JAVA PROGRAMMING

Introduction

Jyoti Lakhani

What is Java

- A general purpose Programming Language
- Most Famous Object Oriented Programming Language.
- Suited for the web and networked services, applications, platform independent desktops, robotics, embedded devices.

Java No Longer a Programming Language but a Platform

History of Java

- Developed by James Gosling, Mike Sheridan, Patrick Naughton et. al;
- When working on green project;
- The goal of this project was to control microprocessor embedded in electronic devices.
- To perform this task, a platform independent, reliable, and compact language was needed

History of Java

- Gosling was given task to identify the proper programming language for the project.
- Gosling decided to develop a new language to avoid the problems of current programming languages(C++ especially)
- The new language developed for this purpose is called “Oak”

History of Java

It consist of –

- Oak Programming Language
- An Operating System
- An User Interface
- A Hardware

Java Versions

- JDK 1.0 1996
- JDK 1.1 1997
- JDk 1.2 1998 (Java 2)
- JDK 1.3 2000 (Java 2)
- JDK 1.4 2002 (Java 2)
- JDK 5.0 2004 (J2SE 5.0)
- JDK 6.0 2006 (Java SE 6)
- JDK 7.0 2010

Different Language for different Platform...

With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms.

Ex: J2EE for Enterprise Applications

J2ME for Mobile Applications.

- Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME respectively.

- Java is guaranteed to be **Write Once, Run Anywhere.**

Java Features

Object Oriented

- Java, everything is an Object.
- Java can be easily extended since it is based on the Object model.

Platform independent

- Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.

Simple

- Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.

Secure

- With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.

Architectural-neutral

- Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

Portable

- Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable.

- Compiler in Java is written in ANSI C

Robust

- Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

Multithreaded

- With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.

Interpreted

- Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.

High Performance

- With the use of Just-In-Time compilers, Java enables high performance

Distributed

- Java is designed for the distributed environment of the internet

Dynamic

- Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

Java Environment

Java™

Java Environment

JDK

(Java Development Environment)

A set of Programming Tools

JSL

(Java Standard Library)

(A collection of Classes and Methods)

It is also called Java API

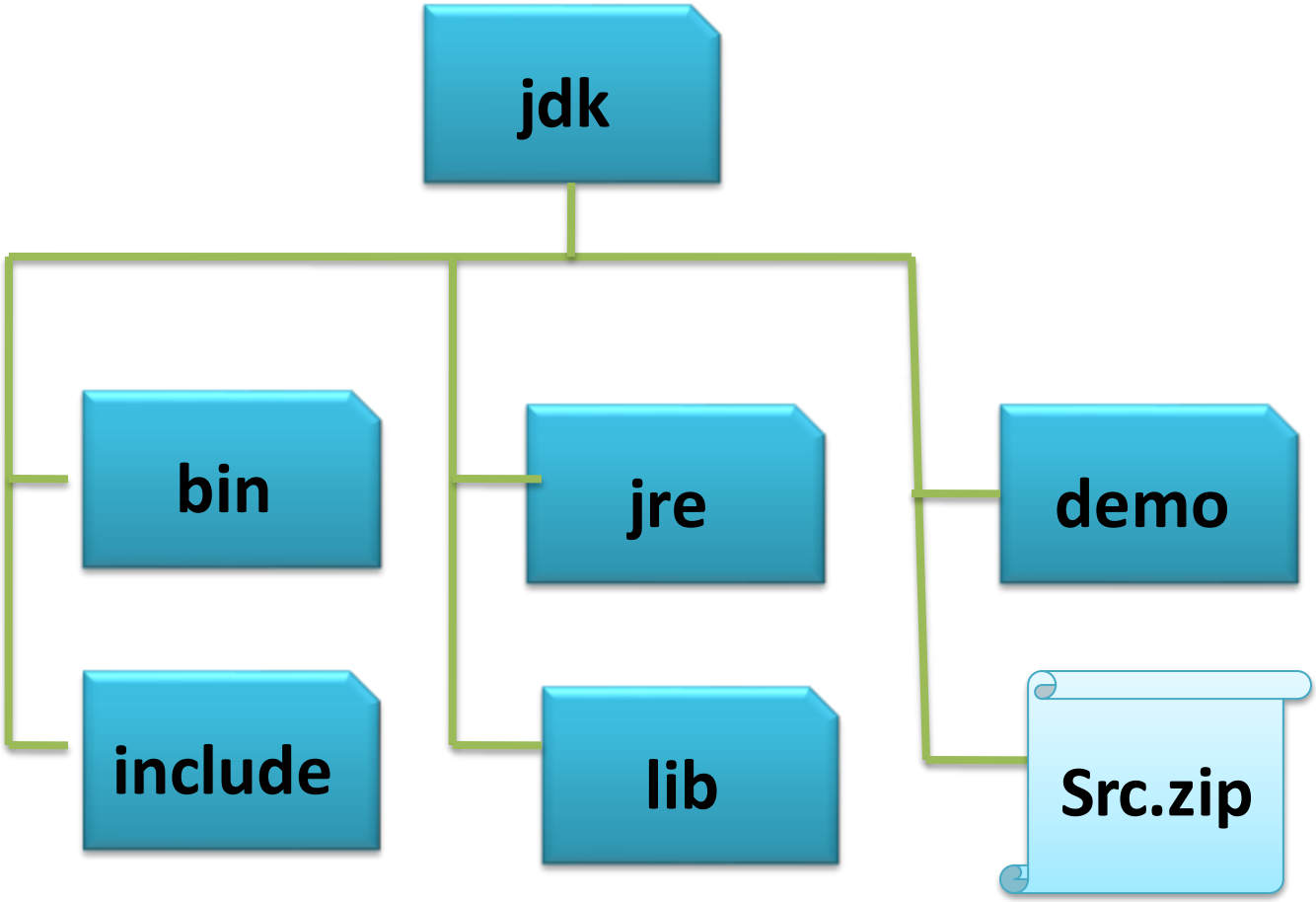
Java Development Kit (JDK)

- A Software Package
- Sun Microsystems made it available free on the website
- Includes all basic components of Java Environment

Java Development Kit (JDK) - Tools

- Java Compiler
- Java Interpreter
- Applet Viewer
- Utilities
- Documentation
- Sample Code

Java Development Kit
File Structure(C:\jdk)



Java Development Kit
bin

- A collection of utilities that help u to develop, execute, debug and document java programs
- Include-
  - *javac.exe*
  - *java.exe*
  - *javadoc.exe*
  - *appletviewer.exe*

Java Development Kit
include

- Contains files that support native code programming
- integrate java with other programming languages.
- Include-
  - *jawet.h*
  - *jni.h*

Java Development Kit
jre

- Root directory of Java Run Time Environment
- Includes
  - JVM(Java Virtual Machine)
  - runtime class libraries
  - java application launcher

Java Development Kit
lib

- Includes additional class libraries and support files required by the development tools
  - *htmlconvertor*
  - *jconsole*

Java Development Kit
demo

- Consist of sample programs with source code

Java Development Kit
src.zip

- Compressed files contain source code for java API classes(that are included in JDK)
- We can unpack these files.
- The source code in these files helps developers to learn and use the java programming language

JDK Tools

- Java Compiler
- The Runtime Interpreter
- Applet Viewer
- Debugger
- Class File Disassembler
- Header and Stub File Generator
- Documentation Generator

Java Compiler (javac)

- A command line tool that reads java source code files and compiles it into executable byte code classes.

Syntax `javac [Options] [File Name]`

Example- `Javac Hello.java`

Java

Java Runtime interpreter (java)

- Java programs are compiled to byte code and not in native code.
- Hence it can not be run on the machine directly.

- Java Runtime Interpreter implements Java Virtual Machine(JVM) and runs java applications.

Java Runtime interpreter (java)

- It is used to run java .class file by doing following two things-
  - Starting Java Runtime Environment(JRE)
  - Loading the specified class
  - Invoking the main() method

Java Runtime interpreter (java)

Syntax `java [ Options ] [ Class Name ] [ Arguments ]`

Example- `Java Hello`

Applet Viewer

- A command line tool
- Used to run applet without the need of a web browser

Java Debugger(jdb)

- Command line tool helps
- Used to find out and fixes bugs in java programs

jdb [Options] [Class Name] [Arguments]

Class File Disassembler (javap)

- A command line tool
- Reads java bytecode class files
- Print human readable version of API defined by those classes.

Header and Stub File Generator(javah)

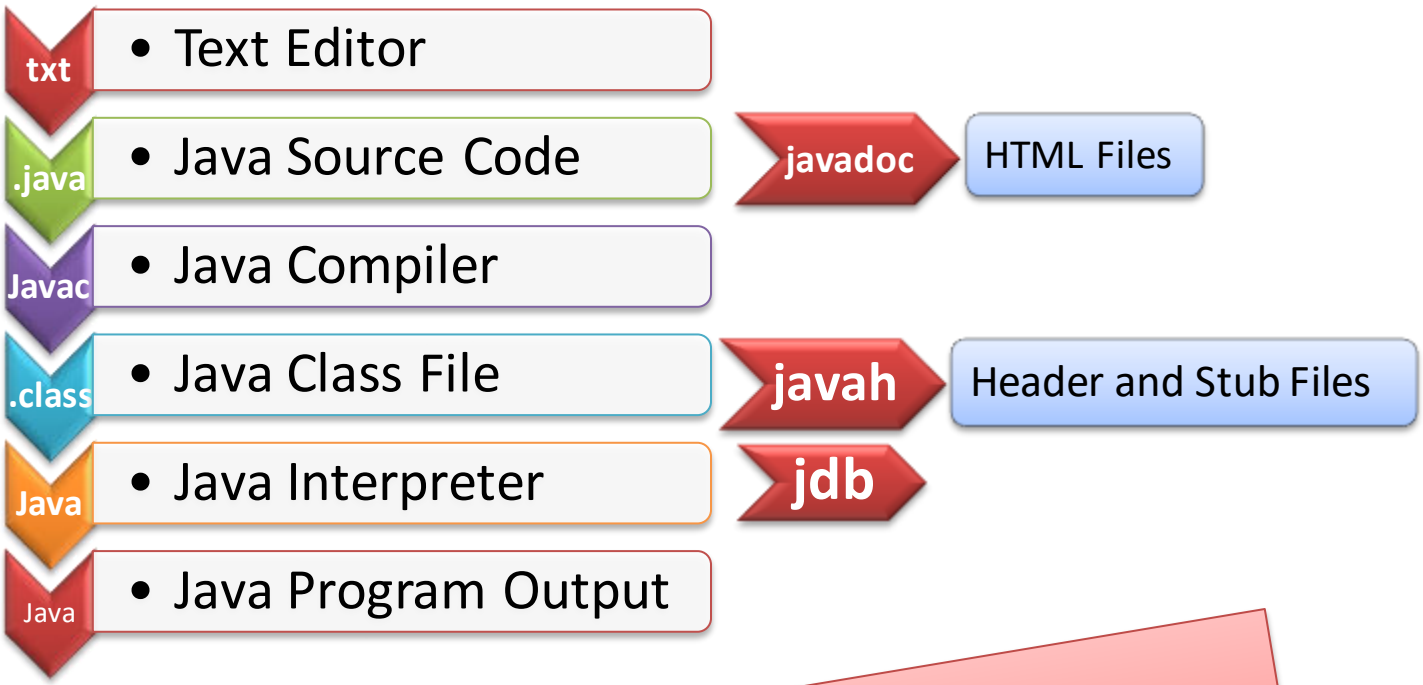
- A command line tool
- Produces two c files
  - C Header file
  - C stub file
- These C files are used to implement native methods

Header and Stub File Generator(javah)

- The header file contains the declaration for the C implementation function and structure definition
- Stub file provide the glue that binds the java method invocations and object references to the C code
- The header file name is appended with .h
- The stub file name is appended with .c

Documentation Generator(javadoc)

- A command line tool
- Used to generate documentation in the form of HTML pages



The Overall Process

Java Virtual Machine (JVM)

Introduction to the JVM

- A Component of Java System
- It interprets and executes the instructions in our class files.

Introduction to the JVM

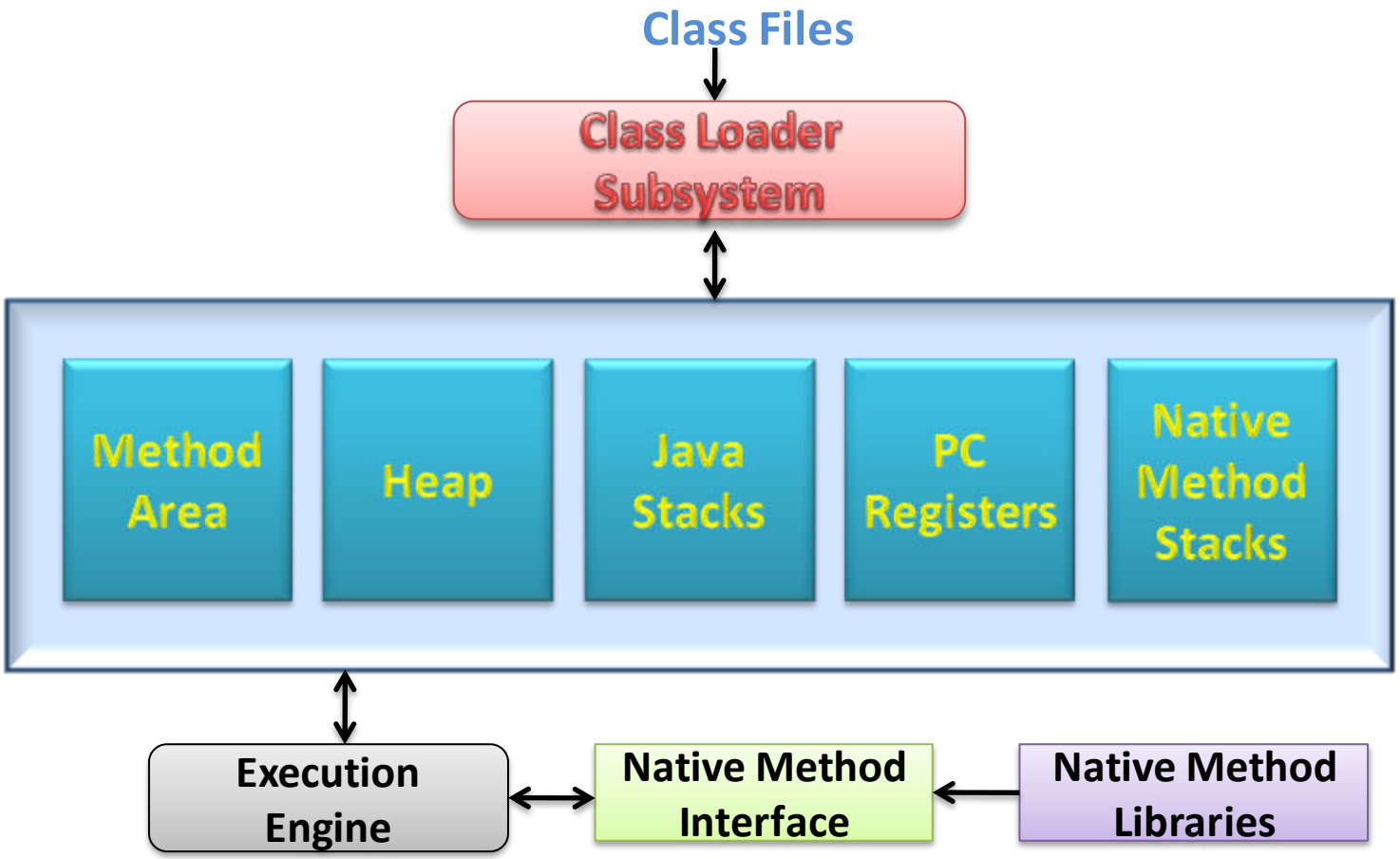


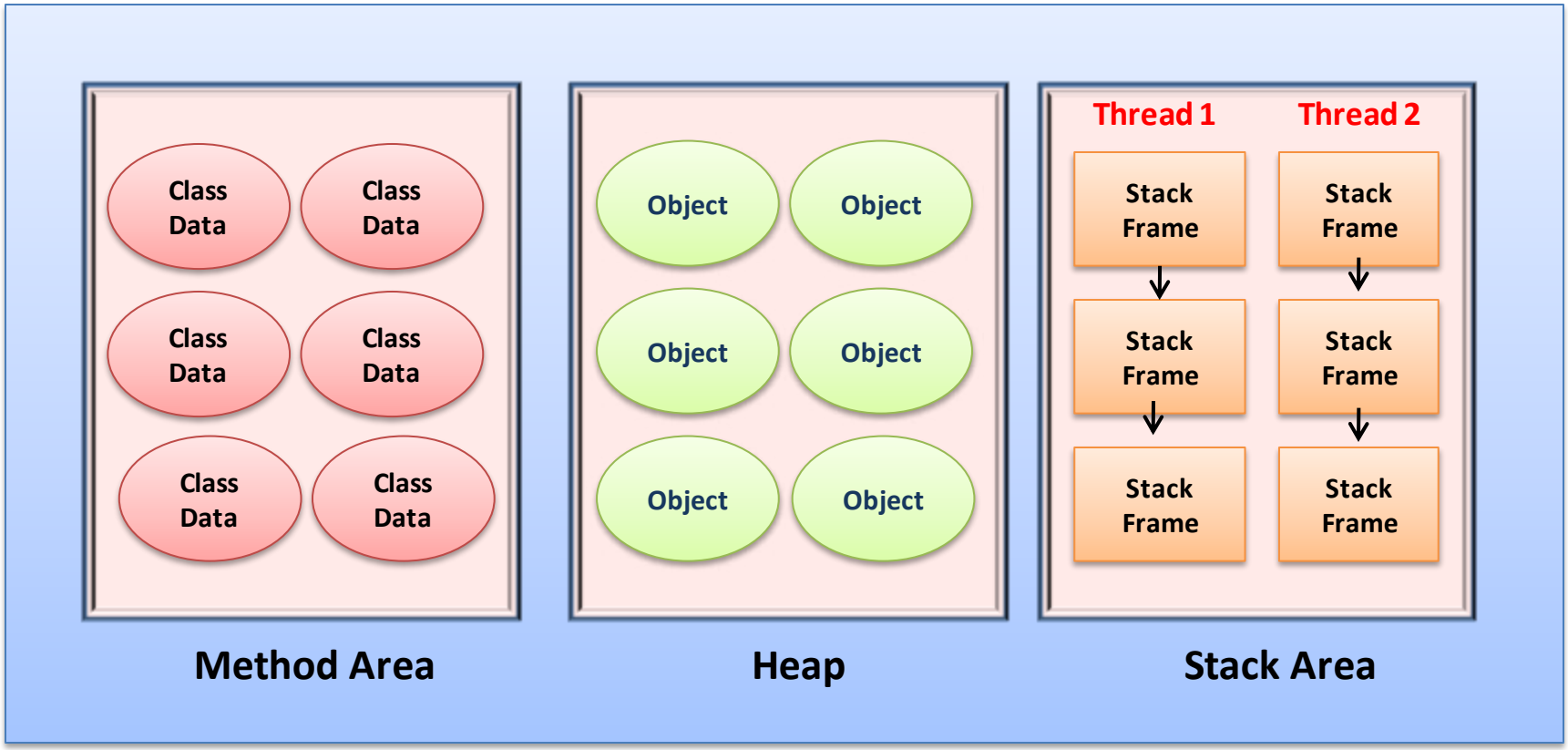
Figure 1: Memory configuration by the JVM.

Java™

Introduction to the JVM

- Each instance of the JVM has one method area, one heap, and one or more stacks - one for each thread
- When JVM loads a class file, it puts its information in the method area
- As the program runs, all objects instantiated are stored in the heap
- The stack area is used to store activation records as a program runs

Memory Blocks at Runtime



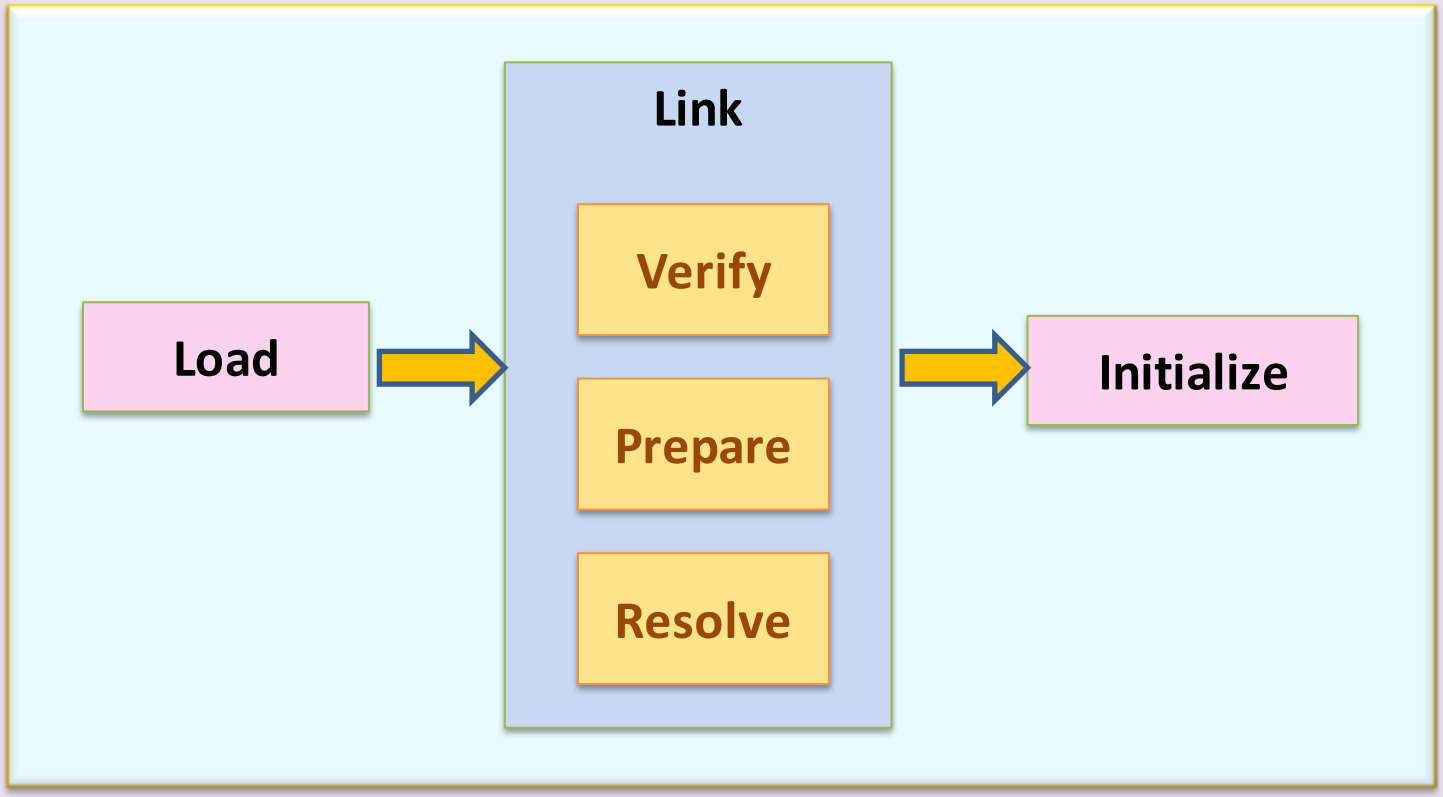
Content of Memory Blocks at Runtime

Java™

The Class Loader Subsystem

- The class loader performs three main functions of JVM, namely:
  - **Loading**
  - **Linking**
  - **Initialization**
- The linking process consists of three sub-tasks, namely,
  - **Verification**
  - **Preparation**
  - **Resolution**

The Class Loader Subsystem



Class Loading Process

Class Loading Process

- Loading means reading the class file for a type, parsing it to get its information, and storing the information in the method area.
- For each type it loads, the JVM must store the following information in the method area:
  - **The fully qualified name of the type**
  - **The fully qualified name of the type's direct superclass or if the type is an interface, a list of its direct super interfaces .**
  - **Whether the type is a class or an interface**
  - **The type's modifiers (public, abstract, final, etc)**
  - **Constant pool for the type: constants and symbolic references.**
  - **Field info : name, type and modifiers of variables (not constants)**
  - **Method info: name, return type, number & types of parameters, modifiers, bytecodes, size of stack frame and exception table.**

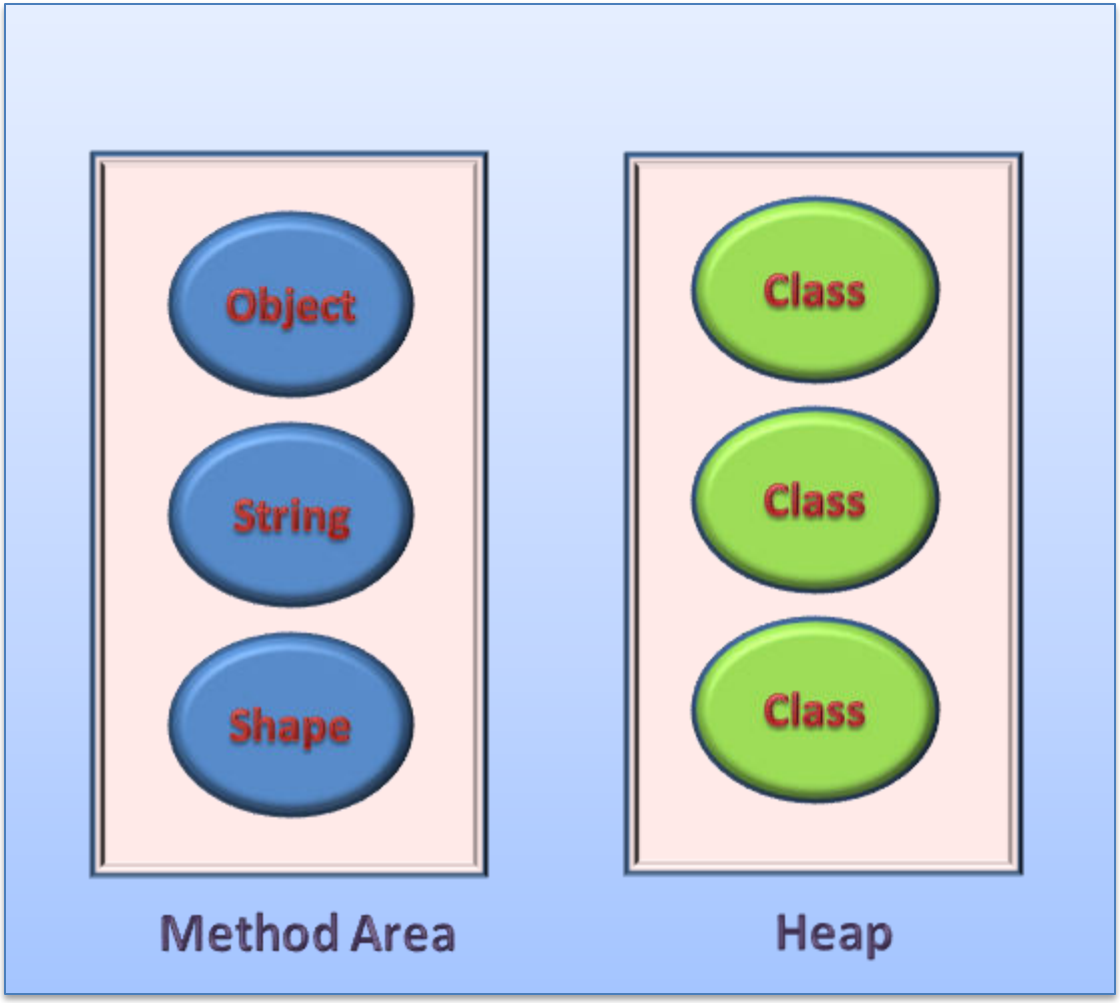
Class Loading Process

- The end of the loading process is the creation of an instance of java.lang.Class for the loaded type.
- The purpose is to give access to some of the information captured in the method area for the type, to the programmer.
- Some of the methods of the class java.lang.Class are:

```
public String getName()
public Class getSupClass()
public boolean isInterface()
public Class[] getInterfaces()
public Method[] getMethods()
public Fields[] getFields()
public Constructor[] getConstructors()
```

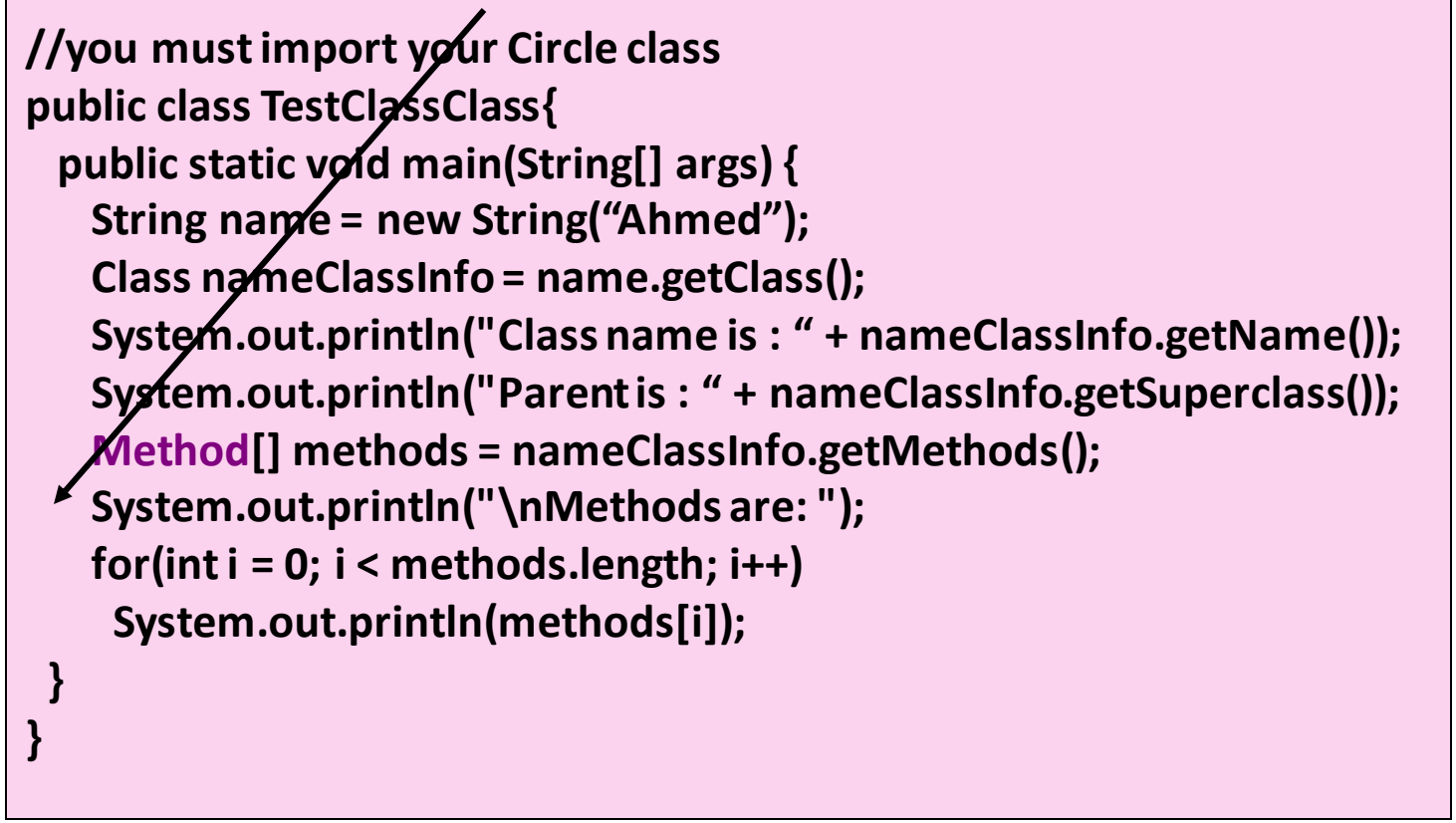
- Note that for any loaded type T, only one instance of java.lang.Class is created even if T is used several times in an application.
- To use the above methods, we need to first call the getClass() method on any instance of T to get the reference to the Class instance for T.

Class Loading Process



Instances of Class objects created in the heap at runtime

Class Loading Process



```
import java.lang.reflect.Method; // Required!

//you must import your Circle class
public class TestClassClass{
  public static void main(String[] args) {
    String name = new String("Ahmed");
    Class nameClassInfo = name.getClass();
    System.out.println("Class name is : " + nameClassInfo.getName());
    System.out.println("Parent is : " + nameClassInfo.getSuperclass());
    Method[] methods = nameClassInfo.getMethods();
    System.out.println("\nMethods are: ");
    for(int i = 0; i < methods.length; i++)
      System.out.println(methods[i]);
  }
}
```

Class Linking Process

Verification During Linking Process

- The next process handled by the class loader is Linking. This involves three sub-processes: Verification, Preparation and Resolution
- Verification is the process of ensuring that binary representation of a class is structurally correct
- The JVM has to make sure that a file it is asked to load was generated by a valid compiler and it is well formed
- Class B may be a valid sub-class of A at the time A and B were compiled, but class A may have been changed and re-compiled
- Example of some of the things that are checked at verification are:
  - Every method is provided with a structurally correct signature
  - Every instruction obeys the type discipline of the Java language
  - Every branch instruction branches to the start not middle of another instruction

Preparation

- In this phase, the JVM allocates memory for the class (i.e static) variables and sets them to default initial values.
- Note that class variables are not initialized to their proper initial values until the initialization phase - no java code is executed until initialization.
- The default values for the various types are shown below:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

Resolution

- Resolution is the process of replacing symbolic names for types, fields and methods used by a loaded type with their actual references.
- Symbolic references are resolved into a direct references by searching through the method area to locate the referenced entity.
- For the class below, at the loading phase, the class loader would have loaded the classes: TestClassClass, String, System and Object.

```
public class TestClassClass{
  public static void main(String[] args){
    String name = new String(“Ahmed”);
    Class nameClassInfo = name.getClass();
    System.out.println("Parent is: “ + nameClassInfo.getSuperclass());
  }
}
```

- lassClass.
- In this phase, the names are replaced with their actual references.

Class Initialization

- This is the process of setting class variables to their proper initial values - initial values desired by the programmer.

```
class Example1 {
        static double rate = 3.5;
        static int size = 3*(int)(Math.random()*5);
        ...
}
```

- Initialization of a class consists of two steps:
  - Initializing its direct superclass (if any and if not already initialized)
  - Executing its own initialization statements
- The above imply that, the first class that gets initialized is Object.
- Note that static final variables are not treated as class variables but as constants and are assigned their values at compilation.

```
class Example2 {
        static final int angle = 35;
        static final int length = angle * 2;
        ...
}
```

Class Initialization (Cont'd)

- After a class is loaded, linked, and initialized, it is ready for use. Its static fields and static methods can be used and it can be instantiated.
- When a new class instance is created, memory is allocated for all its instance variables in the heap.
- Memory is also allocated recursively for all the instance variables declared in its super class and all classes up is inheritance hierarchy.
- All instance variables in the new object and those of its superclasses are then initialized to their default values.
- The constructor invoked in the instantiation is then processed according to the rules shown on the next page.
- Finally, the reference to the newly created object is returned as the result.

Class Instantiation

Rules for processing a constructor:

1. Assign the arguments for the constructor to its parameter variables.
2. If this constructor begins with an explicit invocation of another constructor in the same class (using this), then evaluate the arguments and process that constructor invocation recursively.
3. If this constructor is for a class other than Object, then it will begin with an explicit or implicit invocation of a superclass constructor (using super). Evaluate the arguments and process that superclass constructor invocation recursively.
4. Initialize the instance variables for this class with their proper values.
5. Execute the rest of the body of this constructor.

Class Instantiation: Example 1

```
class GrandFather {
        int grandy = 70;
        public GrandFather(int grandy) {
                this.grandy = grandy;
                System.out.println("Grandy: "+grandy);
        }
}
class Father extends GrandFather {
        int father = 40;
        public Father(int grandy, int father) {
                super(grandy);
                this.father = father;
                System.out.println("Grandy: "+grandy+" Father: "+father);
        }
}
class Son extends Father {
        int son = 10;
        public Son(int grandy, int father, int son) {
                super(grandy, father);
                this.son = son;
                System.out.println("Grandy: "+grandy+" Father: "+father+" Son: "+son);
        }
}
public class Instantiation {
        public static void main(String[] args) {
                Son s = new Son(65, 35, 5);
        }
}
```

Output:

```
Grandy: 65
Grandy: 65 Father: 35
Grandy: 65 Father: 35 Son: 5
```