100
s: Dept. of ICE
i:100
s: Dept. of ICE
```

Abstract Class in Java....

- ✓ A class that is declared using “**abstract**” keyword is known as abstract class.
- ✓ It can have abstract methods(methods without body) as well as concrete methods (regular methods with body).
- ✓ An abstract class can not be instantiated, which means you are not allowed to create an object of it. Because these classes are incomplete, they have abstract methods that have no body so java does not allow you to create object of this class.

What is a real life example of abstract class in Java?

Suppose we have a class **Animal** that has a method **sound()** and the subclasses(inheritance) of it like **Dog, Lion, Horse, Cat** etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like **Lion** class will say “**Roar**” in this method and **Dog** class will say “**Woof**”.

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method(otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the Animal class has an abstract method, you must need to declare this class abstract.

Abstract Class in Java....

✓ Abstract class Example:

```
//abstract parent class
abstract class Animal{
    //abstract method
    public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{
    public void sound(){
        System.out.println("Woof");
    }
    public static void main(String args[])
    {
        Animal obj = new Dog();
        obj.sound();
    } }
```

Output: Woof

Some key points of abstract method:

1. Abstract method has no body.
2. Always end the declaration with a semicolon(;).
3. It must be **overridden**. An abstract class must be extended and in a same way abstract method must be overridden.
4. A class has to be declared abstract to have abstract methods.
5. An abstract class has no use until unless it is extended by some other class.
6. If you declare an abstract method in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
7. It can have non-abstract method (concrete) as well.

Interface Class in Java....

What is an interface in Java?

- ✓ Interface looks like a class but it is not a class.
- ✓ An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method).
- ✓ Also, the variables declared in an interface are public, static & final by default.

What is the use of interface in Java?

- ✓ They are used for full abstraction.
- ✓ The methods in interfaces do not have body, they have to be implemented by the class before you can access them.
- ✓ The class that implements interface must implement all the methods of that interface.
- ✓ Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class. **Example:** *interface A { int x=10;} interface B { int x=100; } class Hello implements A,B { statements..... }.*

Interface Class in Java....

✓ Interface class Example:

```
interface MyInterface {  
    /* compiler will treat them as:  
    * public abstract void method1();  
    * public abstract void method2(); */  
    public void method1();  
    public void method2(); }  
class Demo implements MyInterface {  
    /* This class must have to implement both  
    the abstract methods  
    * else you will get compilation error */  
    public void method1() {  
        System.out.println("implementation  
of method1"); }  
    public void method2() {  
        System.out.println("implementation  
of method2"); }
```

```
public static void main(String arg[]) {  
    MyInterface obj = new Demo();  
    obj.method1(); } }
```

Output: implementation of method1

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation. Because interfaces are used in Java to achieve abstraction.
2. In java, multiple inheritance is not allowed, however you can use interface to make use of it as you can implement more than one interface.

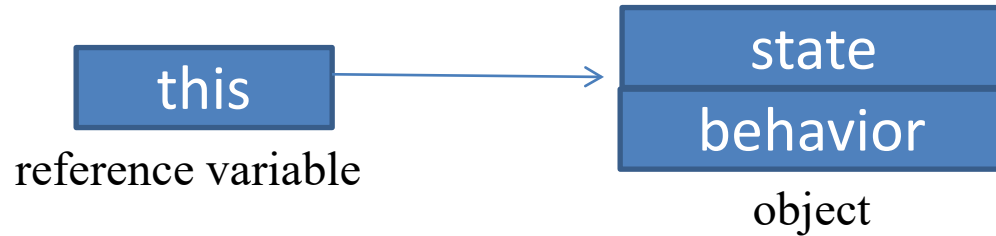
Abstract Vs Interface

✓ Difference between Abstract Class and Interface in Java:

	Abstract Class	Interface
1	An abstract class can extend only one class or one abstract class at a time	An interface can extend any number of interfaces at a time
2	An abstract class can extend another concrete (regular) class or abstract class	An interface can only extend another interface
3	An abstract class can have both abstract and concrete methods	An interface can have only abstract methods
4	In abstract class keyword “abstract” is mandatory to declare a method as an abstract	In an interface keyword “abstract” is optional to declare a method as an abstract
5	An abstract class can have protected and public abstract methods	An interface can have only have public abstract methods
6	An abstract class can have static, final or static final variable with any access specifier	An interface can only have public static final (constant) variable ₁₄

this keyword in Java

In Java, this is a **reference variable** that refers to the current object.



Here is given the 6 usage of java this keyword:

- 1) this can be used to refer current class instance variable.
- 2) this can be used to invoke current class method (implicitly).
- 3) `this()` can be used to invoke current class constructor.
- 4) this can be passed as an argument in the method call.
- 5) this can be passed as argument in the constructor call.
- 6) this can be used to return the current class instance from the method.

See details
only for
known

this: to refer current class instance variable

- ✓ If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Problem if we don't use this keyword by the example given below: class Student{

```
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+"
"+fee);}
}
class TestThis1 {
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

Solution of the above problem by this keyword: class Student{

```
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+"
"+fee);}
}
```

```
class TestThis2 {
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```


this: to refer current class instance variable

- ✓ If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program.

Problem if we don't use this keyword by the example given below: class Student{

```
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+name+"
"+fee);}
}
class TestThis1 {
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
0 null 0.0
0 null 0.0
```

Program where **this** keyword is **not** required: class Student{

```
int rollno;
String name;
float fee;
Student(int r,String n,float f){
rollno=r;
name=n;
fee=f;
}
void display(){System.out.println(rollno+" "+name+"
"+fee);}
}
```

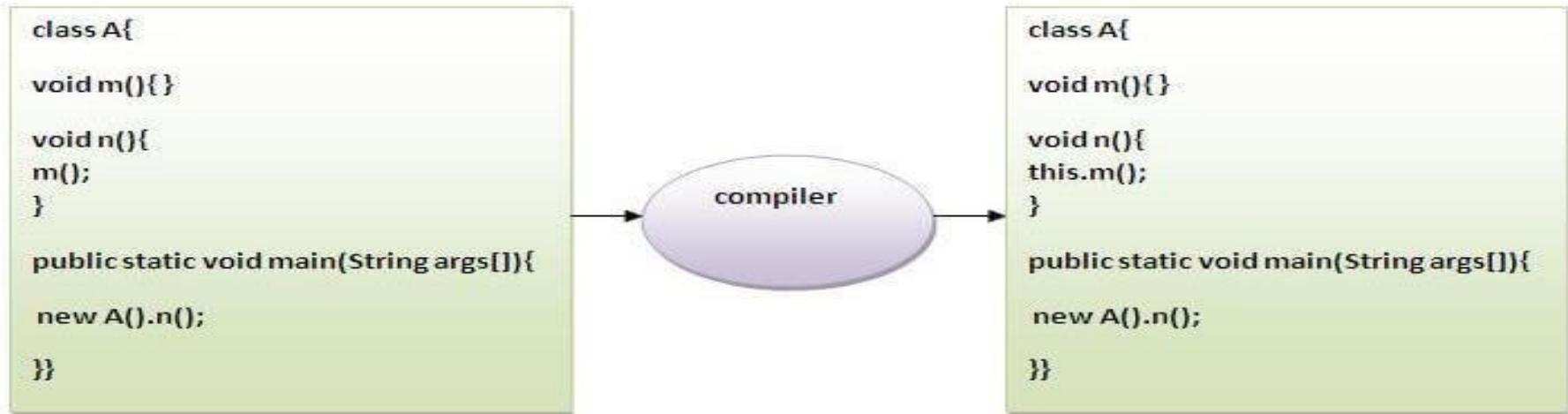
```
class TestThis3 {
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

this: to invoke current class method

- ✓ If you don't use the **this** keyword, compiler automatically adds **this** keyword while invoking the method. An Example given below:

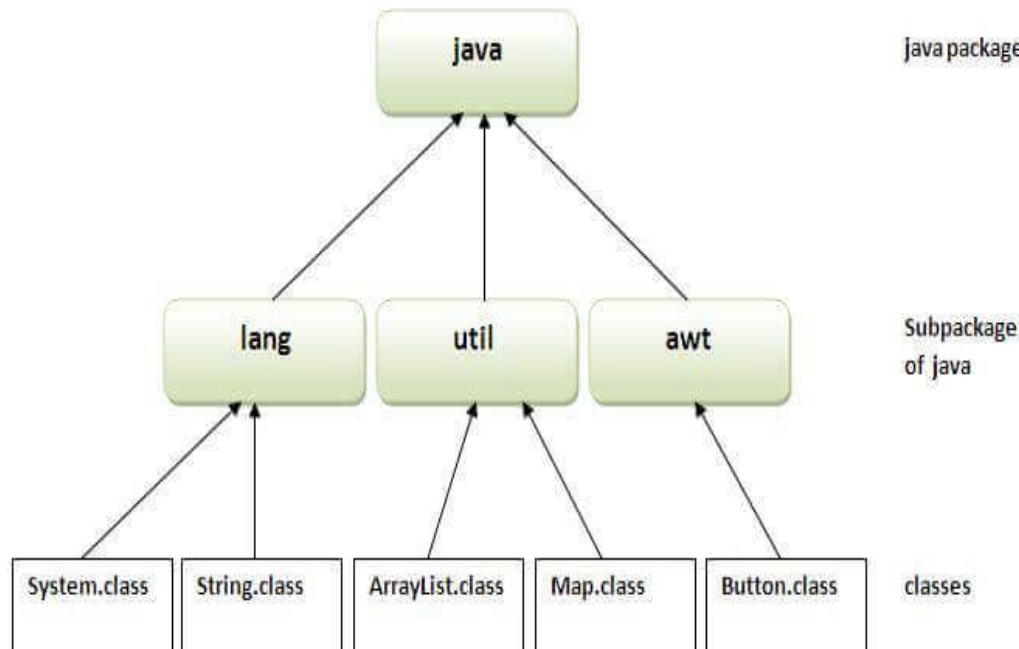


```
class A {
void m()
{System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();//same as this.m()
this.m();
} }
```

```
class TestThis4 {
public static void main(String args[])
{ A a=new A();
a.n();
}}
Output:
hello n
hello m
```

Java Package....

- ✓ A java package is a group of similar types of classes, interfaces and sub-packages.
- ✓ Package in java can be categorized in two form, built-in package and user-defined package.
- ✓ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.



Advantage of Java Package:

- ✓ 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- ✓ 2) Java package provides access protection.
- ✓ 3) Java package removes naming collision.

Java Package....

Simple Example of java Package without using any IDE.

✓ To create :

The package keyword is used to create a package in java:

```
//save as Simple.java  
package mypack;  
public class Simple{  
    public static void main(String args[]){  
        System.out.println("Welcome to package");  
    }  
}
```

✓ To compile:

If you are not using any IDE, you need to follow the syntax given below:

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file.

✓ To Run:

You need to use fully qualified name e.g. mypack.Simple etc to run the class:

```
java mypack.Simple
```

✓ Output: *Welcome to package*

Java Package....

How to access package from another package?

There are three ways to access the package from outside the package.

1. **import package.*;**
2. **import package.classname;**
3. **fully qualified name.**

1) Using **packagename.***

- ✓ If you use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.
- ✓ The import keyword is used to make the classes and interface of another package accessible to the current package.

2) Using **packagename.classname**

- ✓ If you import **package.classname** then only declared class of this package will be accessible.

3) Using **fully qualified name**

- ✓ If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
- ✓ It is generally used when two packages have same class name e.g. **java.util** and **java.sql** packages contain **Date** class.

Java Package....

Example of all packages :

1) Using packagename.*

```
//save by A.java
package pack; //first package
public class A{
    public void msg(){System.out.println("Hello");} }
```

```
//save by B.java
package mypack; //second package
import pack.*; // import first package
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg(); } }
```

2) Using packagename.classname

```
//save by A.java
package pack; ; //first package
public class A{
    public void msg(){System.out.println("Hello");} }
```

```
//save by B.java
package mypack; //second package
import pack.A; // import first package class
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg(); } }
```

3) Using fully qualified name

```
//save by A.java
package pack; ; //first package
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A(); //using fully
qualified name
        obj.msg();
    }
}
```

Java Platform Security Architecture

The inception of Java technology a strong and growing interest around the security of the Java platform as well as new security issues raised by the deployment of Java technology.

✓ **Java security includes two aspects:**

- Provide the Java platform as a secure, ready-built platform on which to run Java-enabled applications in a secure fashion.
- Provide security tools and services implemented in the Java programming language that enable a wider range of security-sensitive applications, for example, in the enterprise world.

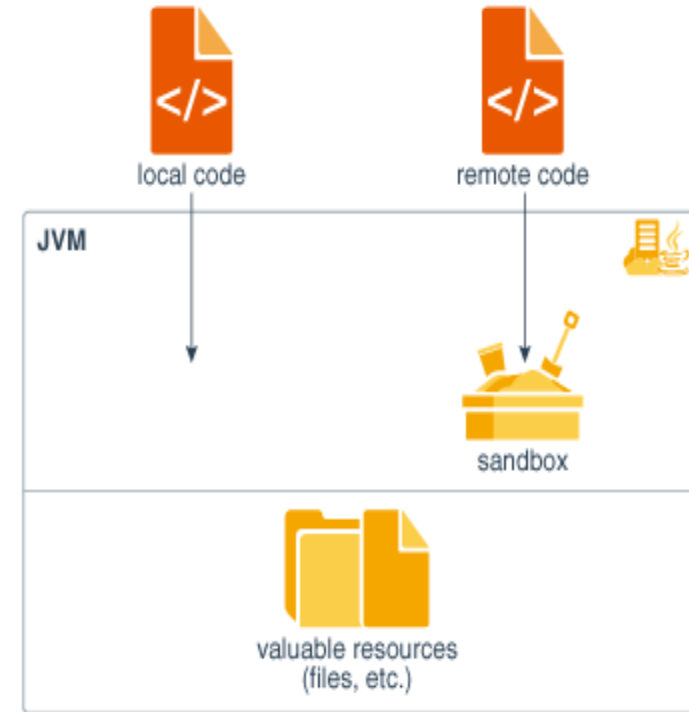
✓ **The Original Sandbox Model**

- The original security model provided by the Java platform is known as the **sandbox model**, which existed in order to provide a very restricted environment in which to run untrusted code obtained from the open network.
- The essence of the sandbox model is that local code is trusted to have full access to vital system resources (such as the file system) while downloaded remote code (an applet) is not trusted and can access only the limited resources provided inside the sandbox.

Java Platform Security Architecture

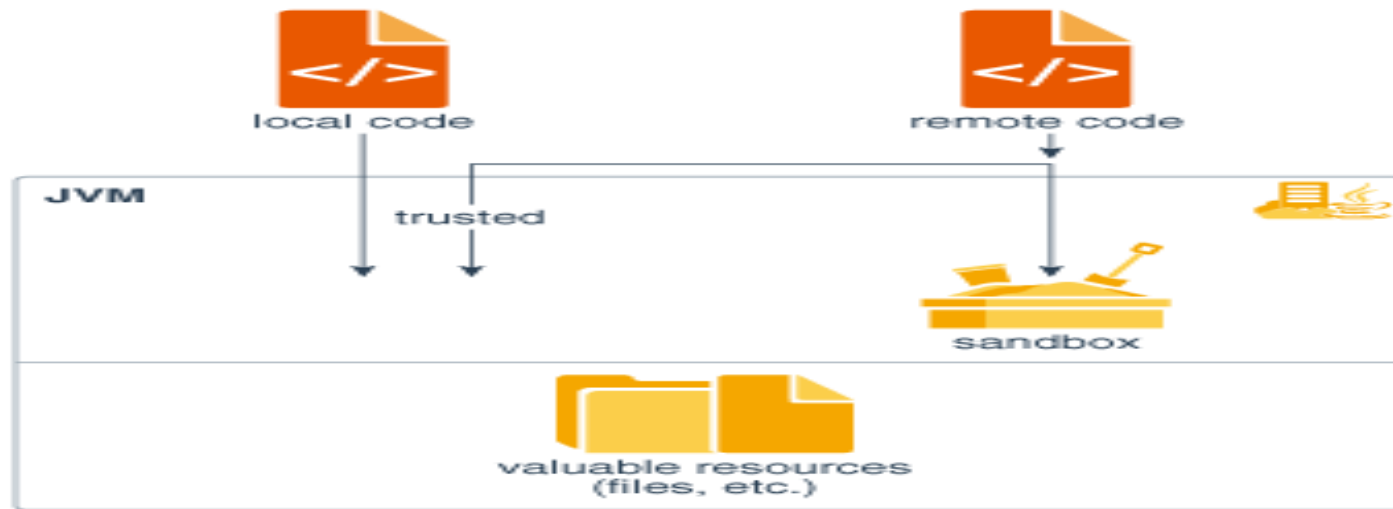
Original Java Platform Security Model:

- ✓ The sandbox model was deployed through the Java Development Kit (JDK), and was generally adopted by applications built with JDK 1.0, including Java-enabled web browsers.
- ✓ Overall security is enforced through a number of mechanisms:
 - **First of all**, the language is designed to be type-safe and easy to use. Language features such as automatic memory management, garbage collection, and range checking on strings and arrays are examples of how the language helps the programmer to write safe code.
 - **Second**, compilers and a bytecode **verifier** ensure that only legitimate Java bytecodes are executed. The bytecode verifier, together with the Java Virtual Machine, guarantees language safety at run time. Moreover, a **classloader** defines a local name space, which can be used to ensure that an untrusted applet cannot interfere with the running of other programs.
 - **Finally**, access to crucial system resources is mediated by the Java Virtual Machine and is checked in advance by a **SecurityManager** class that restricts the actions of a piece of untrusted code to the bare minimum.

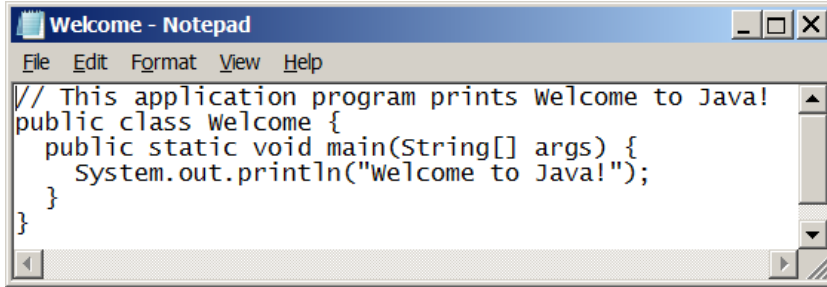


Java Platform Security Architecture

JDK 1.1 Security Model:



- JDK 1.1 introduced the concept of a "signed applet", as illustrated by the figure.
- A correctly digitally signed applet is treated as if it is trusted local code if the signature key is recognized as trusted by the end system that receives the applet.
- Signed applets, together with their signatures, are delivered in the JAR (Java Archive) format. In JDK 1.1, unsigned applets still run in the sandbox.



```
// This application program prints Welcome to Java!
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Source code (developed by the programmer)

```
public class Welcome {
    public static void main(String[] args) {
        System.out.println("Welcome to Java!");
    }
}
```

Bytecode (generated by the compiler for JVM to read and interpret)

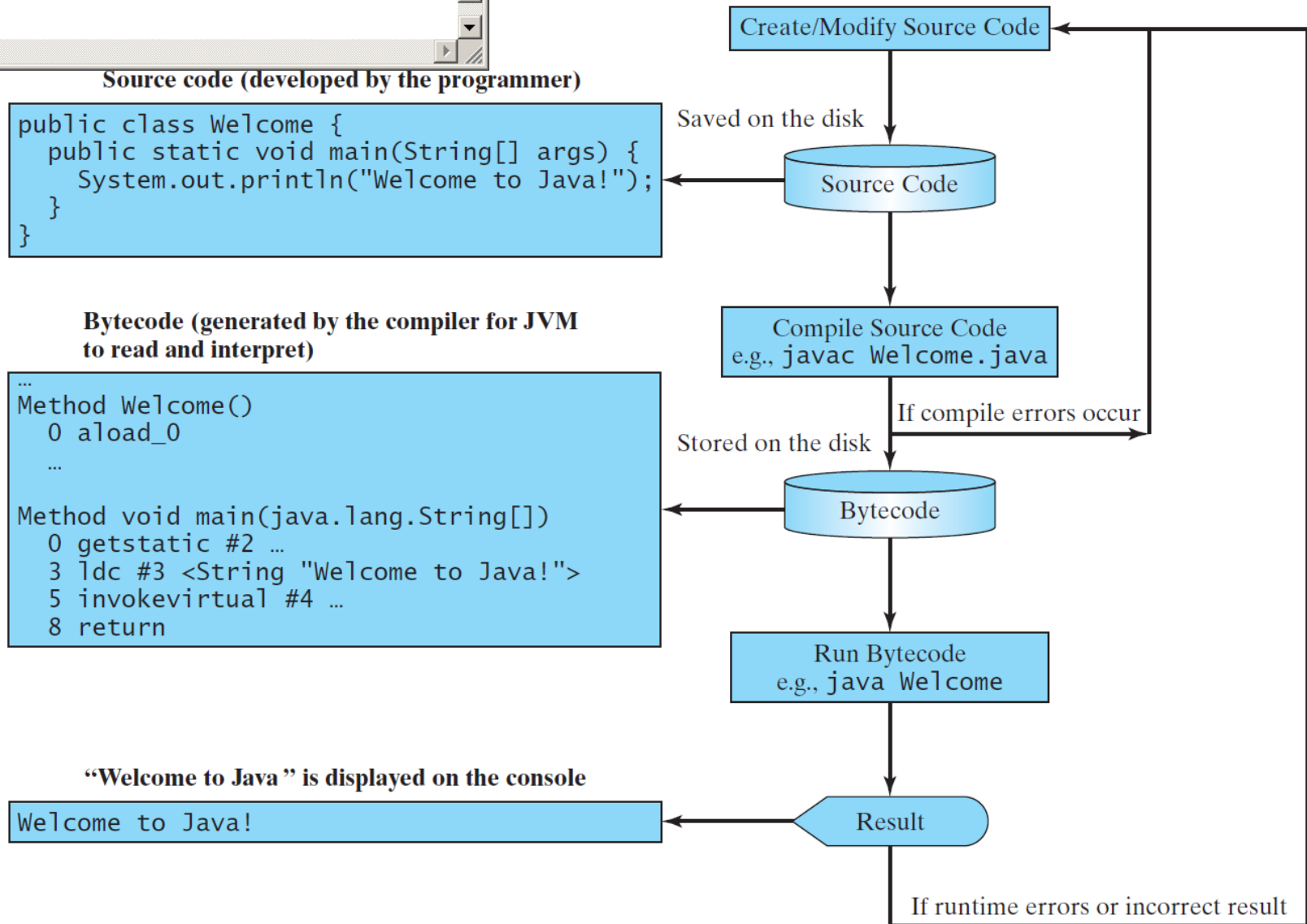
```
...
Method Welcome()
  0 aload_0
  ...

Method void main(java.lang.String[])
  0 getstatic #2 ...
  3 ldc #3 <String "Welcome to Java!">
  5 invokevirtual #4 ...
  8 return
```

“Welcome to Java” is displayed on the console

Welcome to Java!

Creating, Compiling, and Running Programs



Compiling Java Source Code

You can port a source program to any machine with appropriate compilers. The source program must be recompiled, however, because the object program can only run on a specific machine. Nowadays computers are networked to work together. Java was designed to run object programs on any platform. With Java, you write the program once, and compile the source program into a special type of object code, known as *bytecode*. The bytecode can then run on any computer with a Java Virtual Machine, as shown below. Java Virtual Machine is a software that interprets Java bytecode.

