

Core Java with Java 8 Features

Objectives

- | | | | |
|---|----------------------------|---|--------------------|
| 1 | Java Overview | 5 | Class Types |
| 2 | Java Language Fundamentals | 6 | Exception Handling |
| 3 | OOPs Concepts in Java | 7 | Assertions |
| 4 | Object Oriented Concepts | 8 | Collections |

9

Multithreading

10

Java Database Connectivity

11

Java Features

12

13

14

15

16

Java Overview

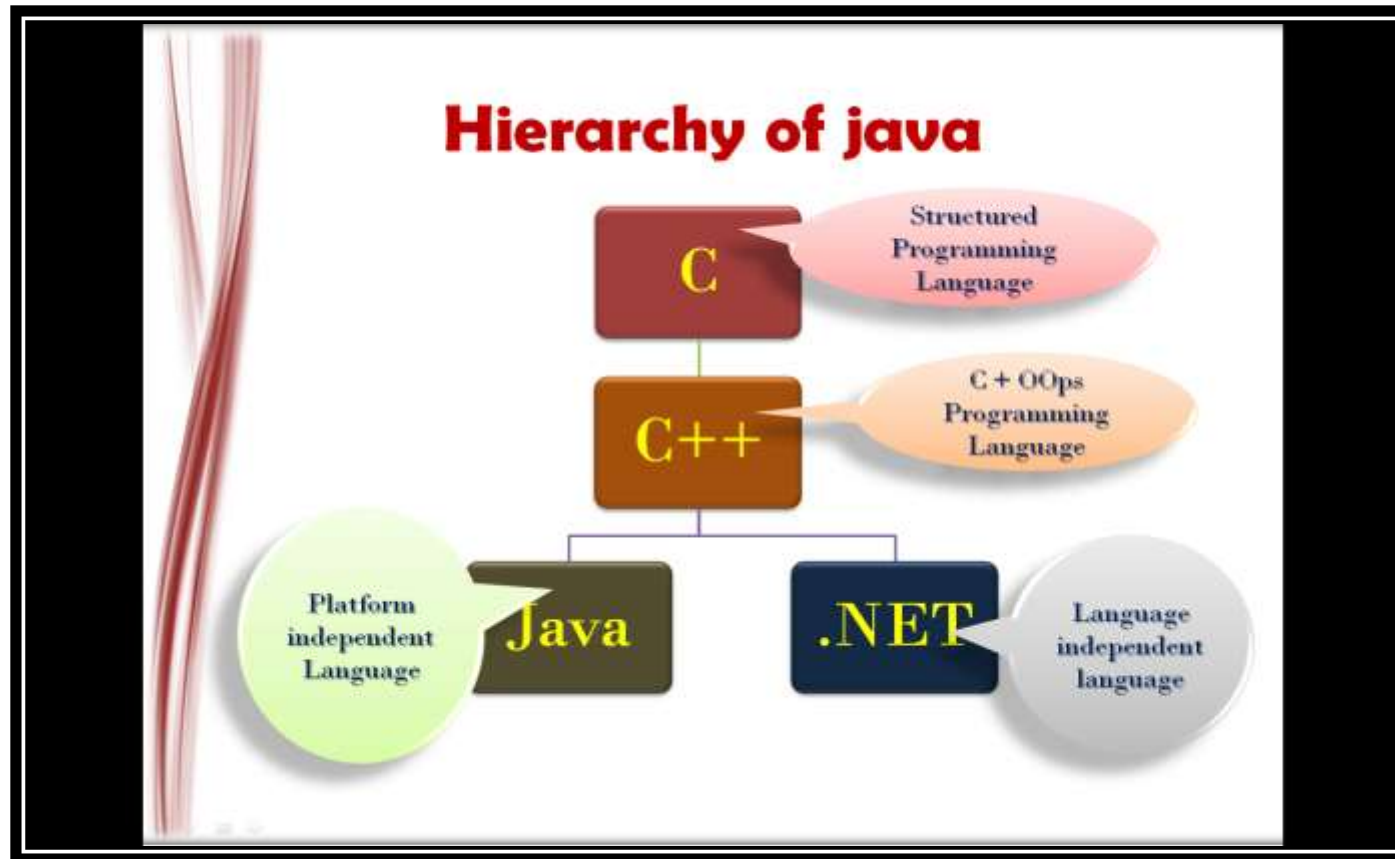
Objective

- ▶ Java Editions
- ▶ Java Based Platforms
- ▶ Type of Applications
- ▶ Frameworks
- ▶ Java Based Servers
- ▶ Java 5.0,6.0,7.0,8.0 Features
- ▶ JAVA APIs
- ▶ J2EE APIs



JAVA Hierarchy

- ▶ Java is both a programming language and a platform.

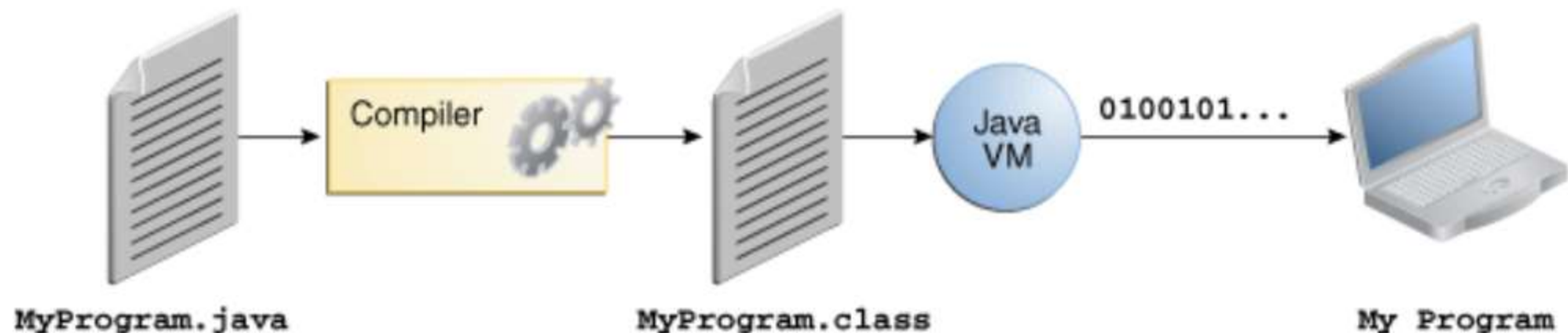


Characteristics of JAVA

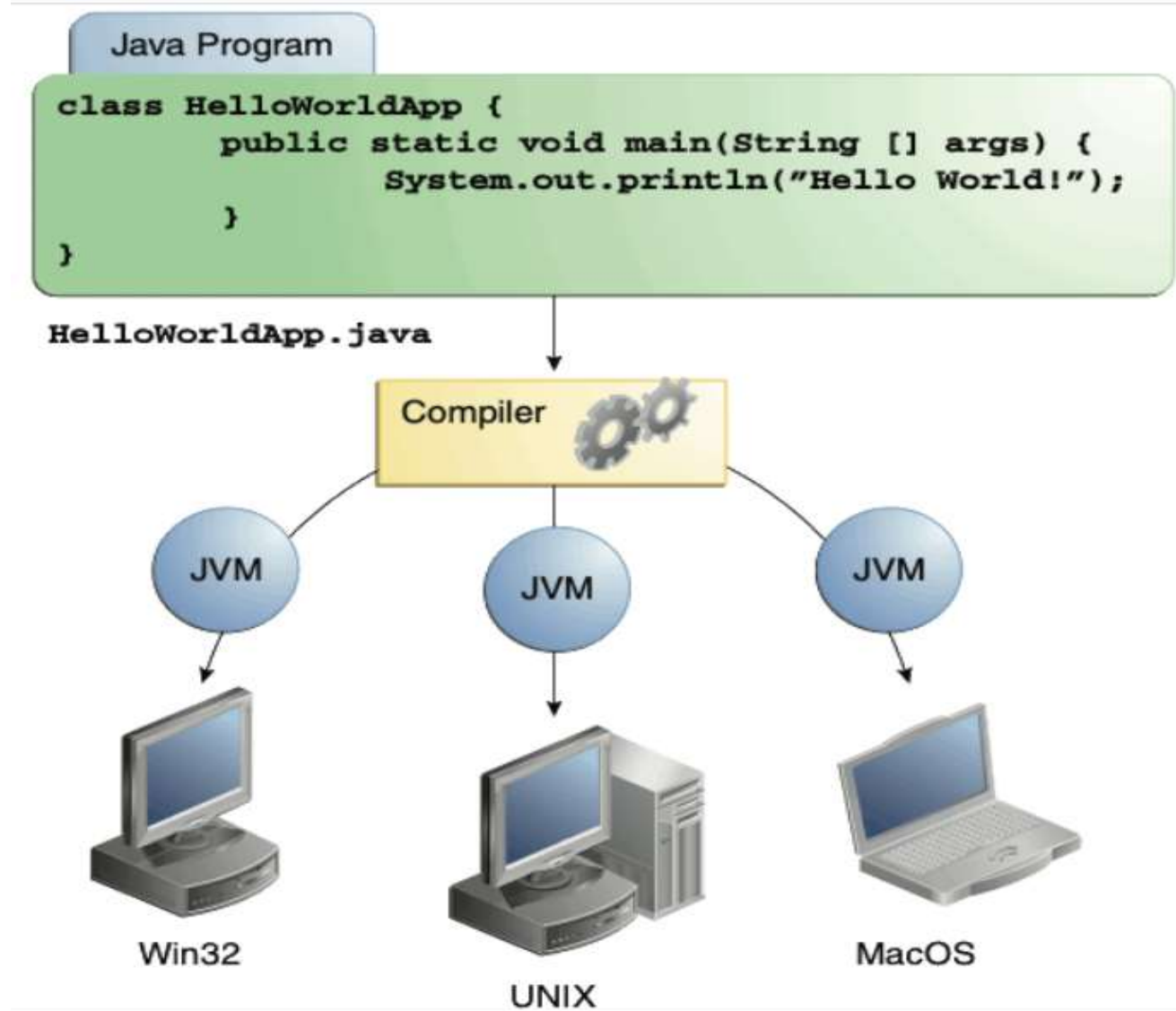
- ▶ Simple and Object Oriented
- ▶ Robust and Secure
- ▶ Standalone & Distributed
- ▶ Architecture Neutral and Portable
- ▶ High Performance
- ▶ Interpreted, Threaded, and Dynamic

An overview of the software development process.

- ▶ All source code is first written in plain text files ending with the .java extension.
- ▶ Those source files are then compiled into .class files by the javac compiler.
- ▶ A .class file does not contain code that is native to your processor; it instead contains bytecodes — the machine language of the Java Virtual Machine¹ (Java VM).
- ▶ The java launcher tool then runs your application with an instance of the Java Virtual Machine.

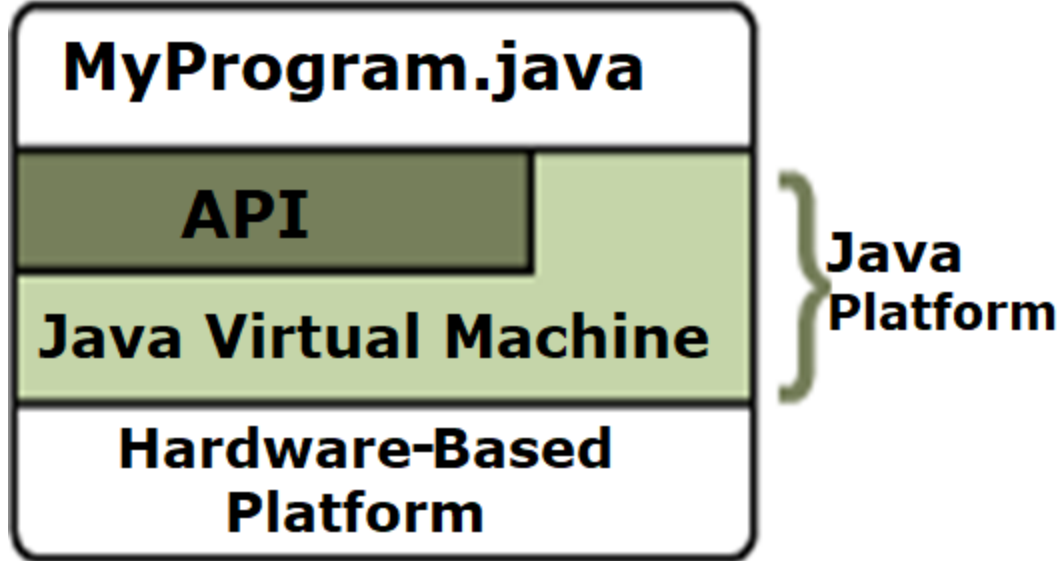


An overview of the software development process.

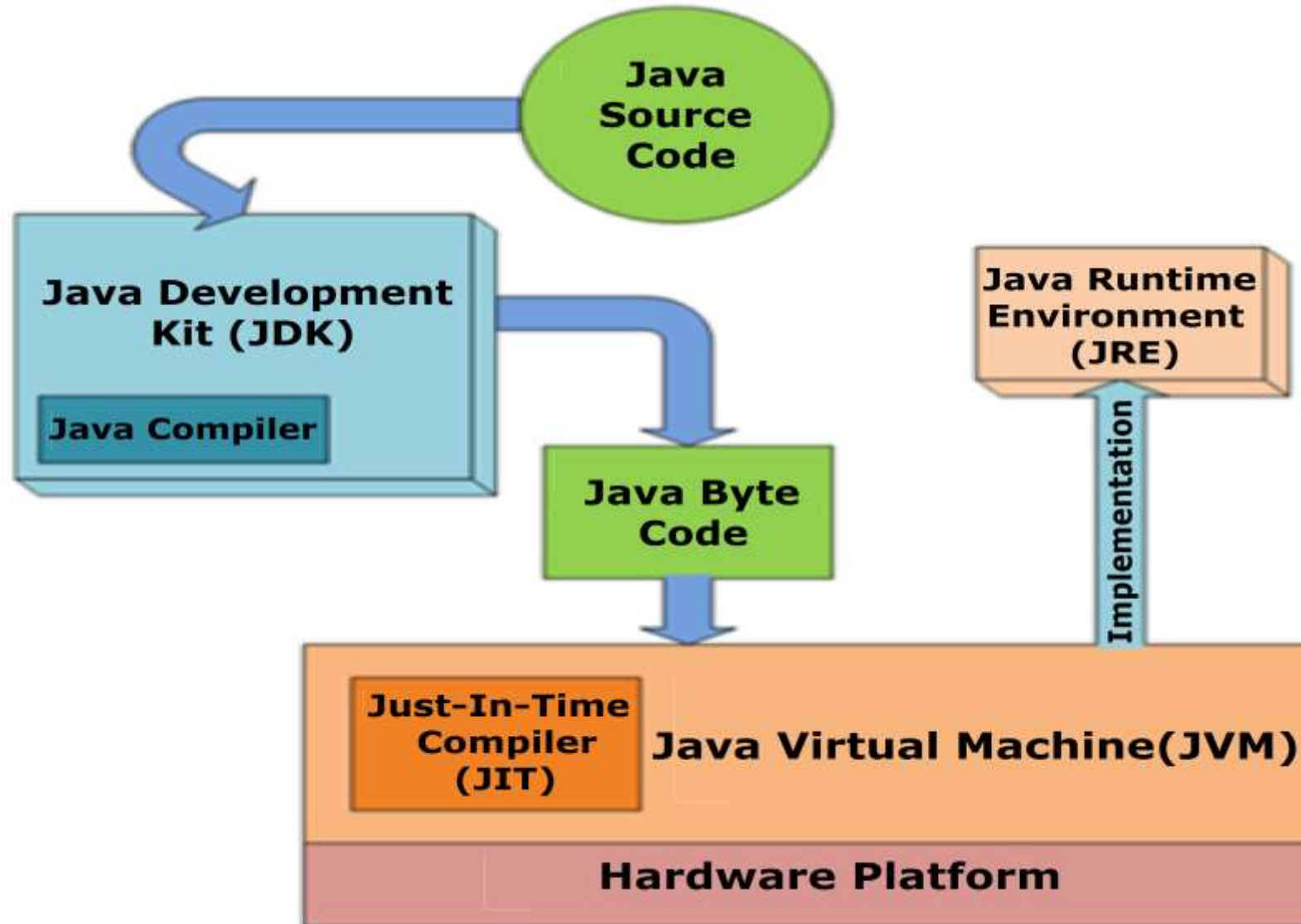


JAVA Platform

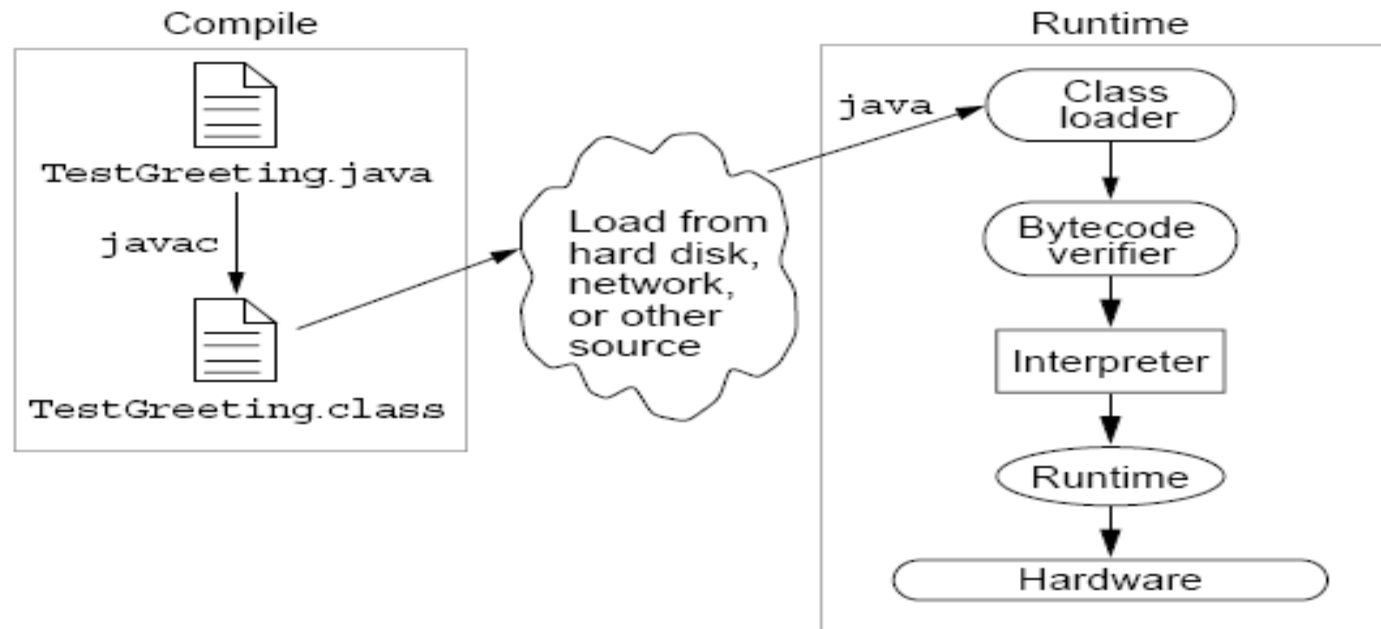
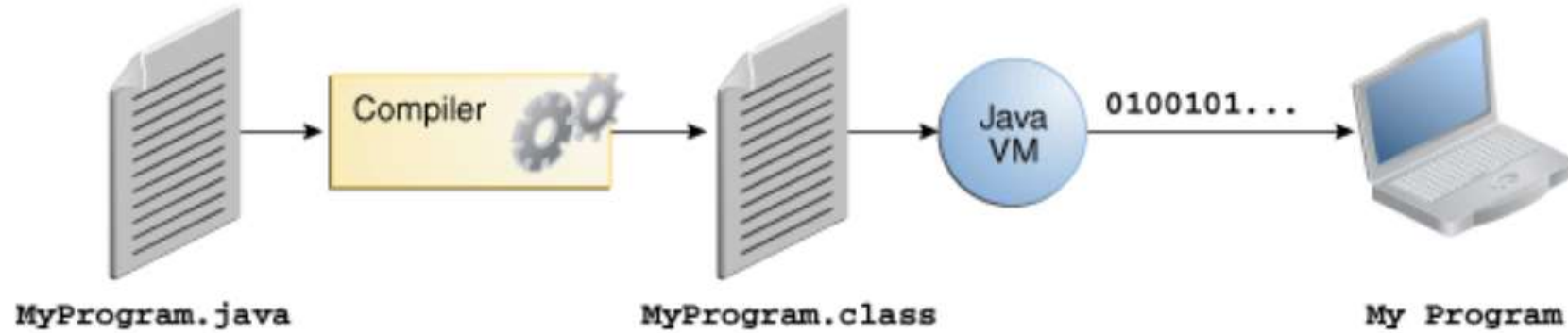
- ▶ Java platform has two components
 - The *Java Virtual Machine*
 - The *Java Application Programming Interface (API)*
- ▶ *JAVA is Platform Independent but JVM is platform dependent.*



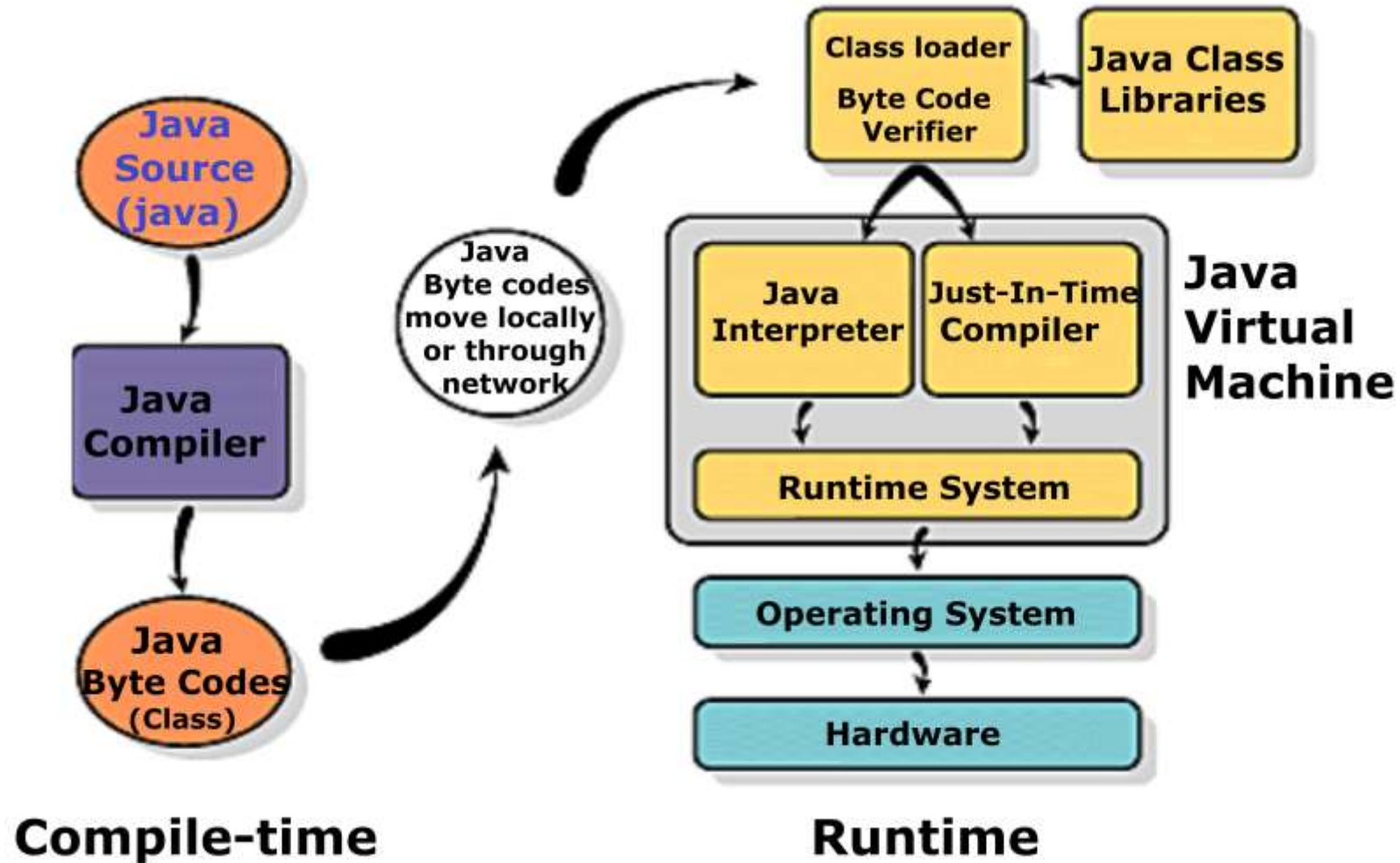
JRE vs. JDK

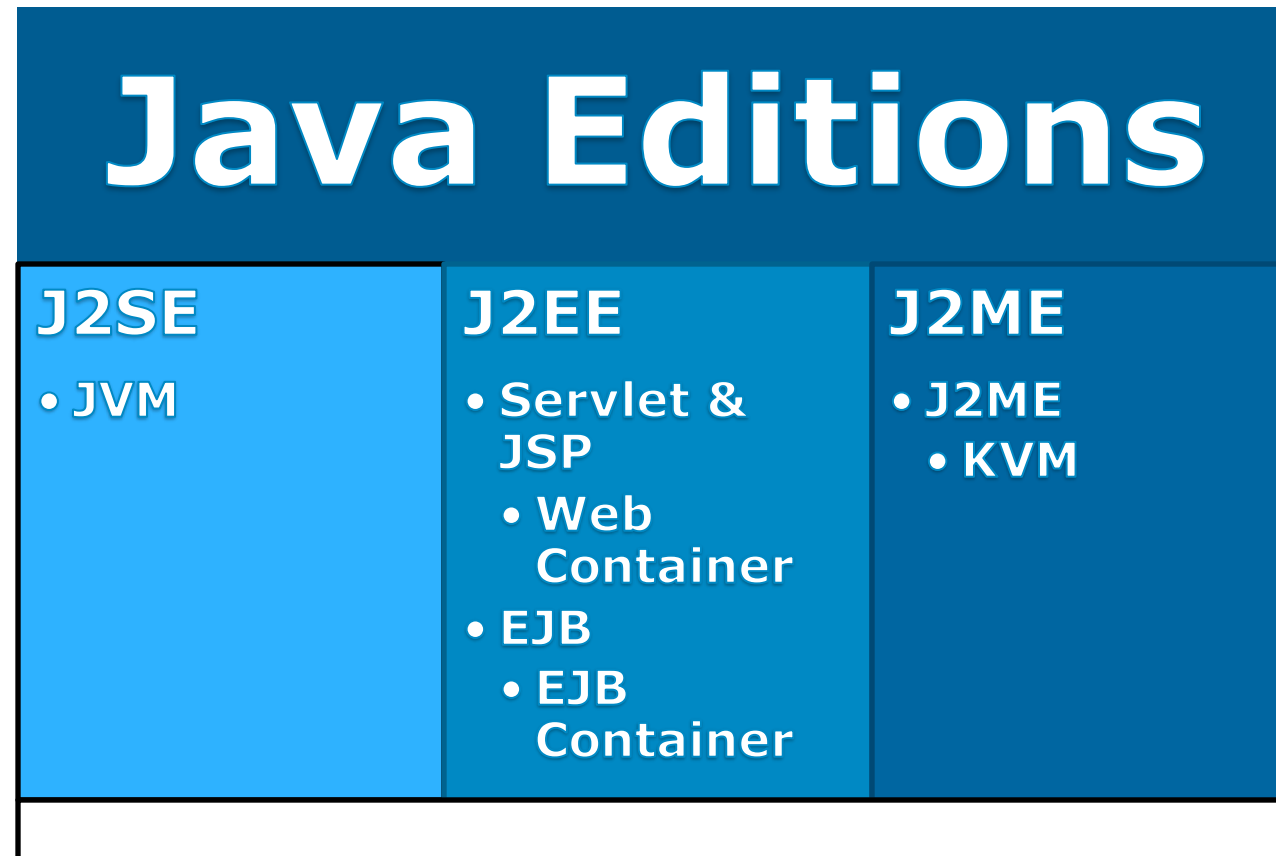


Java Language Runtime Environment

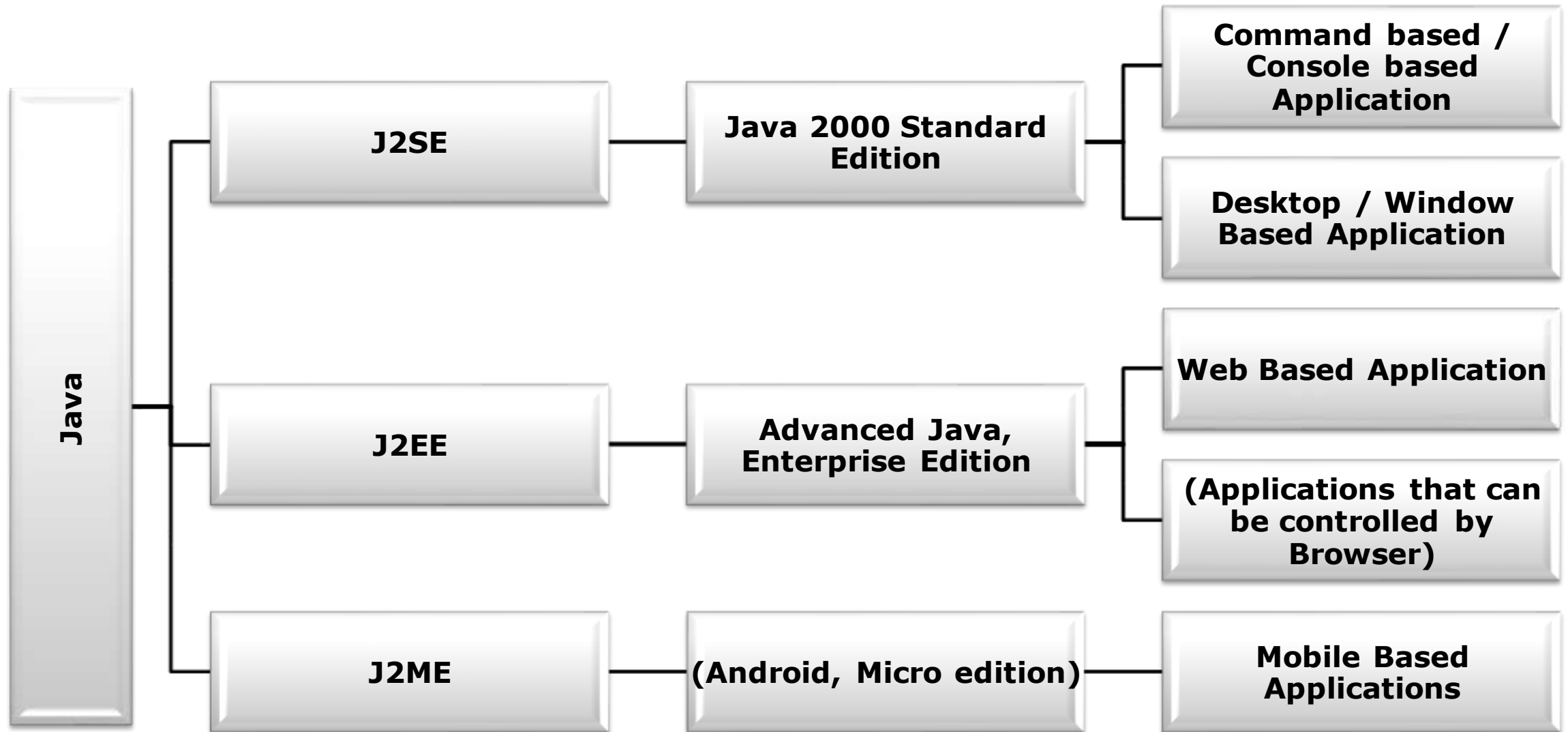


Java Compile-time and Runtime

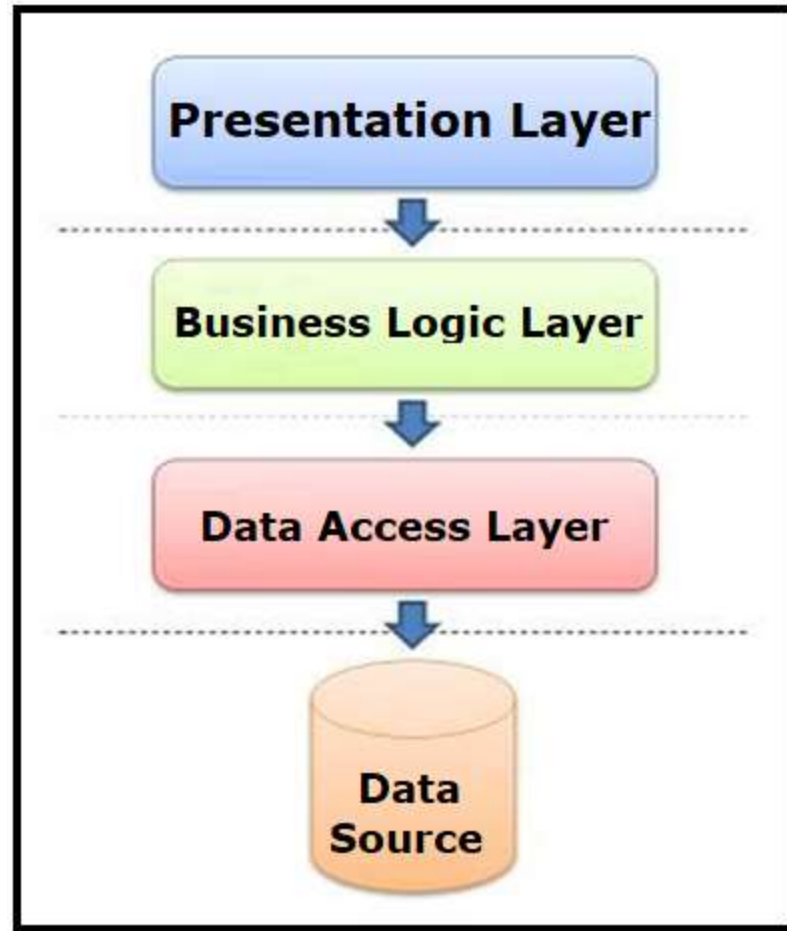




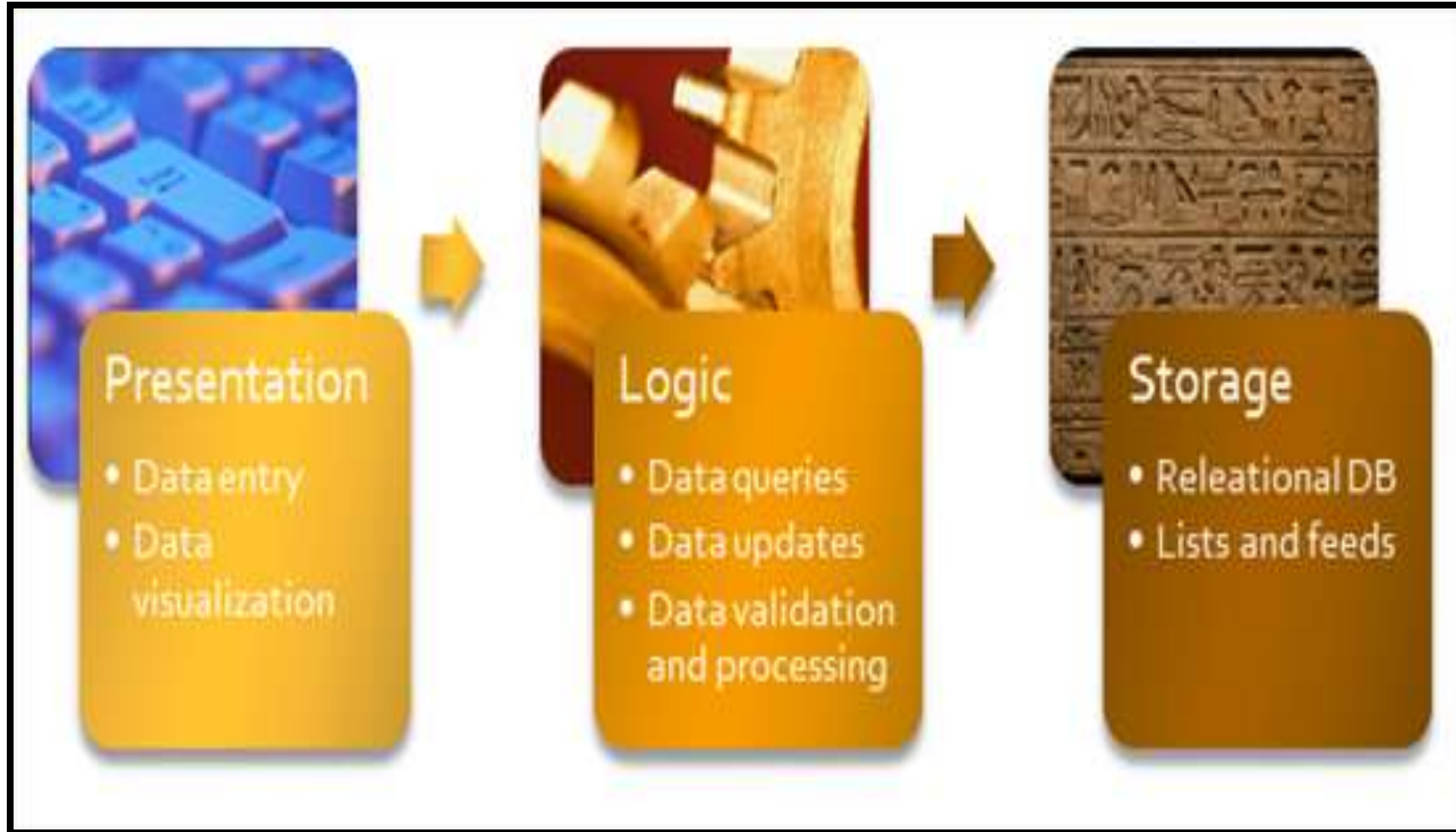
JAVA Editions and Platforms



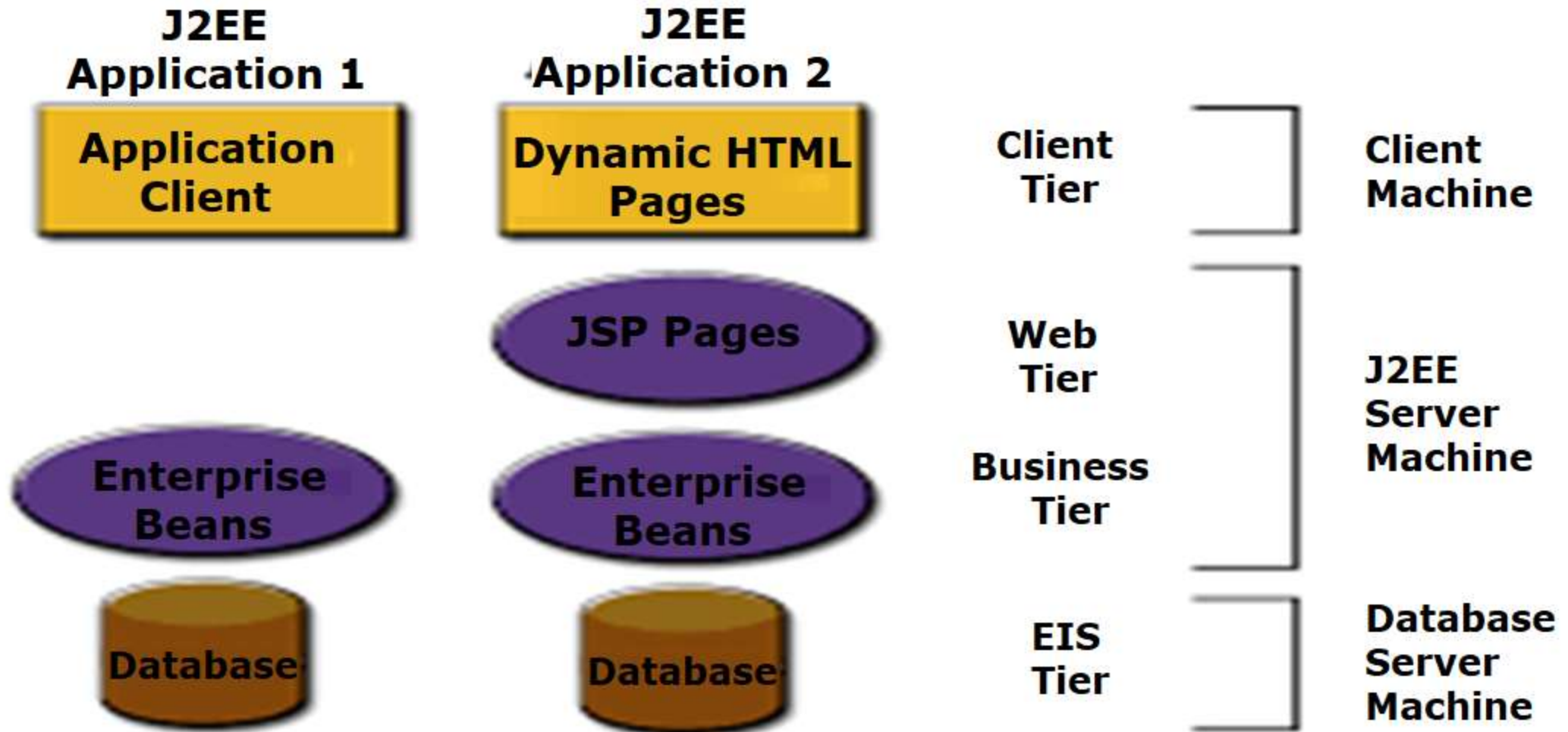
Application Layers



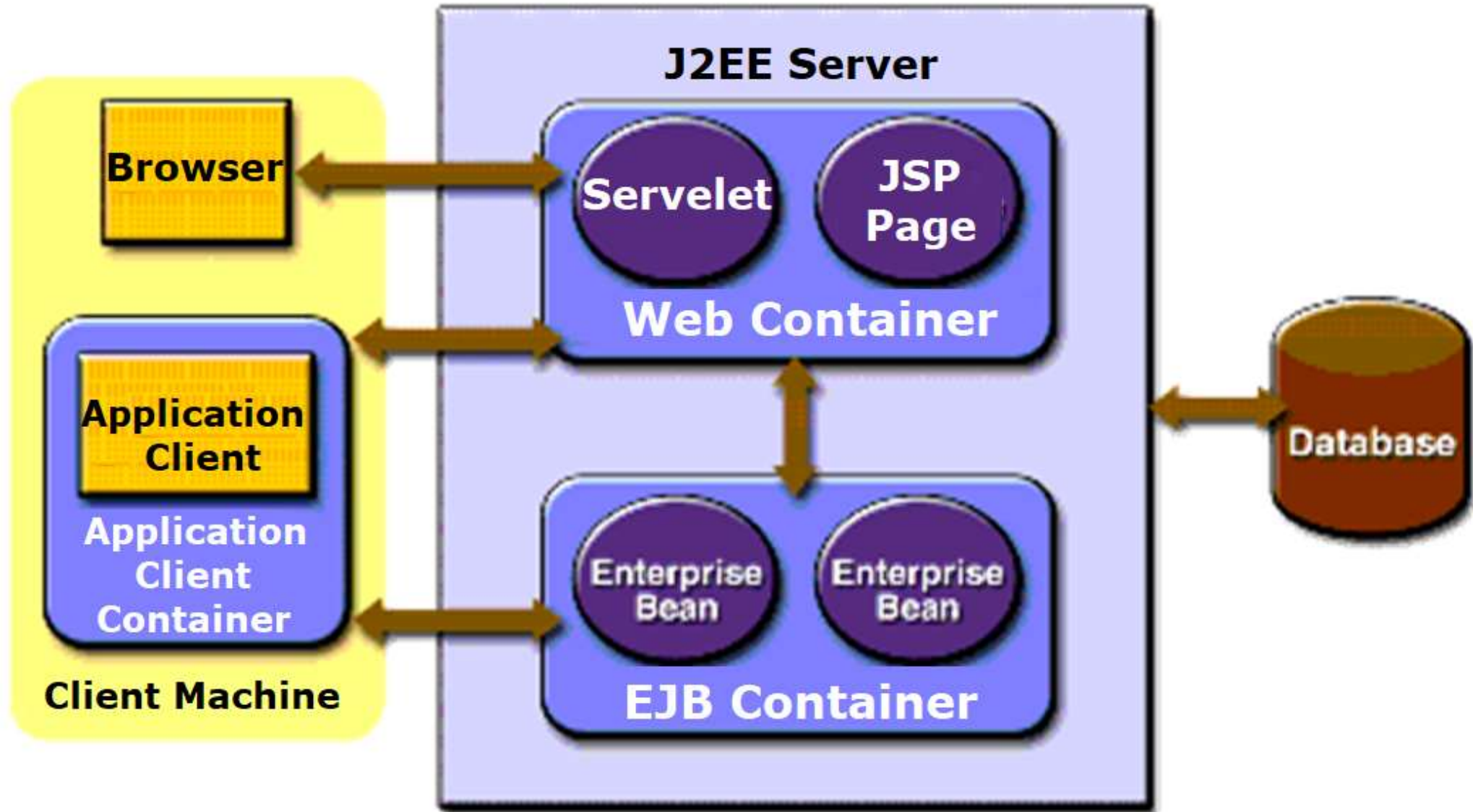
Coding in application Layers



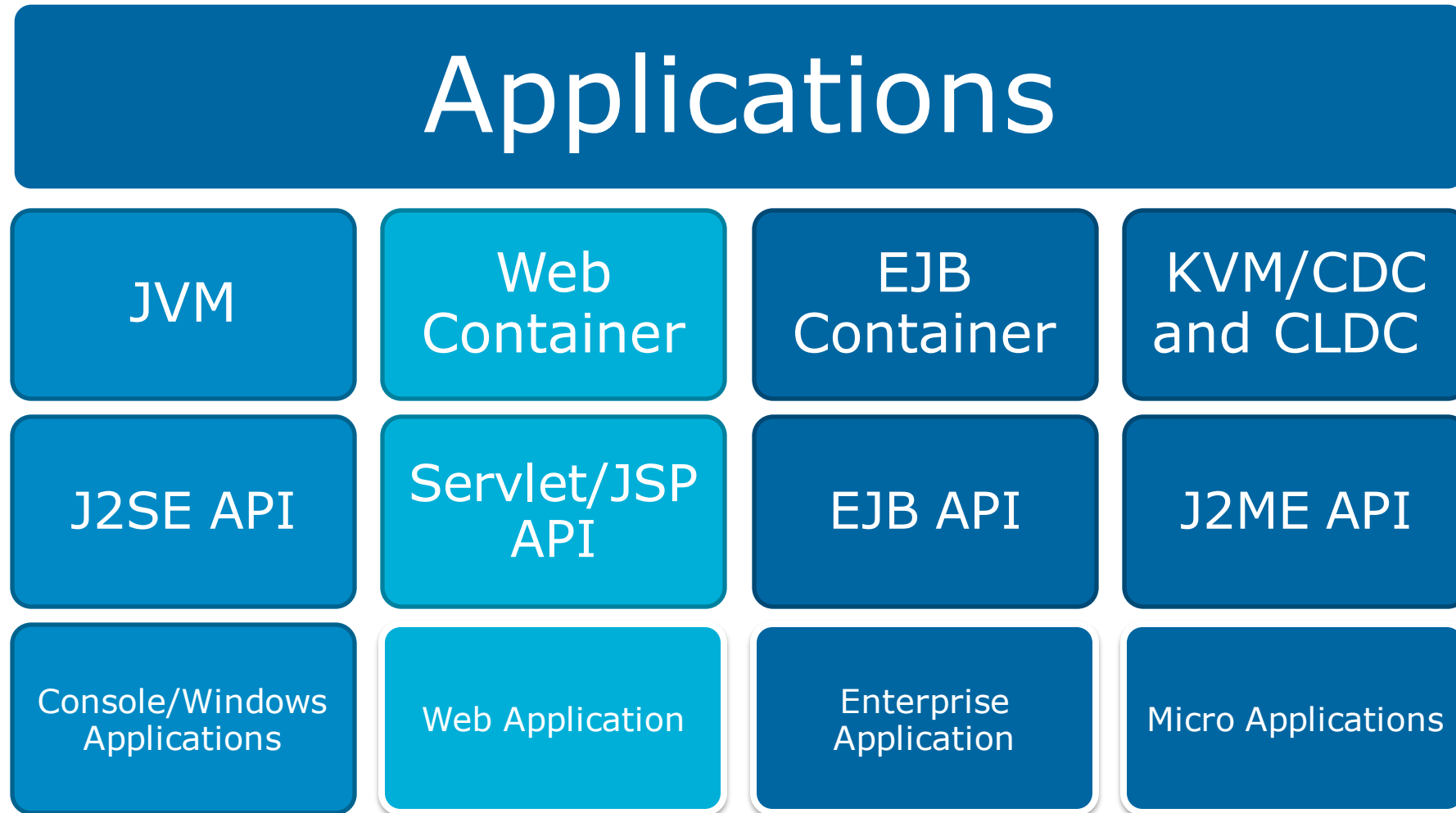
J2EE Applications



J2EE Containers



JAVA Applications vs. Platforms



JAVA/J2EE Frameworks

Struts for Web
Application Development

Hibernate(ORM) for DB
Connectivity
Development

JSF for Web Application
Development

Spring for Everything

JAVAFX for Web
Application Development

JAVA Servers

Glassfish

Apache Tomcat

JBoss

Apache TomEE

Wildfly

Jetty

JBonAs

Apache Geronimo

JAVA 5.0 Features

- **Generics**
- **Enhanced For Loop**
- **Auto boxing / Unboxing**
- **Enums**
- **Varargs**
- **Static Import**
- **Metadata / Annotations**
- **Garbage Collectors**
- **Java Concurrent package**

JAVA 6.0 Features

- **Scripting Language Support**
- **Performance improvements**
- **JAX-WS**
- **JDBC 4.0**
- **Java Compiler API**
- **JAXB 2.0 and StAX parser**
- **Pluggable annotations**
- **New GC algorithms**

JAVA 7.0 Features

- **Strings in switch**
- **Automatic resource management in try-statement**
- **The diamond operator**
- **Binary integer literals**
- **Underscores in numeric literals**
- **Improved exception handling**

JAVA 8.0 Features

- **Lambda expression support in APIs**
- **Functional interface & default methods** Optionals
- **Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications**
- **Annotation on Java Types**
- **Unsigned Integer Arithmetic**
- **Repeating annotations**
- **New Date and Time API**
- **Statically-linked JNI libraries**

JAVA & J2EE APIs

- **Enterprise JavaBeans Technology (EJB)**
- **Java Message Service (JMS)**
- **Java Servlet Technology (Servlet/Web Component)**
- **JavaServer Faces Technology (JSF)**
- **JavaServer Pages Technology (JSP)**
- **JavaServer Pages Standard Tag Library (JSTL)**
- **Java Persistence API (JPA)**
- **Java Transaction API (JTA)**

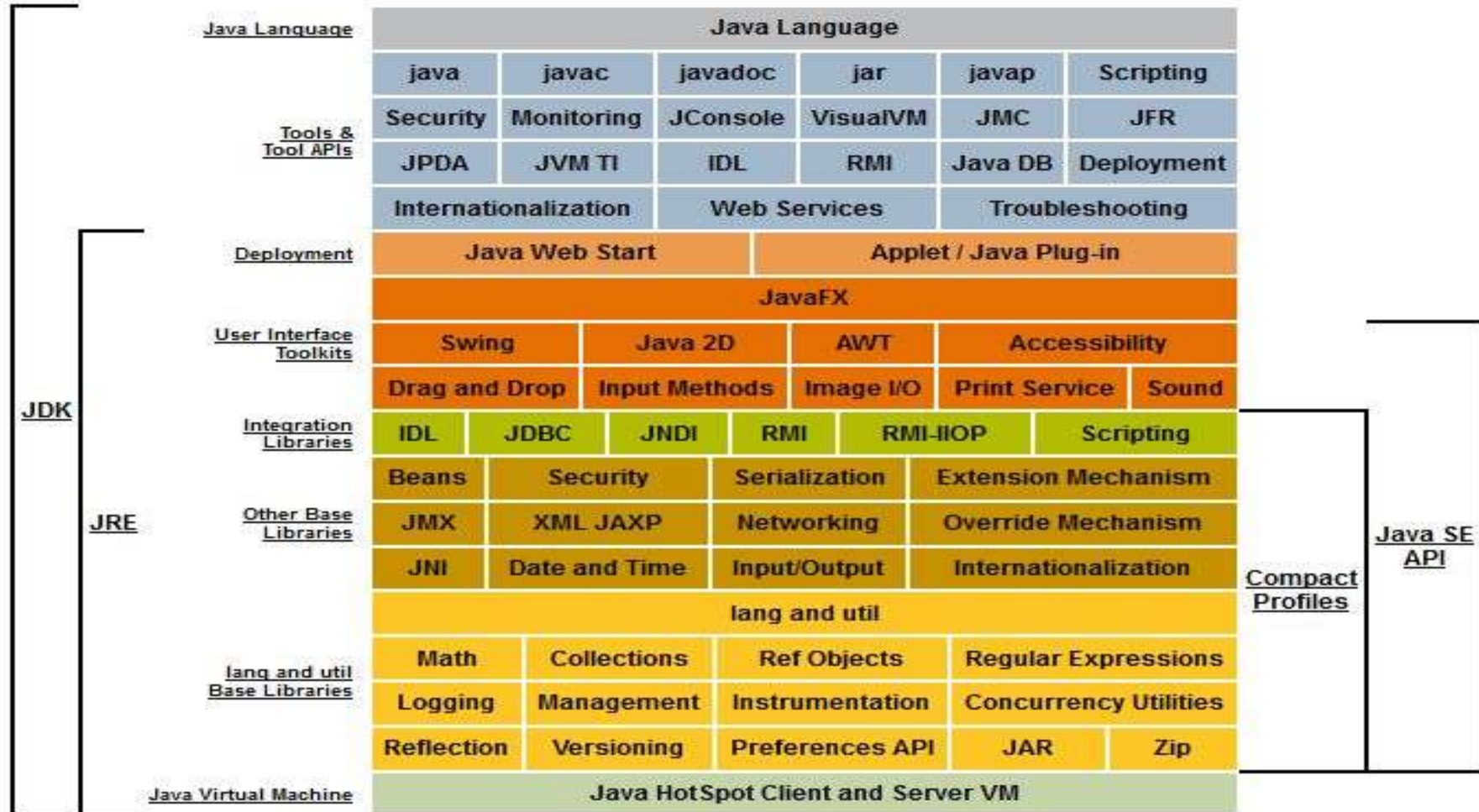
JAVA & J2EE APIs

- **Java API for RESTful Web Services**
- **Contexts and Dependency Injection for the Java EE Platform**
- **Dependency Injection for Java**
- **Bean Validation**
- **Java EE Connector Architecture**
- **JavaMail API**
- **Java Authorization Contract for Containers**
- **Java Authentication Service Provider Interface for Containers**

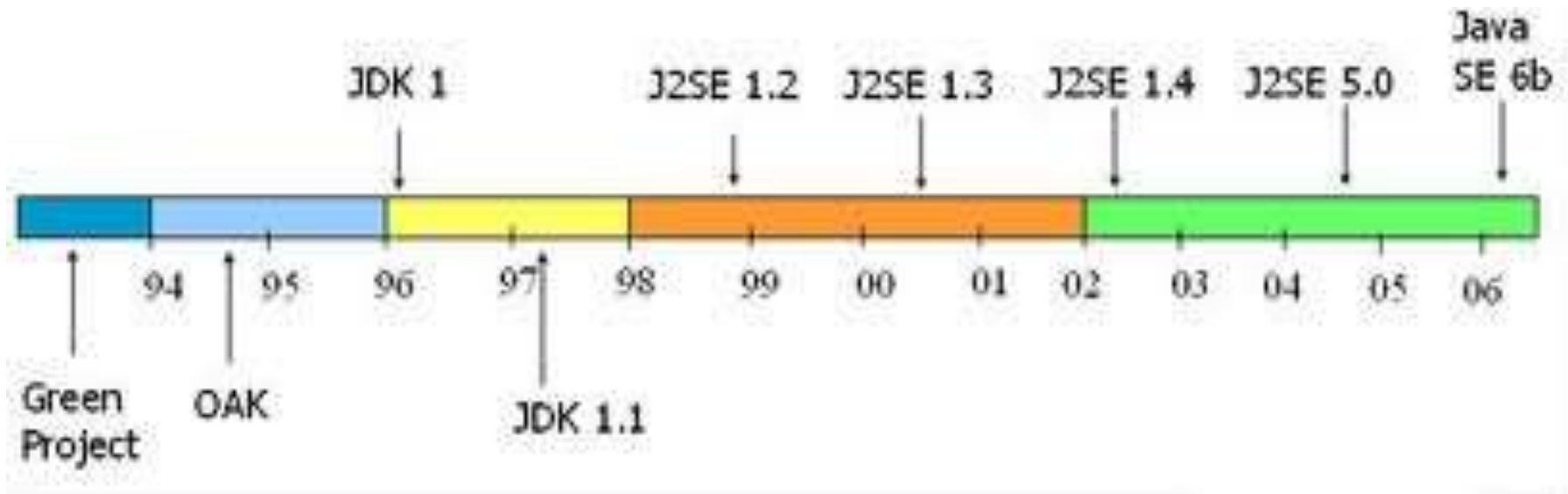
JAVA & J2EE APIs

- **Java Database Connectivity API**
- **Java Naming and Directory Interface API**
- **JavaBeans Activation Framework**
- **Java API for XML Processing**
- **Java Architecture for XML Binding**
- **SOAP with Attachments API for Java**
- **Java API for XML Web Services**
- **Java Authentication and Authorization Service**

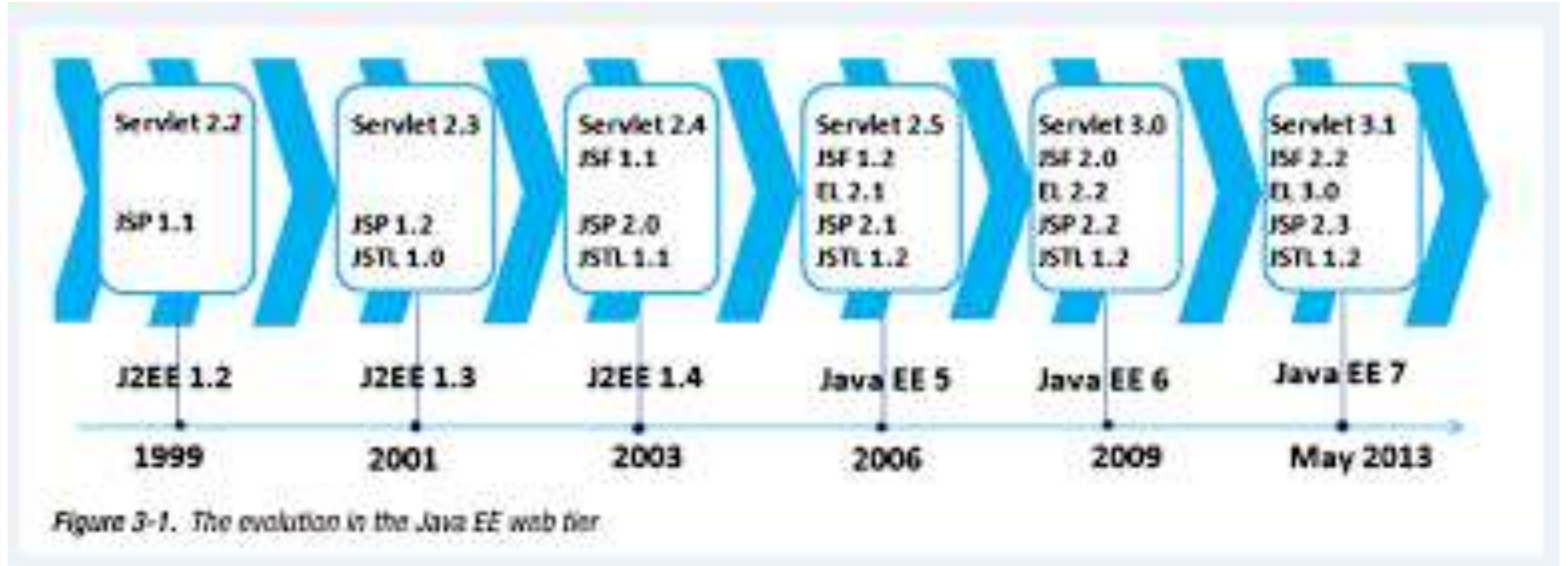
JAVA & J2EE APIs



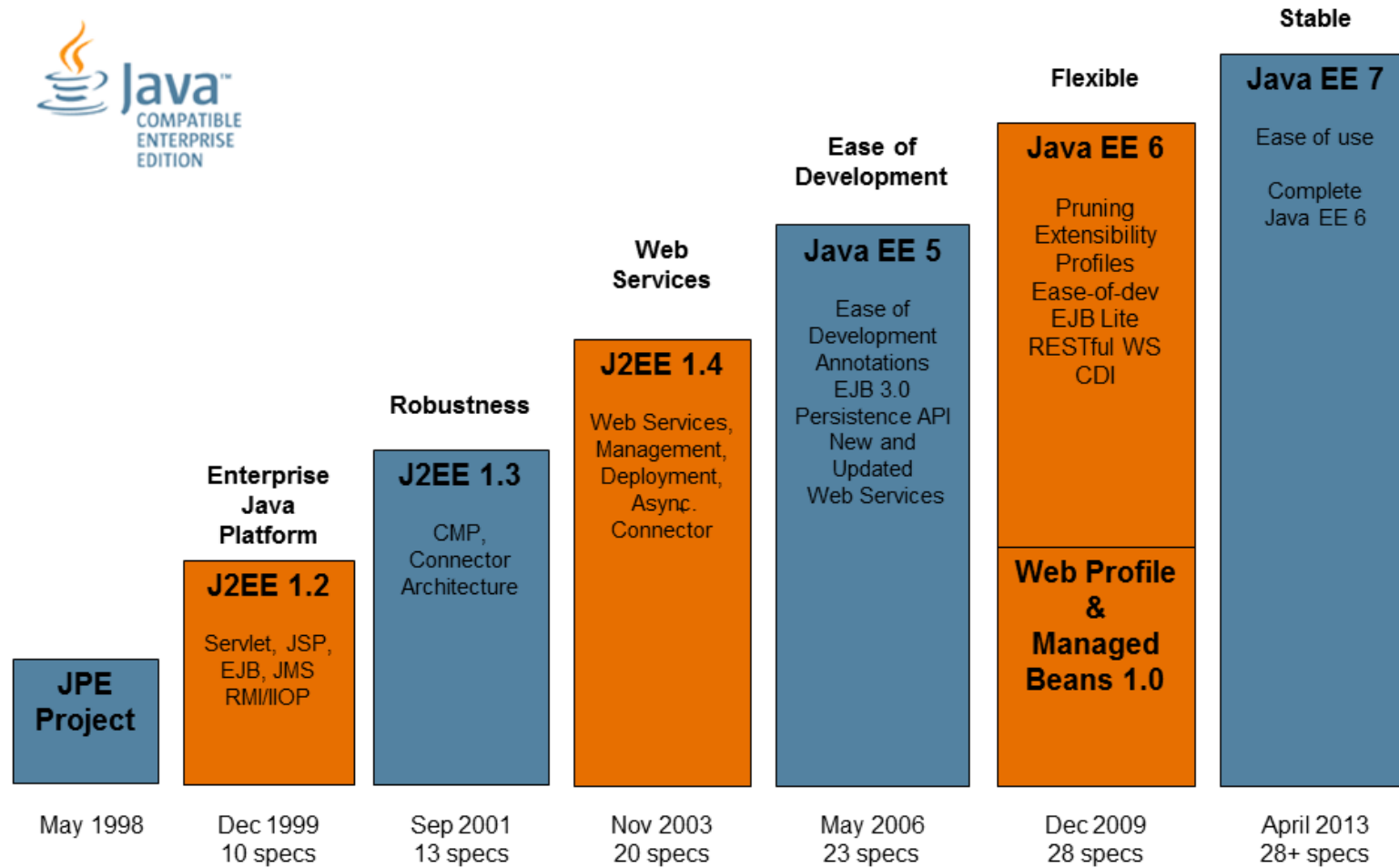
JAVA Evolution



J2EE Evolution



J2EE Evolution



Java Language Fundamentals

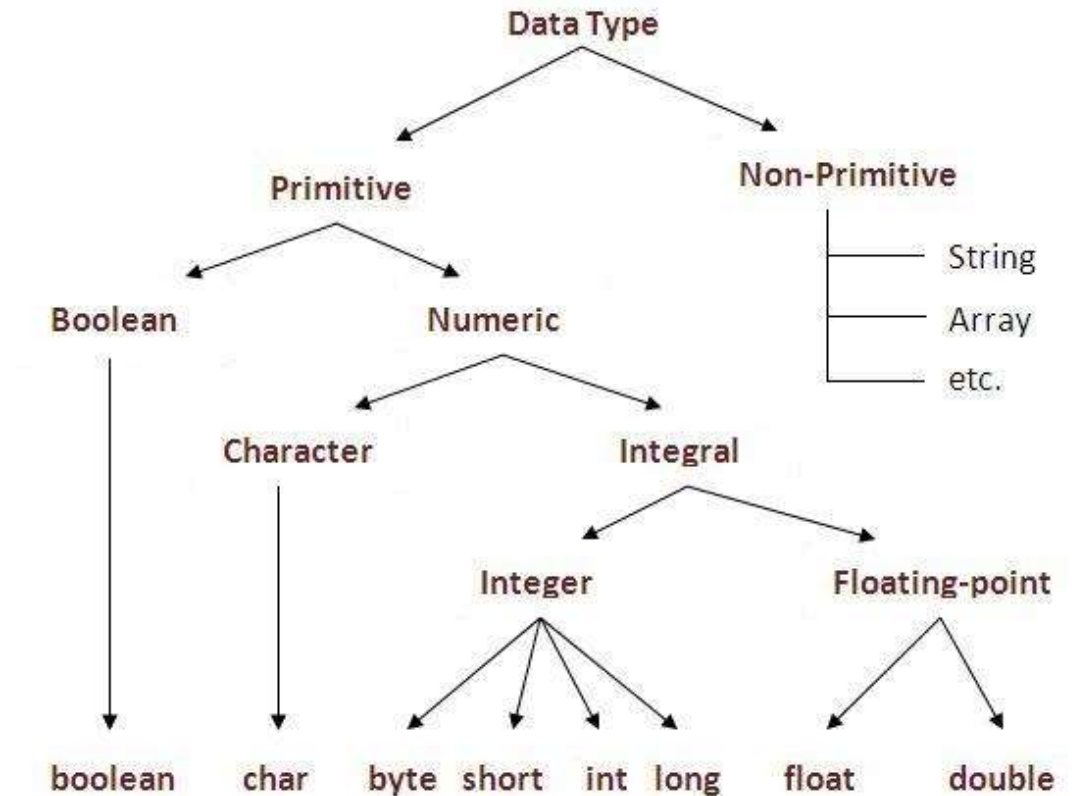
Objectives

- ▶ Data Types
- ▶ Variable
- ▶ Key word
- ▶ Constants and Literals
- ▶ Operators
- ▶ Controls statement
- ▶ If statements
- ▶ Switch statements
- ▶ Loop Statements
- ▶ Array
- ▶ Packages
- ▶ Main Method



Data Types

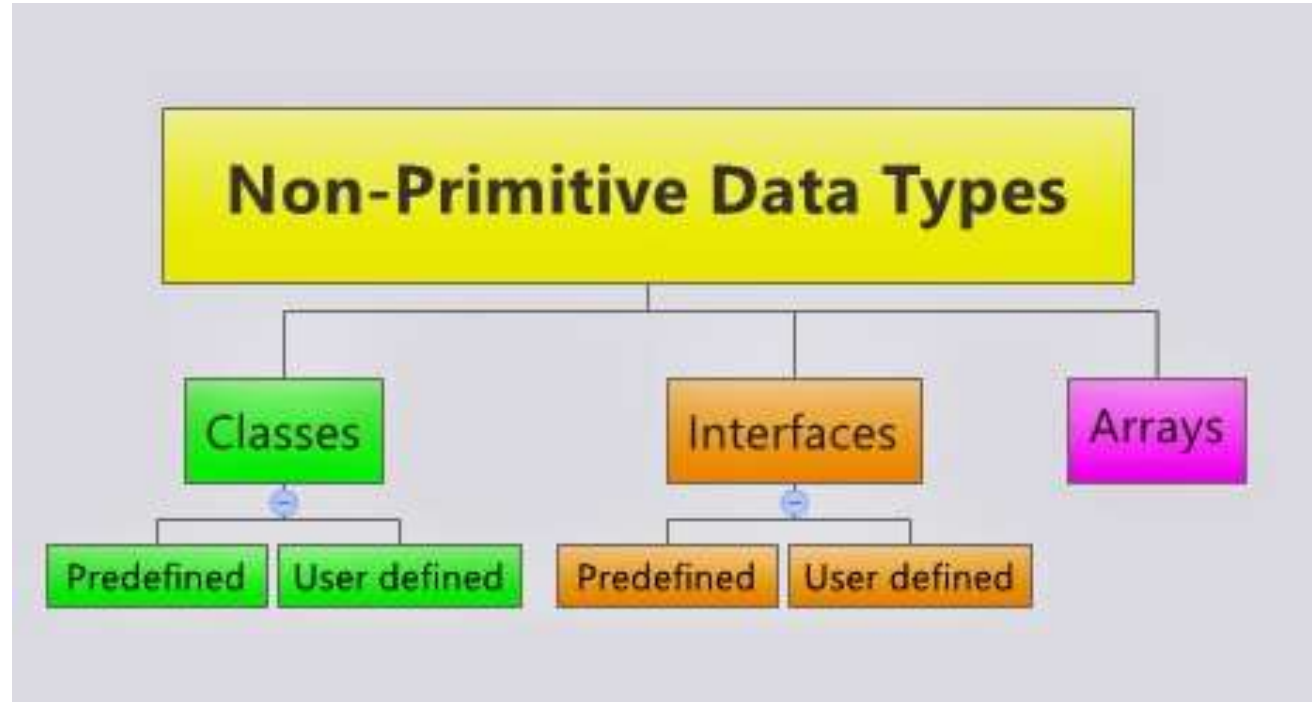
- ▶ Data types are two categories in JAVA
 - Primitive types
 - There are 8 primitive types
 - Non primitives types (Reference Types)
 - All the remaining are Reference types
 - Examples
 - » Enum
 - » Array
 - » Class



Primitive Types

Type	Size	Default
boolean	1 bit	false
byte	8 bits	0
int	32 bits	0
long	64 bits	0L
float	32 bits	0.0f
double	64 bits	0.0d
char	16 bits	'\u0000'
short	16 bits	0

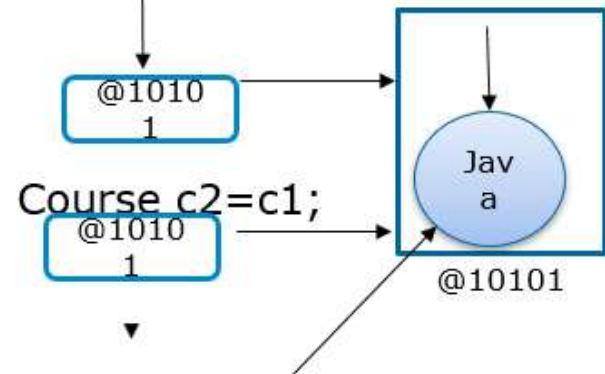
Non Primitive Types



Primitive Vs. reference types



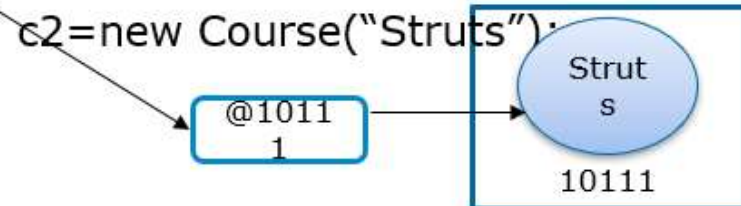
Course c1 = new Course("JAVA");



Course c2 = c1;



c2.setName("J2EE");



```
public class Course {  
    private String name;  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Java Identifiers

- ▶ All java components require names. Names used for classes, variables and methods are called identifiers.
 - All identifiers should begin with a letter (A to Z or a to z), currency character (\$) or an underscore (_).
 - After the first character identifiers can have any combination of characters.
 - A key word cannot be used as an identifier.
 - Most importantly identifiers are case sensitive.
 - Examples of legal identifiers: age, \$salary, _value, __1_value
 - Examples of illegal identifiers : 123abc, -salary

Java Standard Names

- ▶ Package name – should be in lowercase
- ▶ Class name – First letter of the word should be in Uppercase
- ▶ Variable and methods – should be in camel letter
- ▶ Final variable – should be in upper case
- ▶ Sample code:

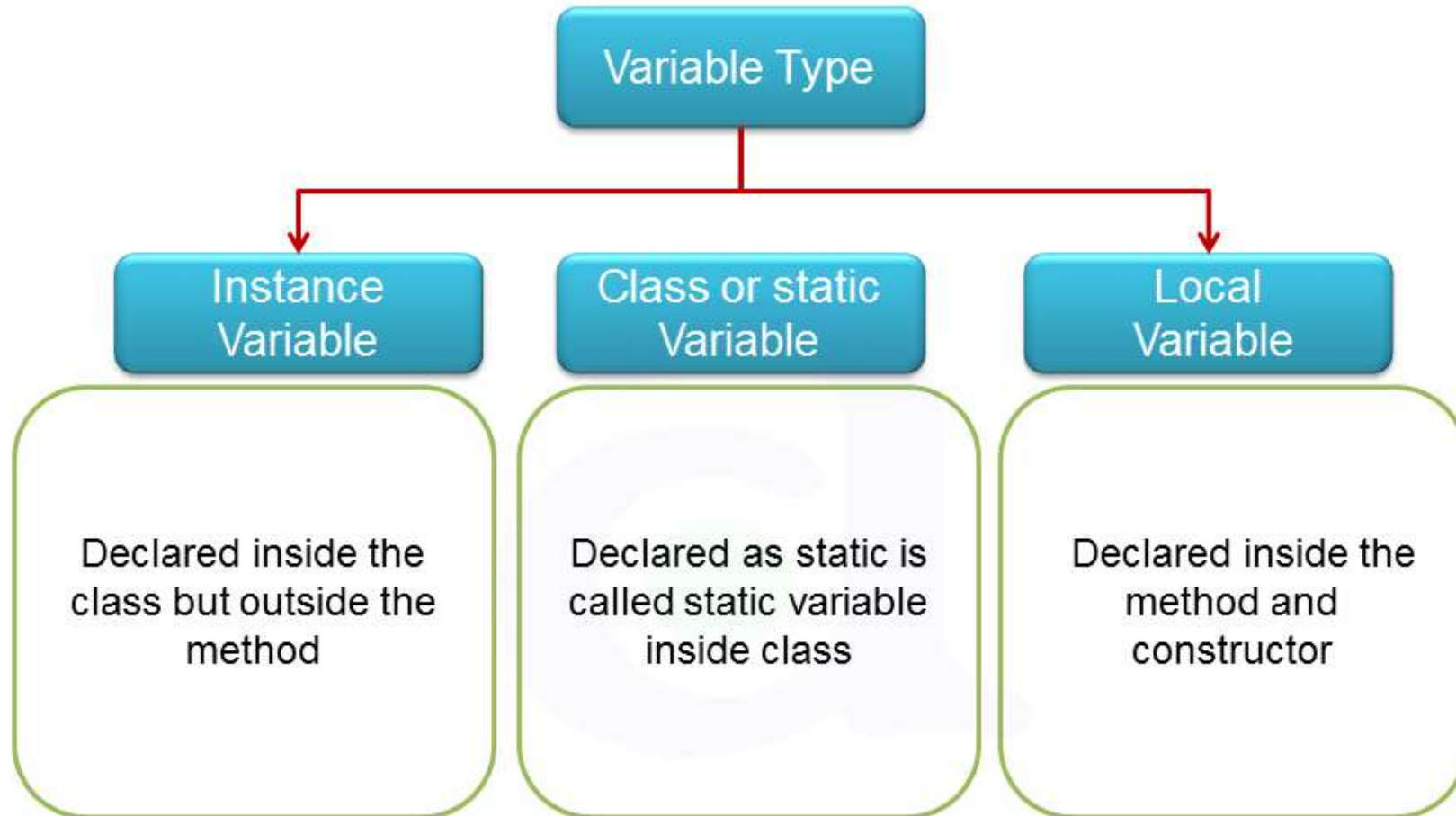
```
package com.training.first2; //- should be in lower case

public class Course { //First letter of each word in the class should be in Upper case

    private String courseName; //should be in camel letter
    private static final String COPYRIGHTS="Syntel"; //should be in upper case

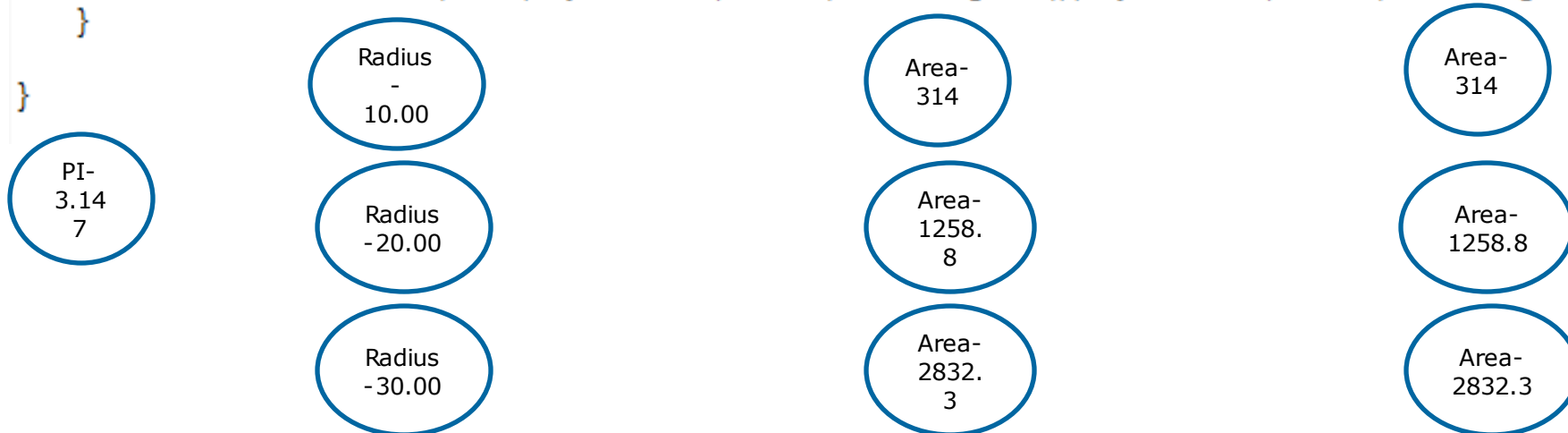
    public String getCourseName() { //should be in camel letter
        return courseName;
    }
    public void setCourseName(String courseName) {
        this.courseName = courseName;
    }
}
```

Scope of Variables



Scope of Variable

```
package com.training.first;
public class Circle {
    private double radius;//created whenever new object is created for this class
    public static final double PI=3.147;//created only once after class is successfully loaded.
    public Circle(double radius) {
        this.radius=radius;
    }
    private double findingArea(){
        double area;//created whenever this method is being called
        area=PI*radius*radius;
        return area;
    }
    public static void main(String[] args) {
        Circle c1=new Circle(10.00);System.out.println(c1.findingArea());System.out.println(c1.findingArea());
        Circle c2=new Circle(20.00);System.out.println(c2.findingArea());System.out.println(c2.findingArea());
        Circle c3=new Circle(30.00);System.out.println(c3.findingArea());System.out.println(c3.findingArea());
    }
}
```



Java Literals

```
public static void main(String[] args) {  
    byte data1=127;  
    short data2=32767;  
    //by default numbers are integer type  
    int data3=2147483647;  
    //Long values should be ending with l  
    long data4=2147483648l;  
    //maximum 4 decimal digits  
    //Float values should be ending with f  
    float data5=123456789012345678901234567890123456789.1234f;  
    //maximum 13 decimal digits  
    //by default decimal values are double type  
    double data6=123456789012345678901234567890123456789.1234567890123;  
    //Binary values should begin with 0B  
    //Binary literal were added in JAVA 1.8  
    int data7=0B144;  
    //Octal values should begin with 0  
    int data8=0144;  
    //Hexadecimal values should begin with 0X  
    int data9=0X144;  
    //single Character value should be given within single quotes  
    char data10='A';/*or*/char data10=65;  
    //boolean does not support zero or one  
    boolean data12=true;data12=false;  
    //String values should be enclosed within double quotes.  
    String data13="JAVA";  
}
```

Underscores in Numeric Literals from Java 7.0

- ▶ Declaring numeric literals with underscores would have caused a compile error in any previous release of the language. But in Java 7 numeric literals with underscore characters are allowed.
 - You can place any number of underscore(_) between digits of numeric literals.
 - This added feature improves the readability of the code.

- ▶ You can't place underscore in the following places:
 - You can't place it at the beginning or at the end of a number.
 - Adjacent to a decimal point in a floating point literal.
 - Before to an F or L suffix.
 - In positions where a string of digits is expected.

- Sample Coding:
 - `long creditCardNumber = 1234_5678_9012_3456L;`
 - `long socialSecurityNumber = 999_99_9999L;`
 - `float pi = 3.14_15F;`
 - `long hexBytes = 0xFF_EC_DE_5E;`
 - `long hexWords = 0xCAFE_BABE;`
 - `long maxLong = 0x7fff_ffff_ffff_ffffL;`

Escape sequences

Escape sequence	Description
\' (single quote)	Output the single quote (') character.
\" (double quote)	Output the double quote (") character.
\\ (backslash)	Output the backslash (\) character.
\b (backspace)	Move the cursor back one position on the current line.
\f (new page or form feed)	Move the cursor to the start of the next logical page.
\n (newline)	Move the cursor to the beginning of the next line.
\r (carriage return)	Move the cursor to the beginning of the current line.
\t (horizontal tab)	Move the cursor to the next horizontal tab position.

Java Operators

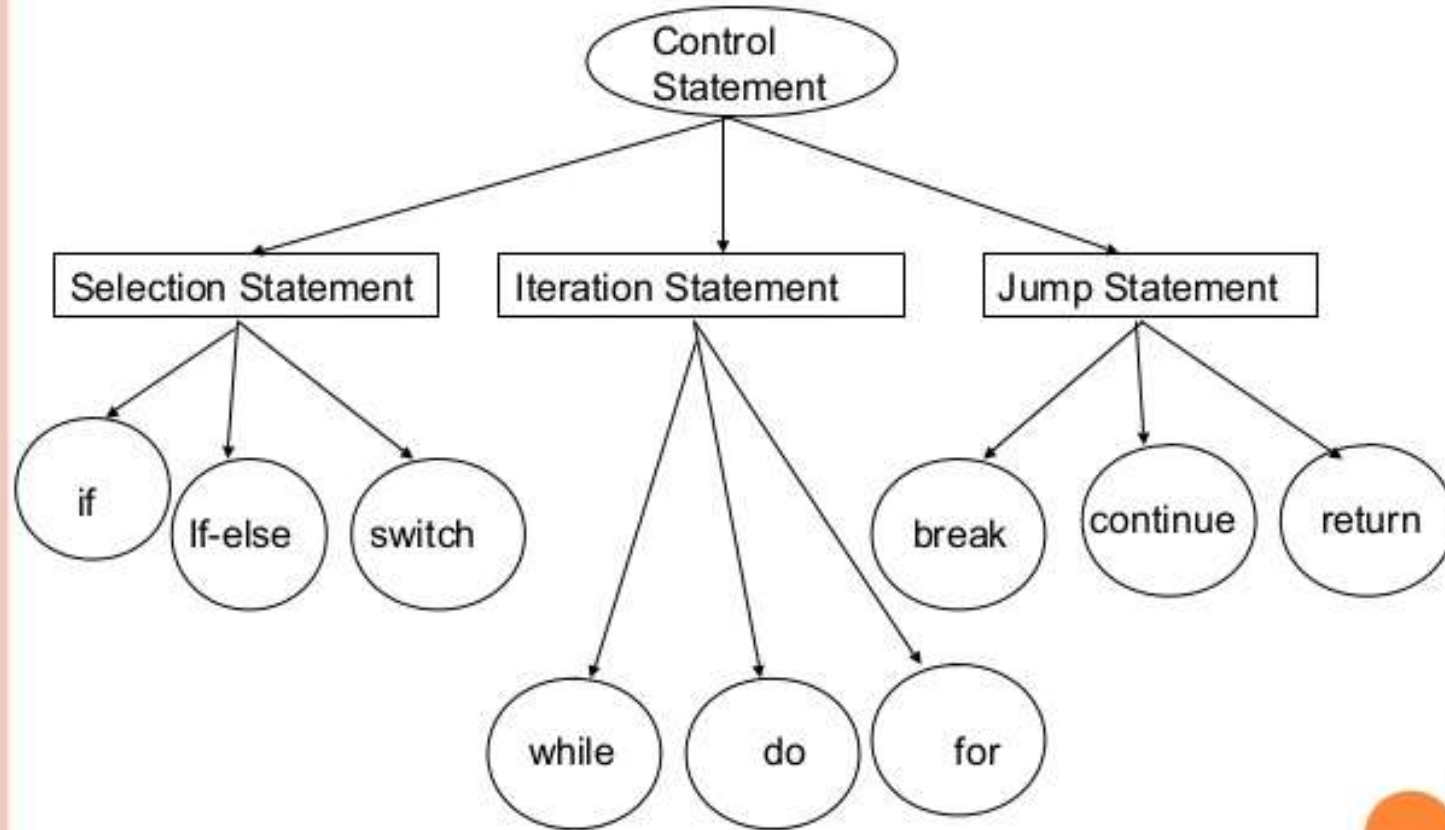
- ▶ Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:
 - Arithmetic Operators
 - Relational Operators
 - Bitwise Operators
 - Logical Operators
 - Assignment Operators
 - Misc. Operators

Precedence of Java Operators

Operators	Associativity	Type
++ --	right to left	unary postfix
++ -- + - ! (type)	right to left	unary prefix
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&	left to right	boolean logical AND
^	left to right	boolean logical exclusive OR
	left to right	boolean logical inclusive OR
&&	left to right	conditional AND
	left to right	conditional OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

JAVA Control Statements

JAVA CONTROL STATEMENTS



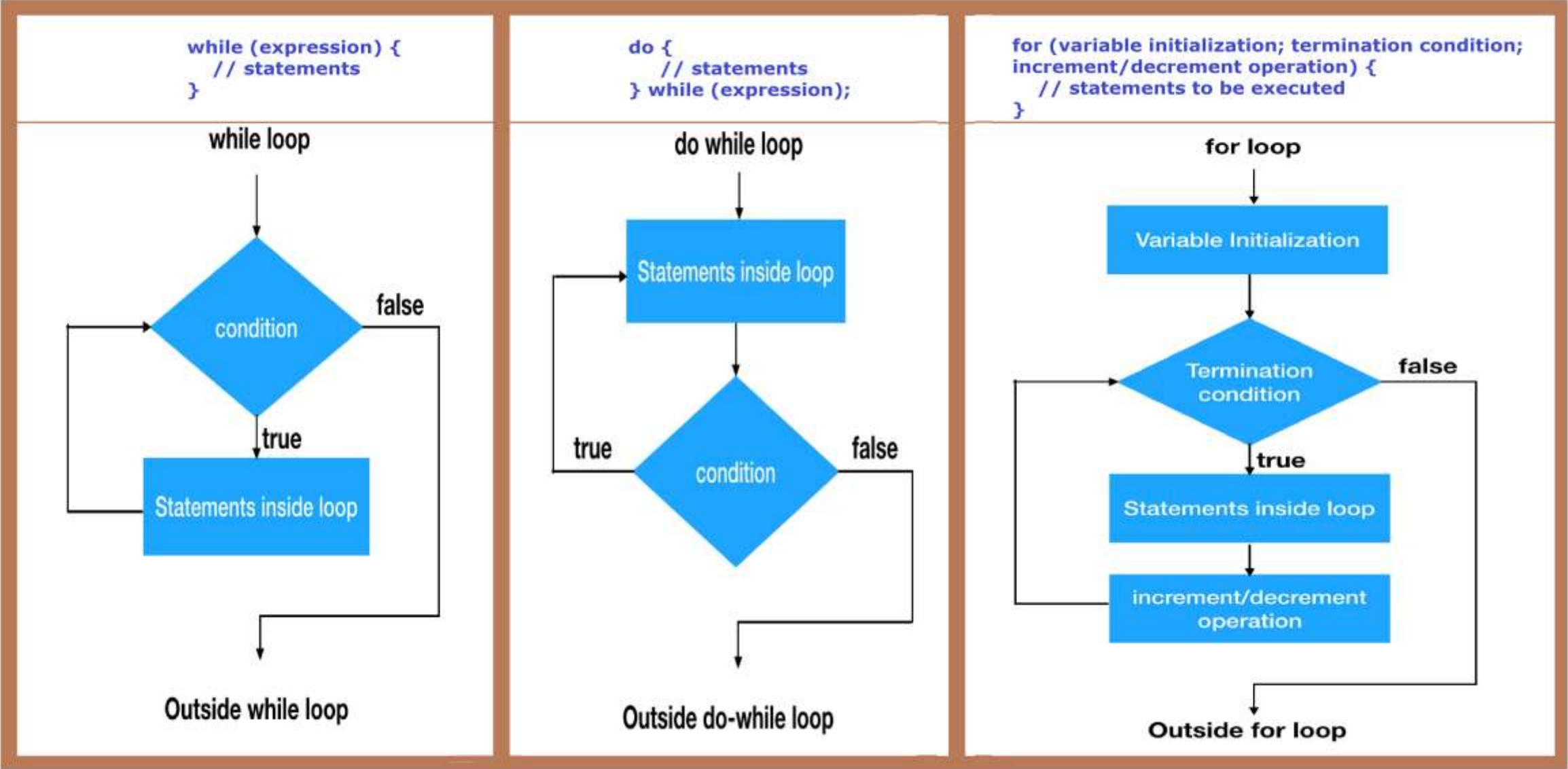
Rules for switch case in JAVA

- ▶ The following rules apply to a switch statement:
 - The variable used in a switch statement can only be a byte, short, int, char, or **(String from Java 1.7)**.
 - You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
 - The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
 - When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
 - When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
 - Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
 - A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Jump Statements

- ▶ The break Keyword:
 - The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.
 - The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.
- ▶ The continue Keyword:
 - The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.
 - In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
 - In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.
- ▶ Return Keyword:
 - A method may return a value. The return value type is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the return value type is the keyword void.
 - When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

Flow Chart for Loop Statements



Array

- ▶ The array, which stores a fixed-size sequential collection of elements of the same type.
- ▶ It allocate memory continuously .
- ▶ Index is used access the elements of array
- ▶ Index always begin from 0 to length-1.
- ▶ Raises exception for “ArrayIndexOutOfBoundsException” if index is invalid.
- ▶ Length property get the size of the array
- ▶ Array can be
 - One dimensional array
 - Storing single row or column values
 - Two dimensional array
 - Storing different structure of values like one table, multiple tables, and Su do Ku.

Array Syntax

```
public static void main(String[] args) {  
    //Array declaration  
    int[] arr1; /*or*/ int arr2[];  
  
    //Array instantiation  
    arr1=new int[5];  
  
    //Array Initialization  
    int[] arr3={10,20,30,40,50};  
  
    //store values using for loop  
    Scanner in=new Scanner(System.in);  
    for(int index=0;index<arr1.length;index++){  
        arr1[index]=in.nextInt();  
    }  
  
    //accessing values using for each loop  
    for(int val:arr3){  
        System.out.println(val);  
    }  
}
```

```
public static void main(String[] args) {  
    //two dimension Array declaration  
    int[][] arr1; /*or*/ int arr2[][];  
  
    //Array instantiation  
    arr1=new int[3][4];  
  
    //Array Initialization  
    int[][] arr3={{10,20,30},{40,50,60}};  
  
    //store values using for loop  
    Scanner in=new Scanner(System.in);  
    for(int row=0;row<arr1.length;row++){  
        for(int column=0;column<arr1[row].length;column++){  
            arr1[row][column]=in.nextInt();  
        }  
    }  
  
    //accessing values using for each loop  
    for(int[] row:arr3){  
        for(int ele:row){  
            System.out.println(ele);  
        }  
    }  
}
```

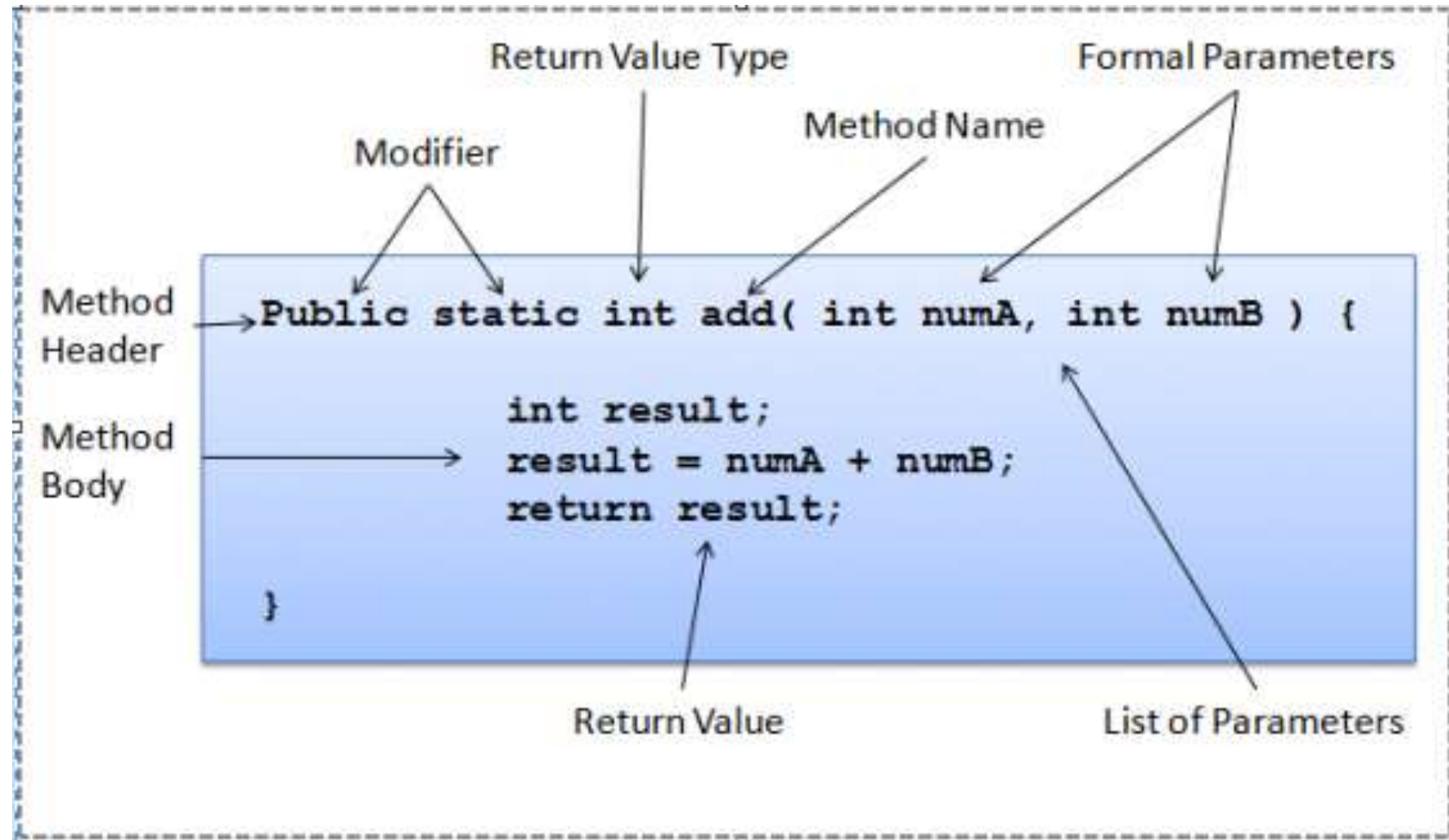
Arrays Class & arraycopy method

```
public static void main(String[] args) {  
    int[] arr3={15,10,30,20,50};  
    Arrays.sort(arr3);  
    System.out.println(Arrays.binarySearch(arr3,20));  
    Arrays.fill(arr3,100);  
    for(int val:arr3){  
        System.out.println(val);  
    }  
  
    int[] arr4=new int[5];  
    System.arraycopy(arr3/*source array*/,1/*index*/,arr4/*Destination array*/,0/*index*/,3/*length*/);  
}
```

Methods

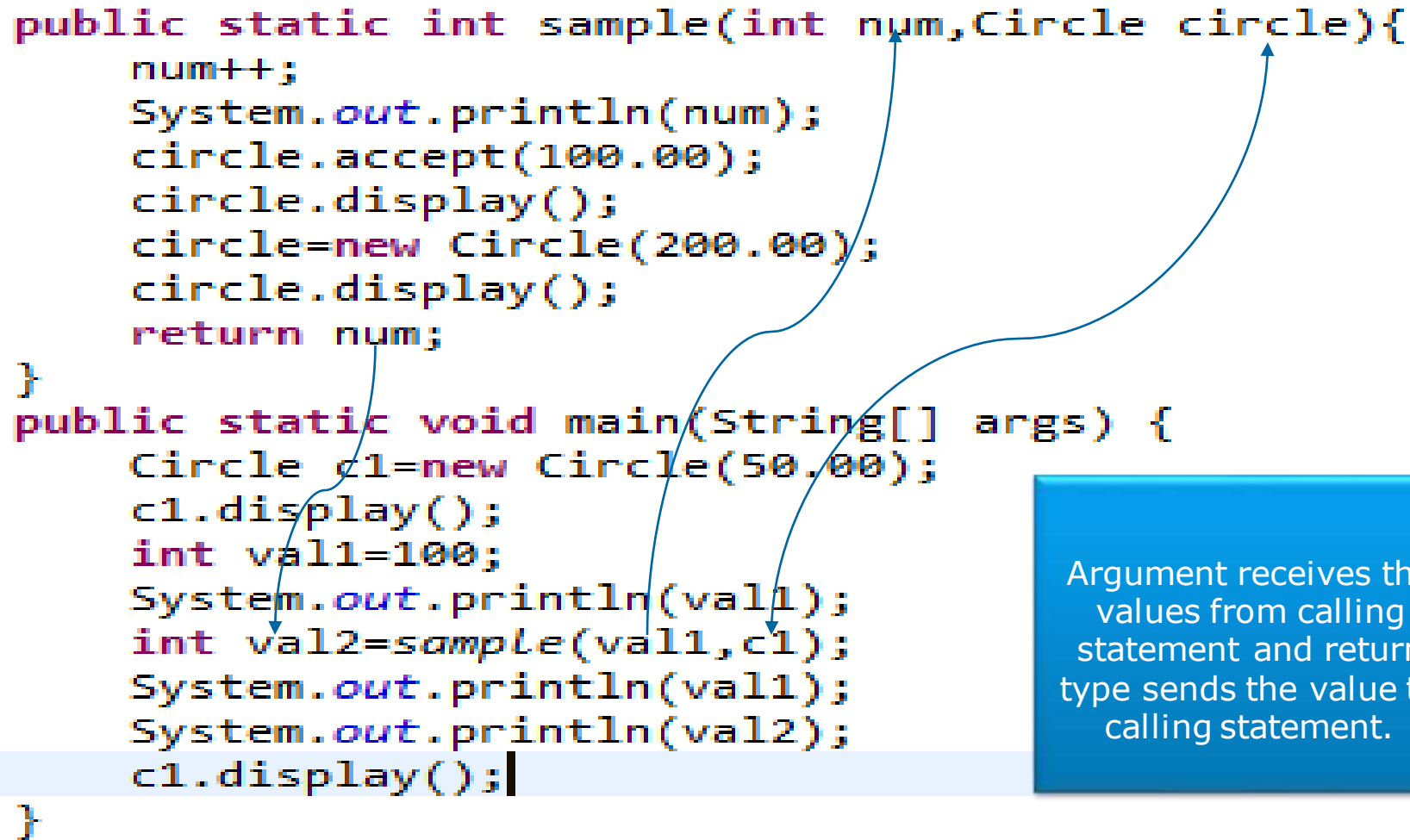
- ▶ A Java method is a collection of statements that are grouped together to perform an operation.
 - Modifiers: The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.
 - Return Type: A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`.
 - Method Name: This is the actual name of the method. The method name and the parameter list together constitute the method signature.
 - Parameters: A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.
 - Method Body: The method body contains a collection of statements that define what the method does.

Syntax of the Method



Pass By Value | Pass By Reference | Return value

```
public static int sample(int num, Circle circle){
    num++;
    System.out.println(num);
    circle.accept(100.00);
    circle.display();
    circle=new Circle(200.00);
    circle.display();
    return num;
}
public static void main(String[] args) {
    Circle c1=new Circle(50.00);
    c1.display();
    int val1=100;
    System.out.println(val1);
    int val2=sample(val1, c1);
    System.out.println(val1);
    System.out.println(val2);
    c1.display();
}
```



The diagram illustrates the flow of data between the `sample` and `main` methods. Blue arrows indicate the following:

- An arrow from `val1` in `main` to `num` in `sample`, representing pass-by-value.
- An arrow from `c1` in `main` to `circle` in `sample`, representing pass-by-reference.
- An arrow from the `return num;` statement in `sample` to `val2` in `main`, representing the return value.

Argument receives the values from calling statement and return type sends the value to calling statement.

Command Line Arguments

- ▶ Command line arguments allow the user to affect the operation of an application.
- **When invoking an application, the user types the command line arguments after the application name.**
- ▶ In the Java language, when you invoke an application, the runtime system passes the command line arguments to the application's main method via an array of Strings. Each String in the array contains one of the command line arguments.

C:/>java Demo5 arg1 arg2 arg3

```
public class Demo5 {  
    public static void main(String[] args) {  
        for(String s:args){  
            System.out.println(s);  
        }  
    }  
}
```

Packages in Java

- ▶ A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types.
 - Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
 - **Creating package**
`package com.syntel.mypackage.`
 - To use a public package member from outside its package, you must do one of the following:

Creating Package (Contd...)

- **Refer to the member by its fully qualified name**

```
java.util.Scanner in=new java.util.Scanner(System.in);
```

- **Import the package member**

```
import java.util.Scanner;import java.util.List;import java.util.LinkedList;
public class Demo {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        List<Dog> list=new LinkedList<Dog>();
    }
}
```

Creating Package (Contd...)

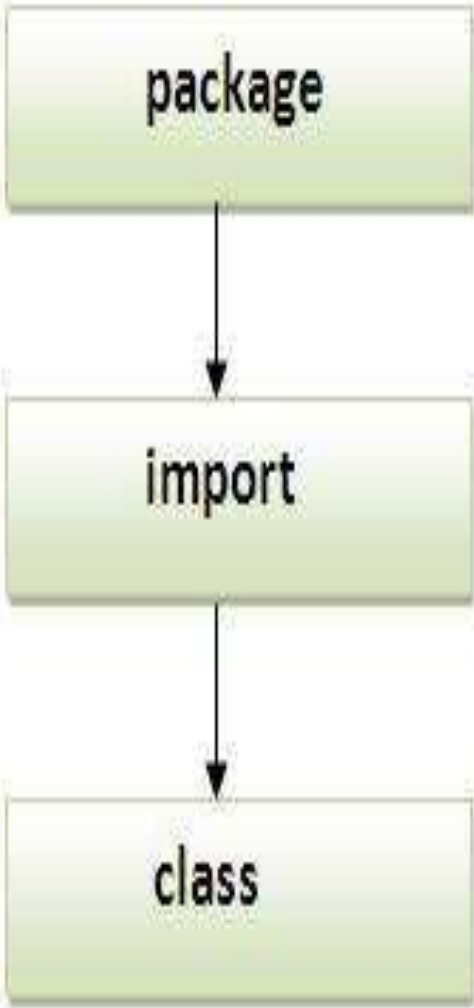
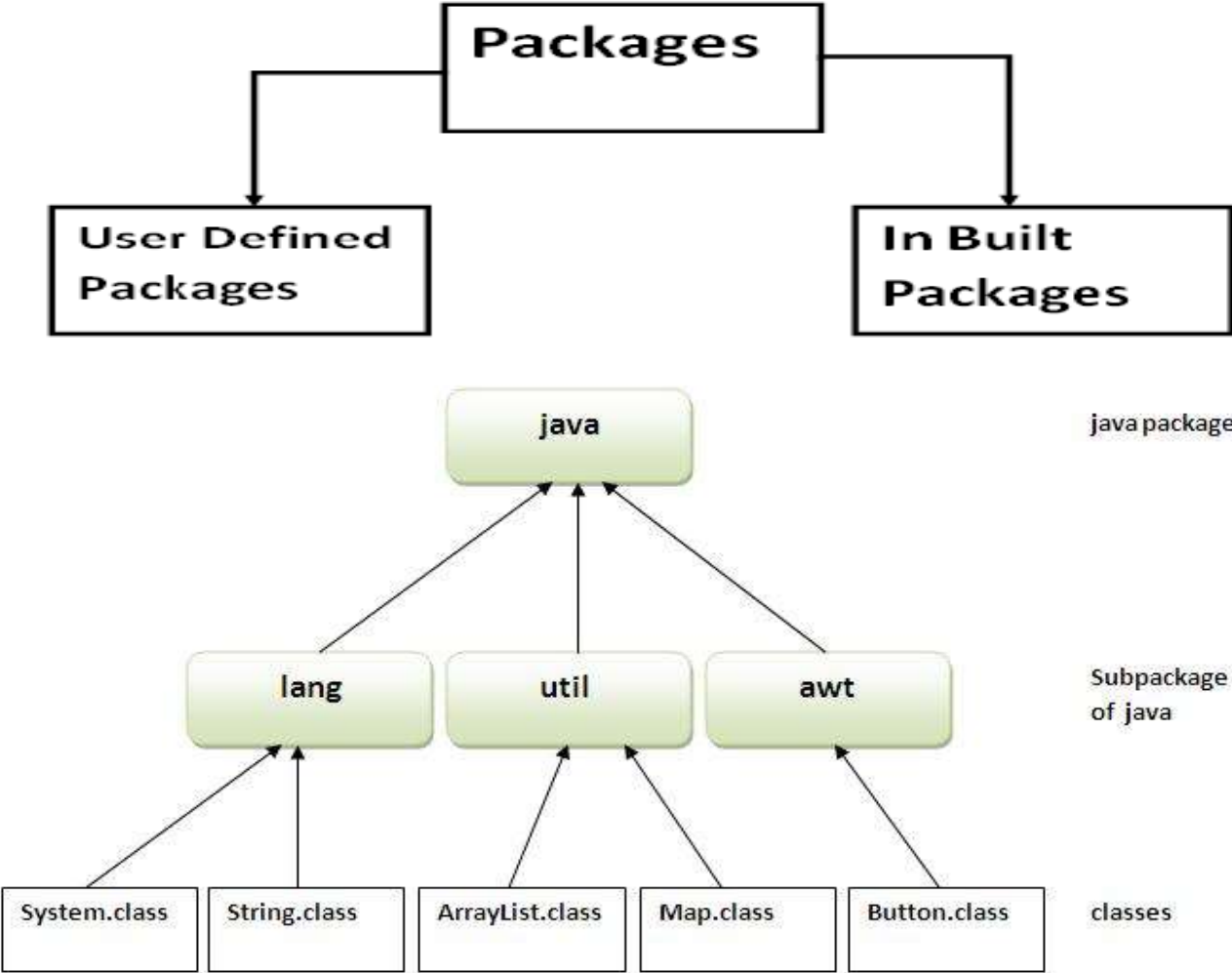
- **Import the entire members of package**

```
import java.util.*;
public class Demo {
    public static void main(String[] args) {
        Scanner in=new Scanner(System.in);
        List<Dog> list=new LinkedList<Dog>();
    }
}
```

- **import only static members of Class introduced from Java 1.5**

```
import static java.util.Map.*;
public class Demo {
    public static void main(String[] args) {
        //this is static member of class Map
        Entry<Integer,String> entry;
    }
}
```

Package in Java



OOPs Concept in Java

What is an Object?

What is an Object?

- ▶ How many people thought:
“An object is a chunk of code consisting of data and the methods/functions that manipulate that data?”
- ▶ That’s o.k., but we can do better...

What is an Object?

- ▶ An object is a “software machine” that:
 - is capable of performing, on request, one or more specialized functions (i.e., it provides services)
 - has explicit interfaces through which it interacts with other objects (provides (and requests!) services)
 - “hides” its implementation from other objects
- Bran Selic, V.P. R & D,
ObjecTime Limited

What is an Object?

- ▶ This definition includes:
 - abstract data types (stacks, queues, complex numbers, customer records)
 - acknowledge-timeout-retry protocols for computer communications software
 - finite state machines
 - layers & subsystems in large systems
- ▶ The object paradigm is based on the idea that we can construct software by assembling communities of interacting objects

Properties of Objects

- ▶ An object has:
 - state
 - behaviours
 - identity

Classes

- ▶ A class defines the characteristics that are common to a group of similar objects
 - defines the variables that hold the object's state
 - defines the methods that describe the object's behaviours

Classes

- ▶ A class is used as a template or pattern to create new objects (instances of the class)
- ▶ Every object is an instance of a class
 - each object has its own name (identity)
 - each object has its own variables (state)
 - all instances of the class share the methods defined by the class (behaviours)

Example: A Complex Number Class

- ▶ Suppose we want to create and manipulate complex numbers
 - a complex number is a value of the form:

$$a + bi$$

$$\text{where } i^2 = -1$$

Example: A Complex Number Class

- ▶ Using an OO programming language, we define a class that allows us to create and manipulate objects that represent complex number values
- ▶ Initial design: what state and behaviours are common to all complex numbers?
- ▶ State
 - real part
 - imaginary part

Example: A Complex Number Class

- ▶ Behaviours
 - initialize a new complex number
 - mathematical operations (add two complex numbers, etc.)
 - relational operations (equal, not equal, etc.)
 - set the values of the real and imaginary parts
 - get the values of the real and imaginary parts
 - others...

Implementing a Complex Number Class in Java

- An overview of a typical Java implementation, contained in the file Complex.java

```
public class Complex implements Cloneable
{
    // State (instance variables)
    // Behaviour (instance methods)
    // constructors
    // accessors
    // mutators
    // operations/messages
}
```

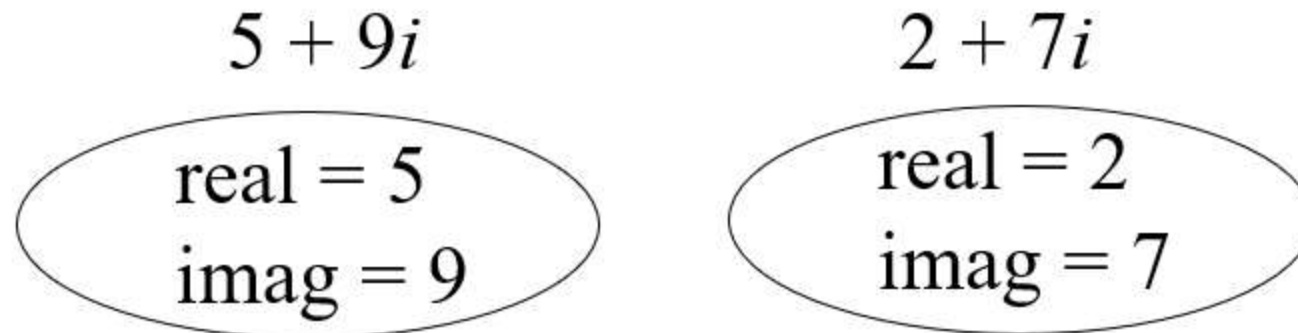
Implementing State

- ▶ An object's state is contained in instance variables

```
// State
```

```
private double real;  
private double imag;
```

- ▶ Each object created from this class will have separate occurrences of the instance variables



Implementing Behaviour : Constructors

- Constructors are methods that describe the initial state of an object created of this class.

Default Constructor

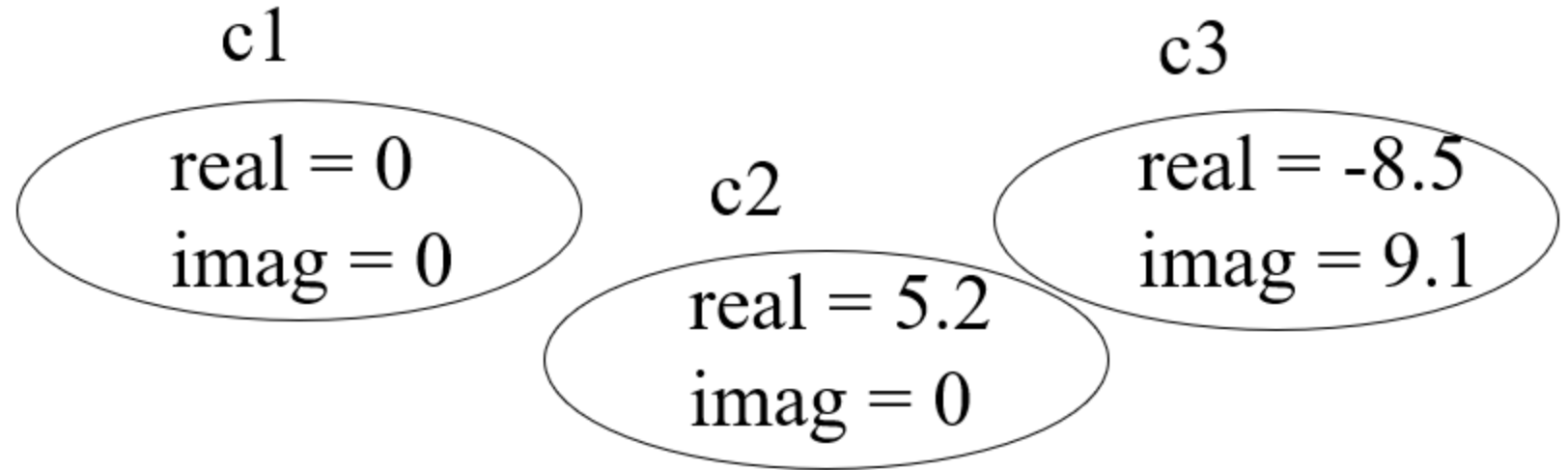
```
public Complex()  
{  
    real = 0.0; imag = 0.0;  
}  
public Complex(double rp)  
{  
    real = rp; imag = 0.0;  
}  
public Complex(double rp, double ip)  
{  
    real = rp; imag = ip;  
}
```

Implementing Behaviour : Constructors

- ▶ Constructors are invoked only when the object is created. They cannot be called otherwise.

```
Complex c1, c2, c3;
```

```
c1 = new Complex();  
c2 = new Complex(5.2);  
c3 = new Complex(-8.5, 9.1);
```



- ▶ Constructors initialise the state of an object upon creation.

Implementing Behaviour : Accessor Methods

- ▶ Objects often have methods that return information about their state
- ▶ Called accessors or getters (because the method name begins with “get”)
- ▶ `getReal/Imag()` returns the real/imag part of the Complex object

```
public double getReal()  
{  
    return real;  
}  
public double getImag()  
{  
    return imag;  
}
```

Implementing Behaviour :

Mutator Methods

- ▶ Objects often have methods that change aspects of their state
- ▶ called mutators or setters (because the method name begins with "set")
- ▶ setReal/Imag() changes the real/imag part of the Complex object

```
public void setReal(double rp)
{
    real = rp;
}
public void setImag(double ip)
{
    imag = ip;
}
```

Exercise : Suggest another mutator ?

Implementing Behaviour : Operations

- ▶ Unlike C++, Java does not let us overload the + operator to perform addition of complex numbers
- ▶ Instead, we define a plus() method as one of the arithmetic operations supported by the Complex class:

```
public Complex plus(Complex c)
{
    Complex sum =
        new Complex(real + c.real,
                    imag + c.imag);
    return sum;
}
```

- ▶ We'll explain how this method works later...

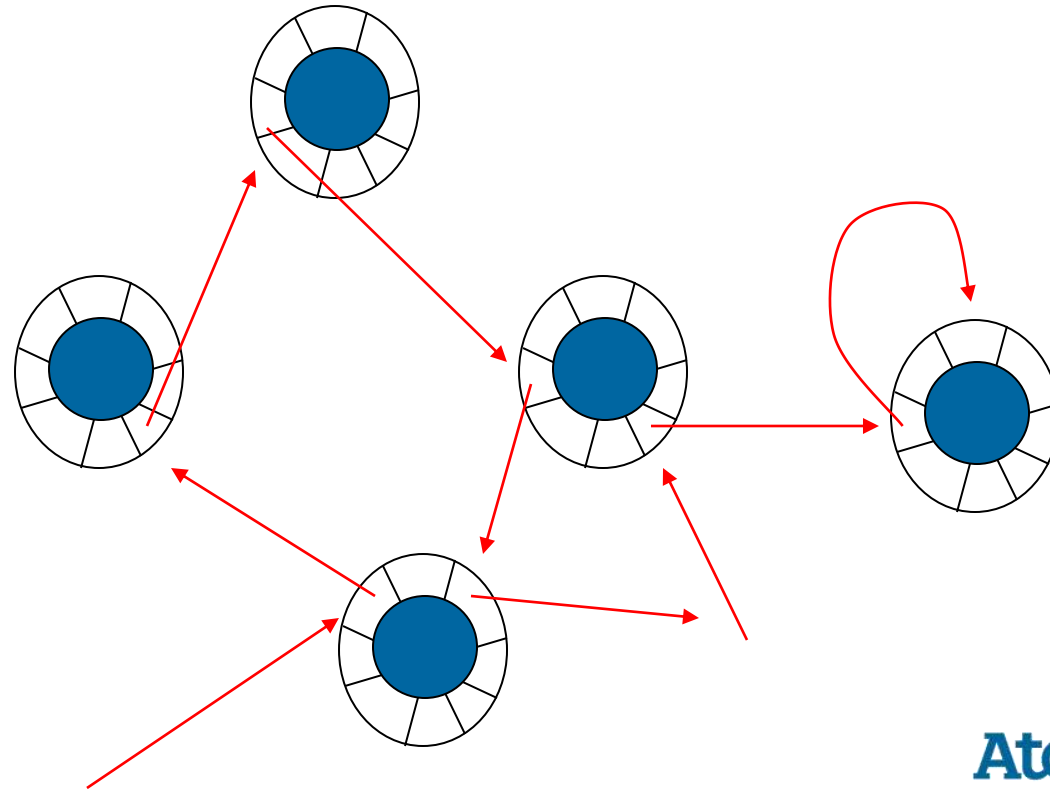
Implementing Behaviour : Message Passing

- ▶ The last category of behaviours – Operations – leads us to the bigger picture !
 - The example of Complex.java considers only a single class in isolation

What is an object-oriented application ?

Message Passing

- ▶ An application is a set of cooperating objects that communicate with each other
- ▶ Objects are not solitary!
- ▶ Objects interact by passing messages to each other
- ▶ In Java, "message passing" is performed by calling methods



Message Passing

- ▶ A message is sent to an object using the “dot notation”:

```
c1.setReal ( 2.3 );  
double I = c2.getReal ( );
```

- ▶ These look like simple function calls, BUT ...To fully use the object orientation, a programmer reads these statements this way:
 - “send the setReal message with argument 2.3 to the object referred to by c1”
 - “send the getReal message to the object referred to by c2”
- ▶ Often, we say that the setReal and getReal methods are invoked (not called)

Message Passing

- ▶ When an object receives a message, it looks for the corresponding method (ie. with the same signature) in its class.
- ▶ The object then invokes and executes the method
- ▶ This is known as run-time binding or late binding or dynamic binding
 - the method that is executed when a message is sent to an object is determined when the program is executed, not when it is compiled
- ▶ The static binding of the Pascal/C/C++ function call/return mechanism seems simpler..
 - ...but the benefits of run-time binding will become apparent when we look at inheritance and polymorphism

Object Oriented Design

- ▶ The general goal of OO design is to:
 - Decouple classes as much as possible
 - Must think ahead about how the internal implementation of a class may change
 - Want to avoid having those changes propagated to users of the class interface (i.e. the set of public methods)
 - Ideally, the public interface should never change
- ▶ Accessor functions should be used to control access to internal data structures
 - Lets the internal data representation change without affecting the users of the class (recall setA, getA, etc.)
 - Accessor functions can be much more sophisticated than we have seen so far

Caveat: Misusing Setter and Getter Methods

- ▶ Novice OO programmers (especially those with prior programming experience with a non-OOP language) sometimes think of objects as analogous to Pascal records or C structs (i.e., aggregations of variables), with setter and getter methods provided to “overcome” information hiding
- ▶ Example: An application needs to add two complex numbers (see code on next slide)
 - getters are invoked to obtain the real and imaginary values of the operands
 - complex addition is performed by the application
 - setters are invoked to store the result

Caveat: Misusing Setter and Getter Methods

```
// c1, c2, and c3 contain references to
// Complex objects.
...
// c3 = c1 + c2
double r1 = c1.getReal();
double i1 = c1.getImag();
double r2 = c2.getReal();
double i2 = c2.getImag();
c3.setReal(r1 + r2);
c3.setImag(i1 + i2);
```


Caveat: Misusing Setter and Getter Methods

- ▶ The proper object-oriented approach is to recognize that Complex objects should be responsible for knowing how to add complex numbers
 - class Complex should provide a method that implements the required addition behaviour (which is what our class does, through the plus() method)
 - the application program requests a complex number to add itself to another complex number by sending it a message:

```
c3 = c1.plus(c2);
```

c3 = c1.plus(c2);

Complex number c1: give me a new complex number that contains the sum of your value and the value of complex number c2



c2

4.9 + 0.0i

c1

5.1 - 8.6i

Object-Oriented Programming Languages

- ▶ To be considered object-oriented, a programming language must support
 - Encapsulation
 - Abstraction/information hiding
 - Inheritance
 - Polymorphism
- ▶ How does Java stack up with regards to the first three (we'll look at polymorphism later)?

How Does Java Support Encapsulation?

- ▶ In each Complex object, the variables and the methods that operate on those variables are grouped into a single entity
- ▶ Java source code promotes this view
 - instance variables and instance methods defined in the same class definition
 - compare with C++

How Does Java Support Information Hiding?

- ▶ Instance variables are declared private
 - a Complex object's private variables can only be accessed by methods defined in class Complex
 - they cannot be accessed by methods defined in another class

How Does Java Support Information Hiding?

- ▶ Instance methods are declared public
 - a Complex object's public methods can be invoked by any object, but...
 - the method signatures (name, parameter list, return type) do not reveal the methods' algorithms and implementation details (these are hidden if you don't have source code)

Information Hiding in Classes

- ▶ Always use private variables and provide interface methods as required
- ▶ Need this protection in case there are constraints that must be imposed on the variables
 - In the imaginary PairClass, what if limits must be imposed on a and b?
 - setA and setB can do this without affecting the interface (thus users) of objects of the type PairClass

```
void setA (int val)
{
    a = (val <= 25)?val:25;
}
```

- What if a must now also be less than 75% of b?
 - No changes are reflected to the user.

How Does Java Support Inheritance?

- ▶ Unless otherwise specified, every class is a subclass of class Object
 - it is also possible to specify superclass/subclass relationships using the extends keyword (later...)
- ▶ This means that class Complex is a specialized form of class Object
 - class Complex inherits variables and methods that are appropriate for all objects, from class Object
 - class Complex defines additional variables and methods that are appropriate for all complex numbers

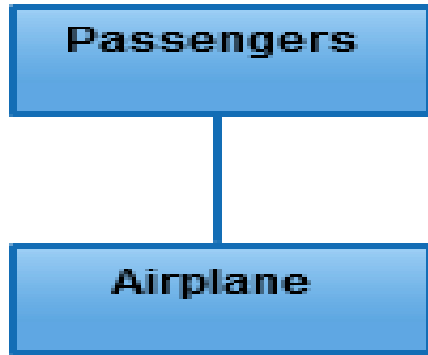
Object Oriented Concepts

Objective

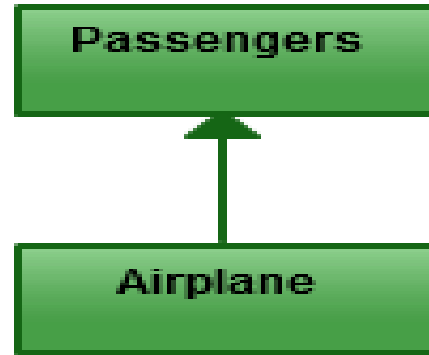
- ▶ Type of Relationships in JAVA
- ▶ Inheritance
- ▶ Type of Inheritances
- ▶ Constructor in Inheritance
- ▶ Object Type Casting
- ▶ Polymorphism
- ▶ Over Loading
- ▶ Overriding
- ▶ Abstract Class
- ▶ Abstract Method
- ▶ Final



Type of Relationships in JAVA



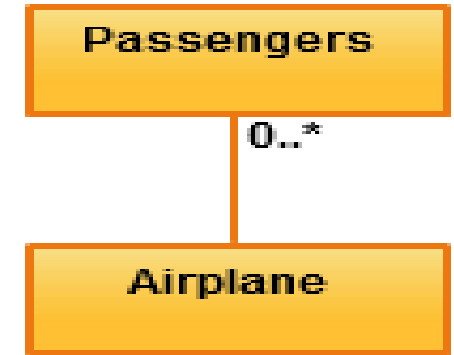
Association



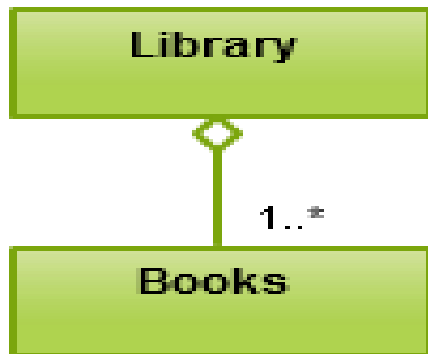
Directed Association



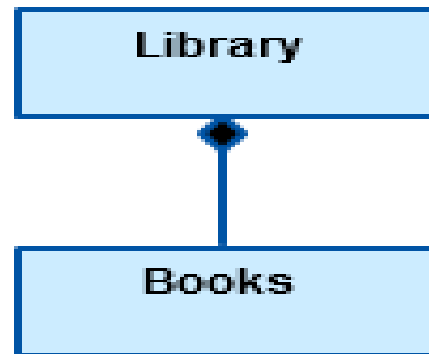
Reflexive Association



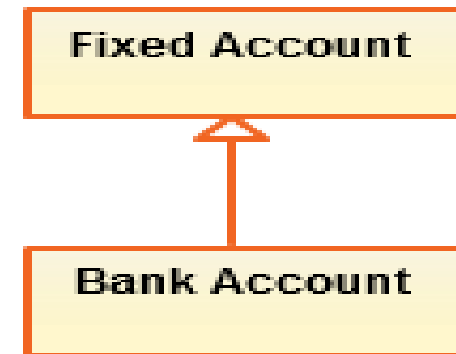
Multiplicity



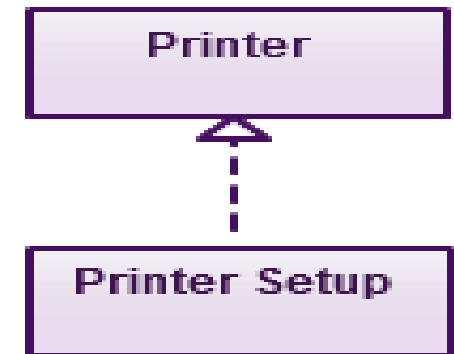
Aggregation



Composition



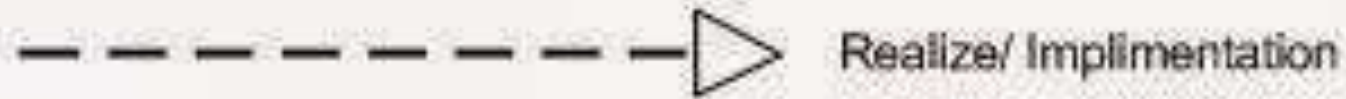
Inheritance



Realization

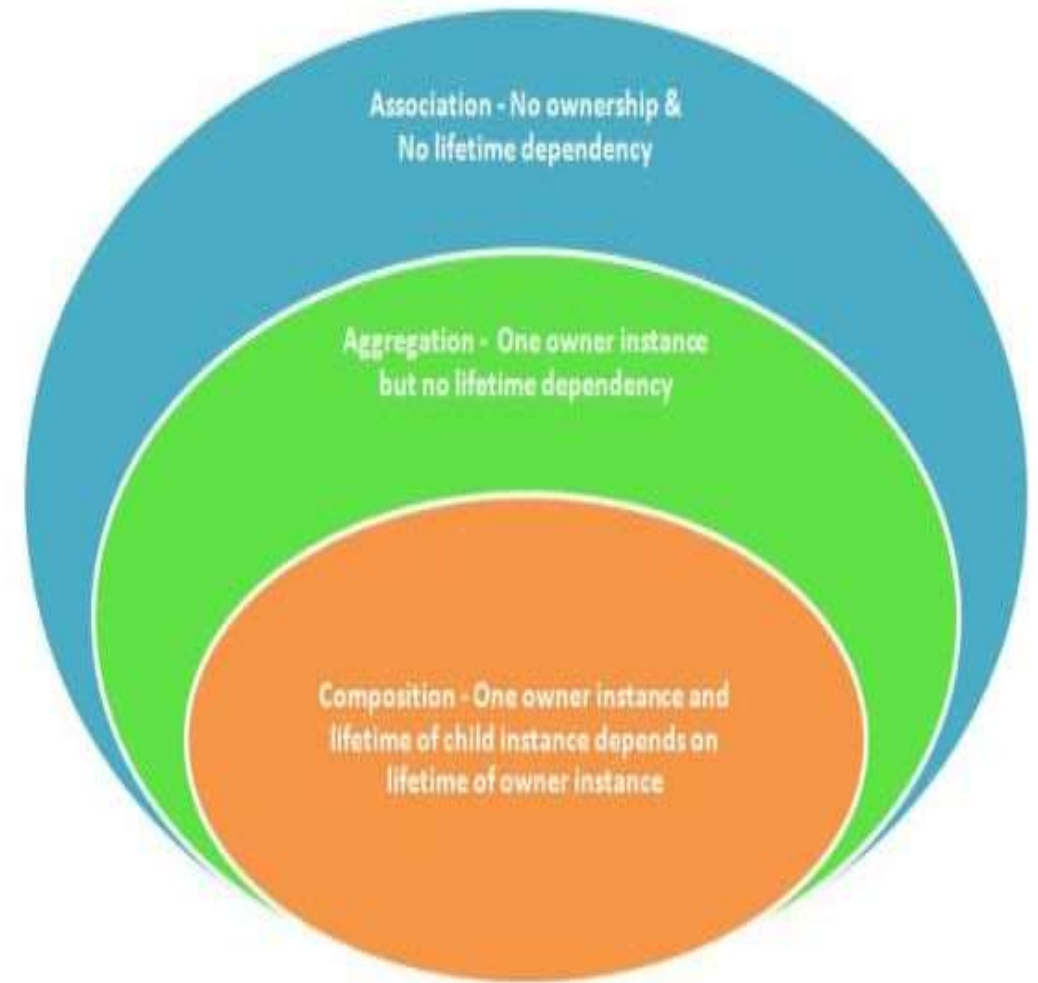
UML Notations for Relationship

UML Notations:



Association

- ▶ Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.
- ▶ In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object.
- ▶ Composition and Aggregation are the two forms of association.



Coding For Association

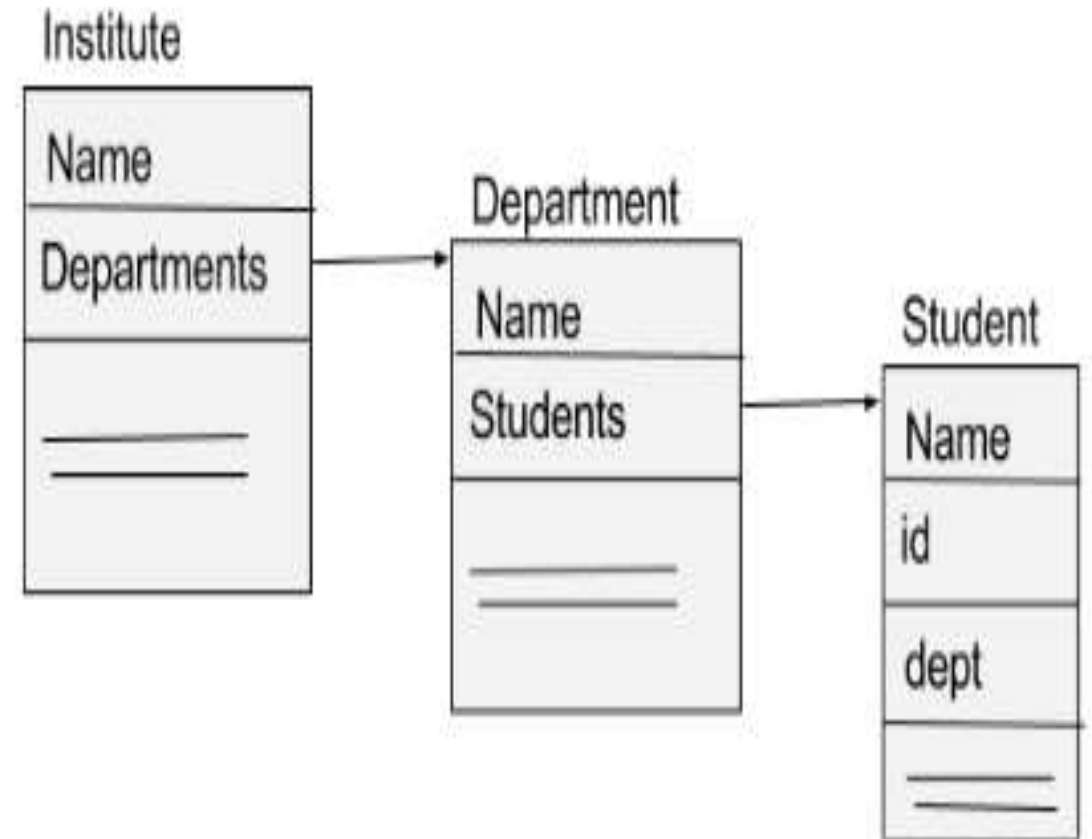
```
public class Bank {  
    private String bankName;  
    public Bank(String bankName) {  
        this.bankName=bankName;  
    }  
    public String getBankName() {  
        return bankName;  
    }  
}
```

```
public class Employee {  
    private String employeeName;  
    public Employee(String employeeName) {  
        this.employeeName=employeeName;  
    }  
    public String getEmployeeName() {  
        return employeeName;  
    }  
}
```

```
// Association between both the  
//classes in main method  
public class Association  
{  
    public static void main (String[] args)  
    {  
        Bank bank = new Bank("Axis");  
        Employee emp = new Employee("Neha");  
  
        System.out.println(emp.getEmployeeName() +  
            " is employee of " + bank.getBankName());  
    }  
}
```

Aggregation

- ▶ It is a special form of Association
- ▶ It represents Has-A relationship.
- ▶ It is a unidirectional association i.e. a one way relationship.
- ▶ In Aggregation, both the entries can survive individually which means ending one entity will not effect the other entity



Coding For Aggregation

```
public class Student {  
    private String name;  
    private int id;  
    private String dept;  
    public Student(String name, int id, String dept)  
    {  
        this.name = name;  
        this.id = id;  
        this.dept = dept;  
    }  
    public void print(){  
        System.out.println("Student Details "+name+" "+id+" "+dept);  
    }  
}
```

```
import java.util.List;  
public class Department {  
    private String name;  
    private List<Student> students;  
    public Department(String name, List<Student> students){  
        this.name = name;  
        this.students = students;  
    }  
    public List<Student> getStudents(){  
        return students;  
    }  
}
```

```
import java.util.List;  
public class Institute {  
    private String instituteName;  
    private List<Department> departments;  
    public Institute(String instituteName, List<Department> departments){  
        this.instituteName = instituteName;  
        this.departments = departments;  
    }  
    // count total students of all departments  
    // in a given institute  
    public int getTotalStudentsInInstitute()  
    {  
        int noOfStudents = 0;  
        List<Student> students;  
        for(Department dept : departments){  
            students = dept.getStudents();  
            for(Student s : students){  
                noOfStudents++;  
            }  
        }  
        return noOfStudents;  
    }  
}
```


Composition

- ▶ Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.
- ▶ It represents part-of relationship.
- ▶ In composition, both the entities are dependent on each other.
- ▶ When there is a composition between two entities, the composed object cannot exist without the other entity.

Aggregation vs. Composition

- ▶ Dependency: Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don't exist separate to a Human
- ▶ Type of Relationship: Aggregation relation is "has-a" and composition is "part- of" relation.
- ▶ Type of association: Composition is a strong Association whereas Aggregation is a weak Association.

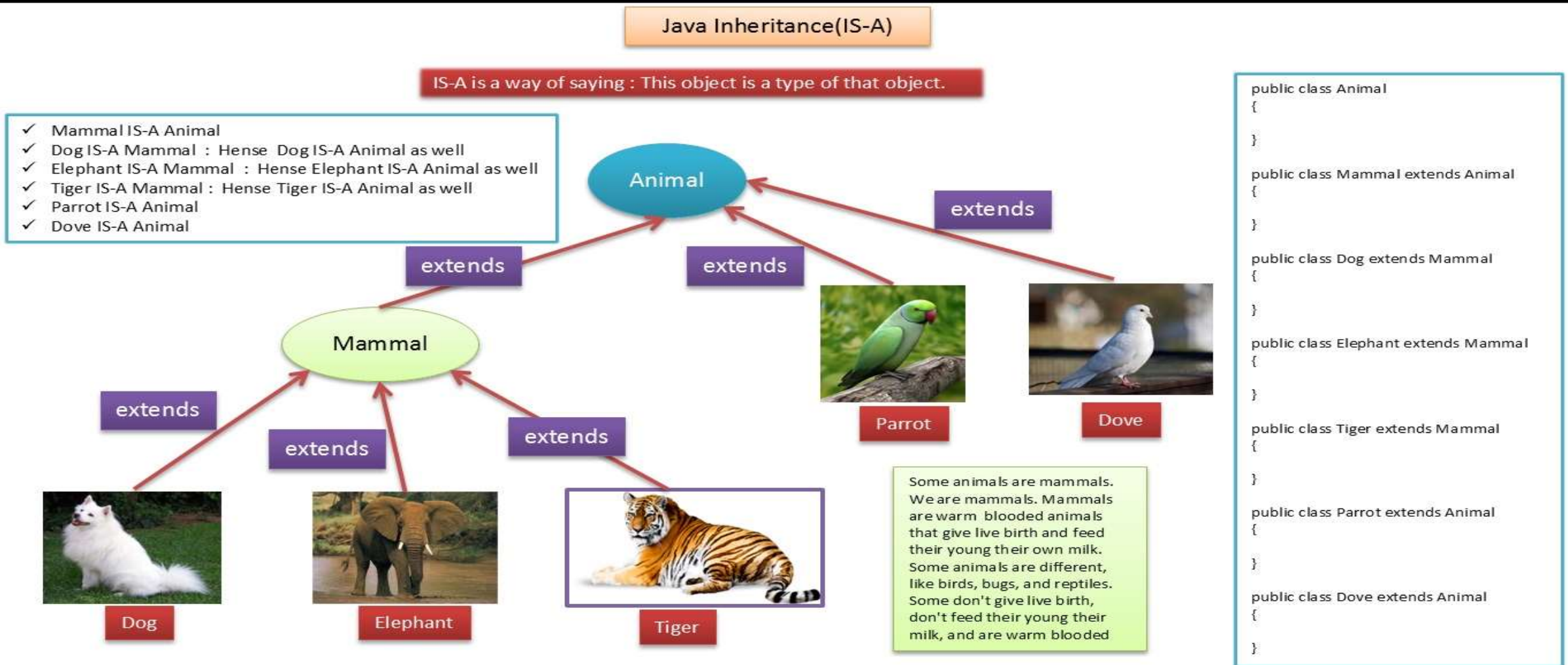
Coding For Composition

```
//Engine class which will be used by car. so 'Car'  
//class will have a field of Engine type.  
public class Engine {  
    // starting an engine.  
    public void work(){  
        System.out.println("Engine of car has been started ");  
    }  
}
```

```
public class Composition {  
    public static void main(String[] args) {  
        // making an engine by creating  
        // an instance of Engine class.  
        Engine engine = new Engine();  
        // Making a car with engine.so we are passing a engine  
        //instance as an argument while creating instance of Car.  
        Car car = new Car(engine);  
        car.move();  
    }  
}
```

```
public class Car {  
    // For a car to move, it need to have a engine.  
    private final Engine engine; // Composition  
    Car(Engine engine){ // Aggregation  
        this.engine = engine;  
    }  
    // car start moving by starting engine  
    public void move(){  
        engine.work();  
        System.out.println("Car is moving ");  
    }  
}
```

Inheritance



Creating Objects for Inheritance classes

► `Animal animal=new Animal();`

**Instance of
Animal**

► `Dog dog=new Dog();`

**Instance of
Animal**

**Instance of
Mammal**

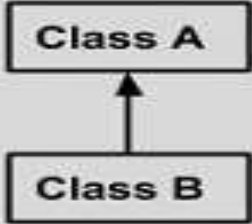
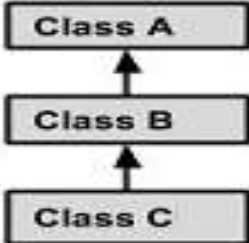
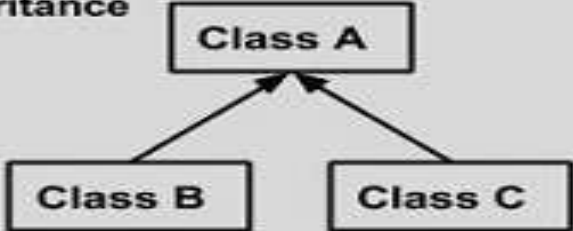
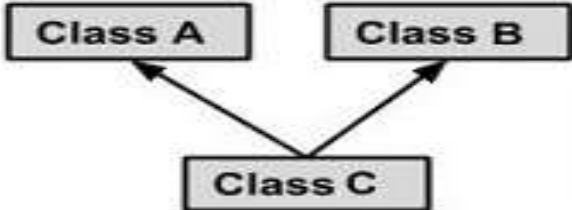
Instance of Dog

► `Parrot parrot=new Parrot();`

**Instance of
Animal**

**Instance of
Parrot**

Type of Inheritances

Single Inheritance	 <pre>graph BT; B[Class B] --> A[Class A];</pre>	<pre>public class A { } public class B extends A { }</pre>
Multi Level Inheritance	 <pre>graph BT; C[Class C] --> B[Class B]; B --> A[Class A];</pre>	<pre>public class A {} public class B extends A {.....} public class C extends B {.....}</pre>
Hierarchical Inheritance	 <pre>graph BT; B[Class B] --> A[Class A]; C[Class C] --> A;</pre>	<pre>public class A {} public class B extends A {.....} public class C extends A {.....}</pre>
Multiple Inheritance	 <pre>graph BT; C[Class C] --> A[Class A]; C --> B[Class B];</pre>	<pre>public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance</pre>

Important terminology in Inheritance

- ▶ Super Class: The class whose features are inherited is known as super class(base, or parent class).
- ▶ Sub Class: The class that inherits the other class is known as sub class(derived, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- ▶ Reusability: Inheritance supports the concept of “reusability”.
- ▶ Default superclass: Except Object class, which has no superclass. In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

Important terminology in Inheritance

- ▶ Superclass can only be one: A superclass can have any number of subclasses. But a subclass can have only one superclass.
- ▶ Inheriting Constructors: A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- ▶ Private member inheritance: A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods (getters and setters) for accessing its private fields.

What all can be done in a Subclass?

- ▶ The inherited fields can be used directly, just like any other fields.
- ▶ We can declare new fields in the subclass that are not in the superclass.
- ▶ The inherited methods can be used directly as they are.
- ▶ We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, toString() method is overridden).
- ▶ We can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.

What all can be done in a Subclass?

- ▶ We can declare new methods in the subclass that are not in the superclass.
- ▶ We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

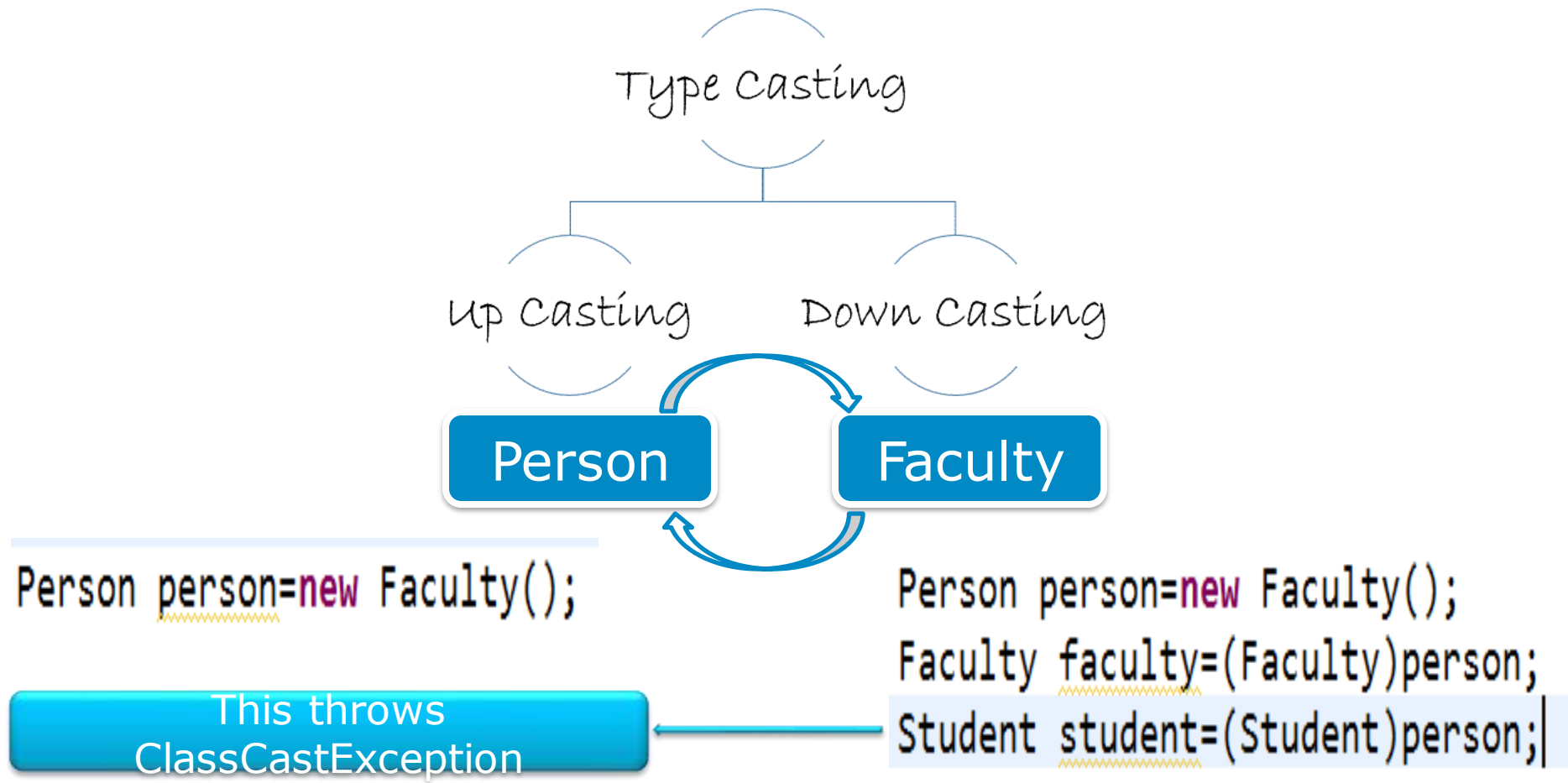
Constructor in Inheritance

```
public class Person {  
    private int id;  
    private String name;  
    public Person() {  
        this.id=0;  
        this.name="";  
    }  
    public Person(int id, String name) {  
        super();  
        this.id = id;  
        this.name = name;  
    }  
}
```

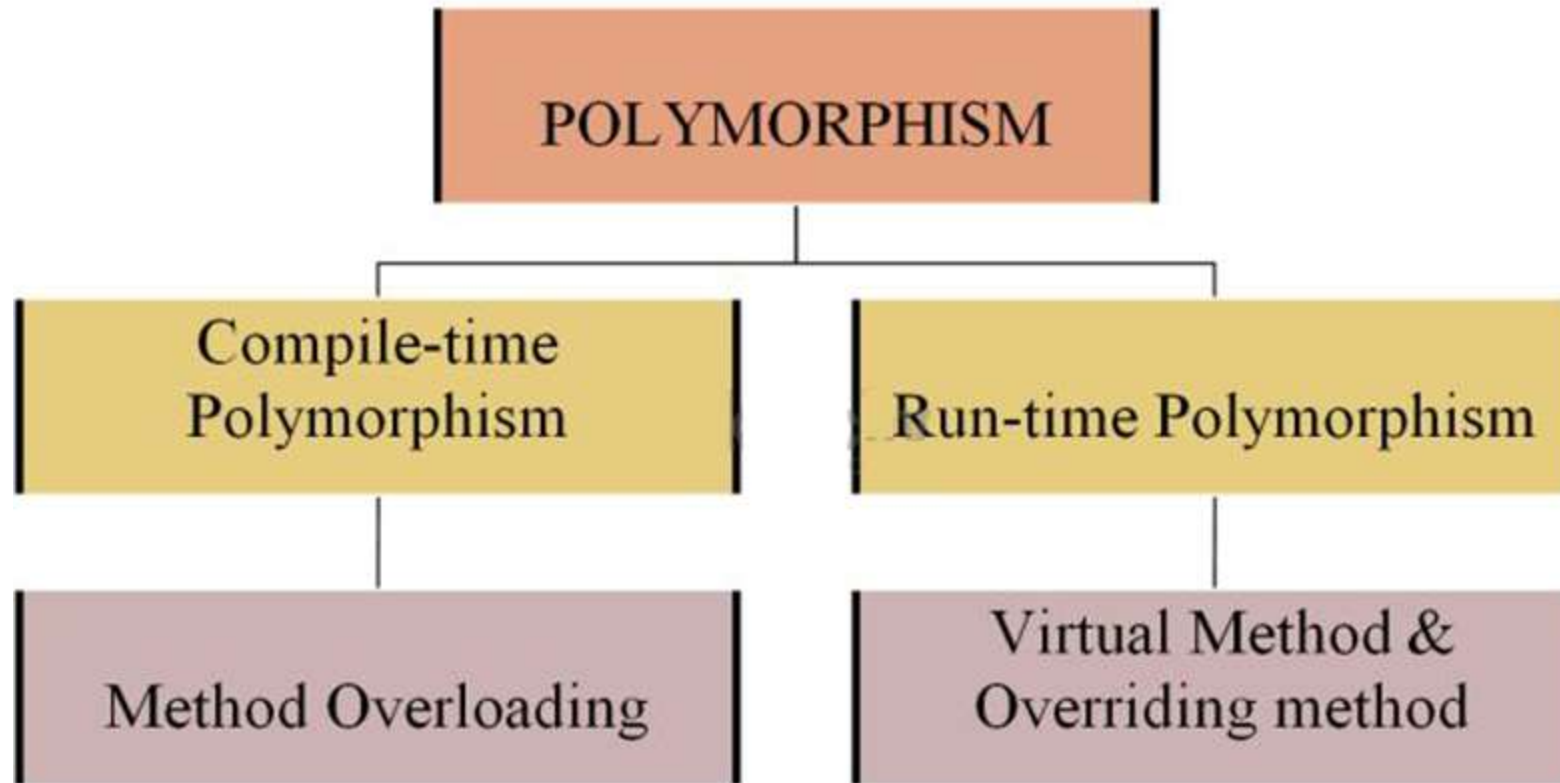
```
public class PersonMain {  
    public static void main(String[] args) {  
        Faculty faculty1=new Faculty();  
        Faculty faculty2=new Faculty(100,"Jhon","JAVA");  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public Faculty() {  
        //Here it is calling the default constructor of the Super class  
        this.subject="";  
    }  
    public Faculty(int id, String name,String subject) {  
        super(id,name);//This is calling the two argument constructor of Super class  
        this.subject = subject;  
    }  
}
```

Object Type Casting



Polymorphism



Polymorphism

Compile-time Polymorphism	Run-time Polymorphism
Is implemented through method overloading .	Is implemented through method overriding .
Is executed at the compile-time since the compiler knows which method to execute depending on the number of parameters and their data types.	Is executed at run-time since the compiler does not know the method to be executed, whether it is the base class method that will be called or the derived class method.
Is referred to as static polymorphism.	Is referred to as dynamic polymorphism.

Method Overloading

- ▶ If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
- ▶ If we have to perform only one operation, having same name of the methods increases the readability of the program
- ▶ Advantage of method overloading
 - Method overloading increases the readability of the program.
- ▶ Different ways to overload the method
 - There are two ways to overload the method in java
 - By changing number of arguments
 - By changing the data type

Method Overloading

```
public class Person {  
    private int id;  
    private String name;  
    public void store() {  
        this.id=0;  
        this.name="";  
    }  
    public void store(int id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public void store(int id, String name,String subject) {  
        super.store(id,name);  
        this.subject = subject;  
    }  
    public void store(String subject) {  
        super.store();  
        this.subject = subject;  
    }  
}
```

```
public class PersonMain {  
    public static void main(String[] args) {  
        Faculty people=new Faculty();  
        people.store();  
        people.store(100,"Jhon","Java");  
    }  
}
```


Method Overriding

- ▶ If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java.
- ▶ Usage of Java Method Overriding
 - Method overriding is used to provide specific implementation of a method that is already provided by its super class.
 - Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- ▶ method must have same name as in the parent class
- ▶ method must have same parameter as in the parent class.
- ▶ must be IS-A relationship (inheritance).
- ▶ If it is a private method in the Parent class, it cannot be overridden.
- ▶ Overriding method should have more access or equal access than Parent class method which is overridden.

Method Overriding

```
public class Person {  
    private int id;  
    private String name;  
    public void store() {  
  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public void store() {  
  
    }  
}
```

```
public class PersonMain {  
    public static void main(String[] args) {  
        Person people=new Faculty();  
        people.store();  
    }  
}
```

Method Hiding

- ▶ If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in java. Both methods must be static methods.

```
public class Person {  
    private int id;  
    private String name;  
    public static void store() {  
  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public static void store() {  
  
    }  
}
```

```
public class PersonMain {  
    public static void main(String[] args) {  
        Person.store();  
        Faculty.store();  
    }  
}
```

Abstract Class

- ▶ There is a super class used it only by sub classes and creating instance of the class is no use. It should a abstract class.
- ▶ A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).
- ▶ Abstract classes may or may not contain abstract methods, i.e., methods without body (`public void get();`)
- ▶ But, if a class has at least one abstract method, then the class must be declared abstract.
- ▶ If a class is declared abstract, it cannot be instantiated.
- ▶ To use an abstract class, you have to inherit it from another class.
- ▶ If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract Class

```
public abstract class Person {  
    private int id;  
    private String name;  
    public static void store() {  
  
    }  
}
```

```
public class PersonMain {  
    public static void main(String[] args) {  
        Person person=new Person();  
        person=new Faculty();  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public static void store() {  
  
    }  
}
```

Abstract Method

- ▶ If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.
- ▶ `abstract` keyword is used to declare the method as abstract.
- ▶ You have to place the `abstract` keyword before the method name in the method declaration.
- ▶ An abstract method contains a method signature, but no method body.
- ▶ Instead of curly braces, an abstract method will have a semi colon (;) at the end.

Abstract Method

```
public abstract class Person {  
    private int id;  
    private String name;  
    public abstract void accept();  
    public void store() {  
    }  
}
```

```
public class Faculty extends Person{  
    private String subject;  
    public void accept() {  
    }  
    public void store() {  
    }  
}
```

```
public class Student extends Person{  
    private String dept;  
    public Student(String name, int id, String dept){  
        this.dept = dept;  
    }  
}
```


Final

- ▶ Final variables are variables that cannot be changed
- ▶ we can have final method. Means method that cannot be changed. Behaviour of a method can only be changed by overriding it in another class. So, final methods are not allowed to override.
- ▶ Final class is a class that cannot be extended

Notes on final keyword in Java :

- ▶ Final keyword can be applied to variable, method and class in Java.
- ▶ Final variables cannot be changed, final methods cannot be override and final class cannot be extended.
- ▶ Final variables should be initialised always. At the time of declaration, inside constructor, inside static method (for static final variables) or inside instance initializer block.
- ▶ A constructor cannot be final
- ▶ All variables declared inside interface are final
- ▶ Using final variables, methods and classes in Java improves performance.

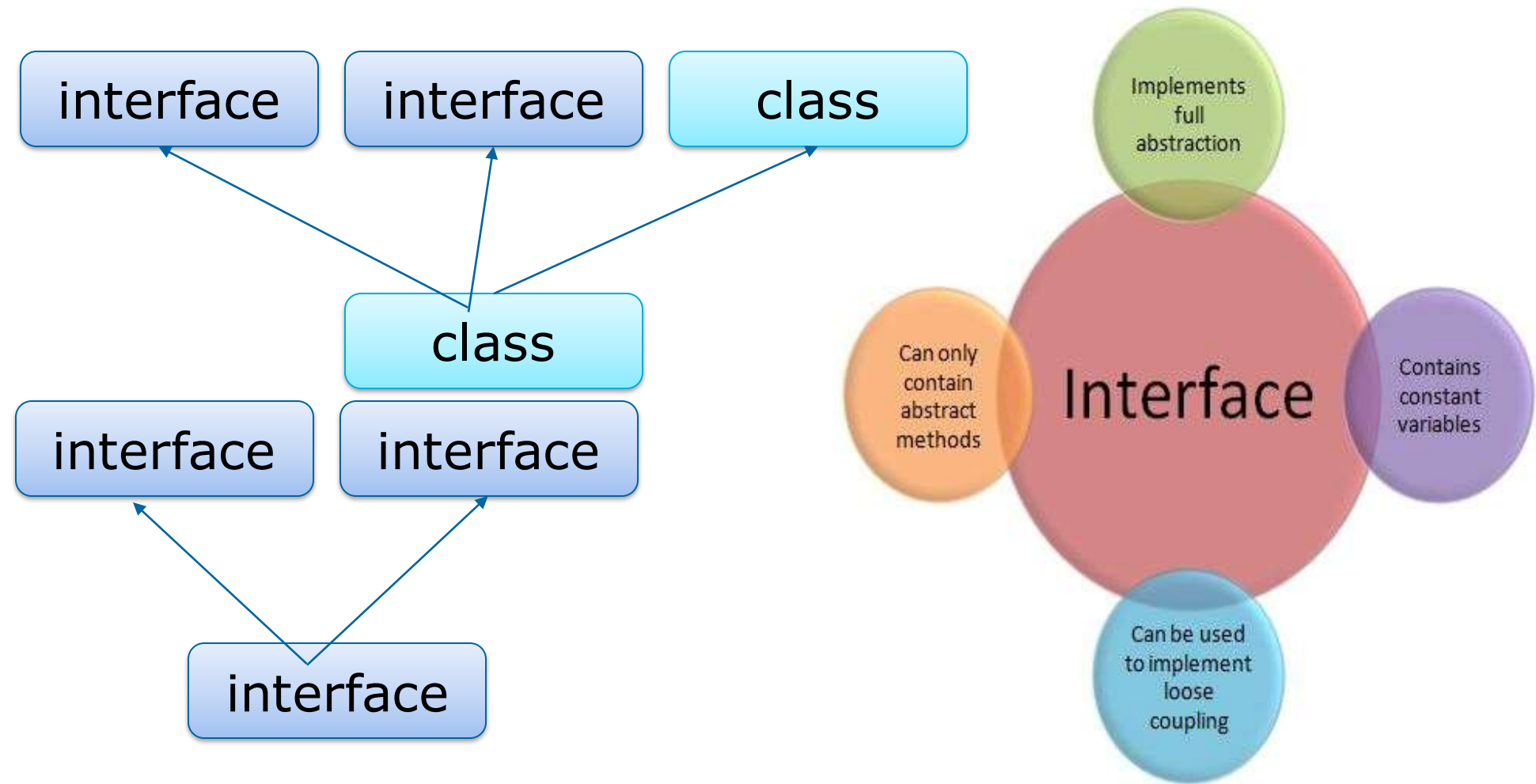
Interface

- ▶ Interfaces specify what a class must do and not how. It is the blueprint of the class.
- ▶ If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
- ▶ Why do we use interface ?
- ▶ It is used to achieve total abstraction.
- ▶ Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- ▶ It is also used to achieve loose coupling.
- ▶ Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

Abstract class vs. Interface

Interface	Abstract
Java interface are implicitly abstract and cannot have implementations	A Java abstract class can have instance methods that implements a default behavior
Variables declared in a Java interface is by default final	An abstract class may contain non-final variables.
Members of a Java interface are public by default	A Java abstract class can have the usual flavors of class members like private, protected, etc
Java interface should be implemented using keyword "implements"	A Java abstract class should be extended using keyword "extends"
An interface can extend another Java interface only	an abstract class can extend another Java class and implement multiple Java interfaces.
Interface is absolutely abstract and cannot be instantiated	A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
java interfaces are slow as it requires extra indirection	Comparatively fast

Interface



Interface Coding

```
import java.sql.Connection;
import java.sql.SQLException;

public interface CRUDMS {
    String DRIVER="MS Driver Class Name";
    String URL="MS Host name";
    String ID="MS User";
    String PASSWORD="MS Password";
    Connection getConnection()throws SQLException;
}
```

```
public interface CRUDOracle {
    String DRIVER="Oracle Driver Class Name";
    String URL="Oracle Host name";
    String ID="Oracle User";
    String PASSWORD="Oracle Password";
    Connection getConnection()throws SQLException;
}
```

```
public class StudentDAO implements CRUDOracle,CRUDMS{
    @Override
    public Connection getConnection() throws SQLException{
        return DriverManager.getConnection(CRUDOracle.URL,
            CRUDOracle.ID,CRUDOracle.PASSWORD);
    }
}
```

Class Types

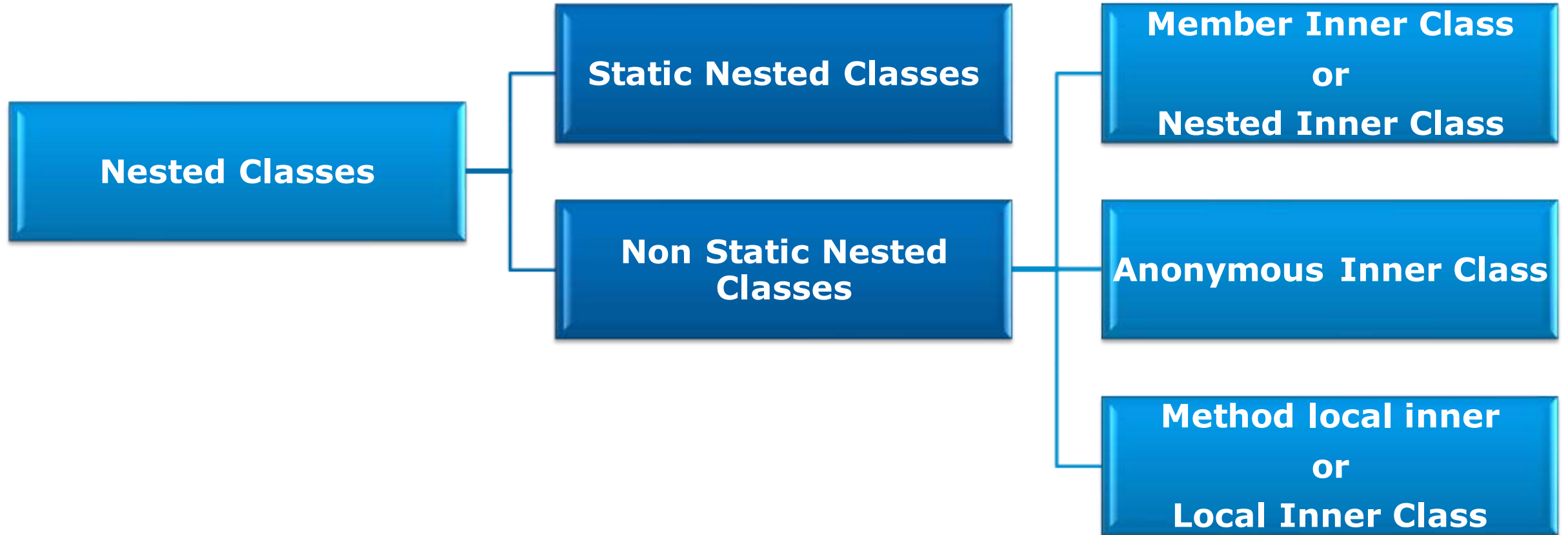
Nested Classes

- ▶ Class within another class
- ▶ The scope of a nested class is bounded by the scope of its enclosing class
- ▶ A nested class has access to the members, including private members, of the class in which it is nested. However, reverse is not true i.e. the enclosing class does not have access to the members of the nested class.
- ▶ A nested class is also a member of its enclosing class.
- ▶ As a member of its enclosing class, a nested class can be declared private, public, protected, or package private(default).
- ▶ Nested classes should be used to reflect and enforce the relationship between two classes

Purpose of Nested Classes

- ▶ They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation.
- ▶ create more readable and maintainable code
- ▶ Nested classes are of two types:
 - Static
 - Non-static aka Inner Class

Type of Nested Classes



Type of Nested Classes

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Coding Static Nested Class

```
//Creating object for Static nested class  
Outer.StaticNested sn1=new Outer.StaticNested(200);  
sn1.show(new Outer(100));
```

```
//This is class Outer  
class Outer{  
    private int outer1;  
    private static int outer2=20;  
    public Outer(int outer1) {  
        this.outer1=outer1;  
    }  
    //Here static Nested class is declared  
    static class StaticNested{  
        private int static1;  
        private static int static2=30;  
  
        public StaticNested(int static1) {  
            this.static1=static1;  
        }  
        public void show(Outer o){  
            System.out.println(static1);  
            System.out.println(static2);  
            //cannot access instance of outer class System.out.println(outer1);  
            //instance can be accessed only through reference name  
            System.out.println(o.outer1);  
            System.out.println(outer2);  
        }  
    }  
}
```

Coding Member Inner Class

```
//This is class Outer
class Outer{
    private int outer1;
    private static int outer2=20;

    public Outer(int outer1) {
        this.outer1=outer1;
    }
}
```

//Here non-static Nested/Member inner class is declared

```
class MemberNested{
    private int member1;
    //does allow static members
    //private static int member2=30;

    public MemberNested(int member1) {
        this.member1=member1;
    }

    public void show(){
        System.out.println(member1);
        System.out.println(outer1);
        System.out.println(outer2);
    }
}
```

```
//creating object for Member Nested class
Outer o1=new Outer(100);
Outer o2=new Outer(200);
Outer.MemberNested mn1=o1.new MemberNested(300);
Outer.MemberNested mn2=o2.new MemberNested(400);
mn1.show();
mn2.show();
```

Coding Local Inner Class

```
class Outer{
    private int outer1; private static int outer2=20;
    public Outer(int outer1) {
        this.outer1=outer1;
    }
    public void print(){
        int check1=10;
        final int check2=20;
        //Here Local Nested class or Local inner class is declared
        class LocalNested{
            private int local1;
            public LocalNested(int local1) {
                this.local1=local1;
            }
            public void show(){
                System.out.println(local1);
                System.out.println(outer1);
                System.out.println(outer2);
                //local
                //System
                System.out.println();
            }
        }
        //Local class object creation
        LocalNested n1=new LocalNested(10);
        n1.show();
    }
}
```

```
//creating object for Member Nested class
Outer o1=new Outer(100);
Outer o2=new Outer(200);
Outer.MemberNested mn1=o1.new MemberNested(300);
Outer.MemberNested mn2=o2.new MemberNested(400);
o1.print();
o2.print();
```

Coding Anonymous Inner Class

```
//Anonymous class created which extends Do class
Do do1=new Do(){
    @Override
    public void test() {
        System.out.println("test method overridden here");
    }
};
do1.test();

//Anonymous class created which implement Implement interface
Implement implement=new Implement() {

    @Override
    public void check() {
        System.out.println("check method overridden here");
    }
};
implement.check();
```

Enum

- ▶ Enum in java is a data type that contains fixed set of constants.
- ▶ Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.
- ▶ First line inside enum should be list of constants and then other things like methods, variables and constructor.
- ▶ Every enum constant represents an object of type enum.
- ▶ Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name. Since it is final, we can't create child enums.

Enum

► Enum and Inheritance :

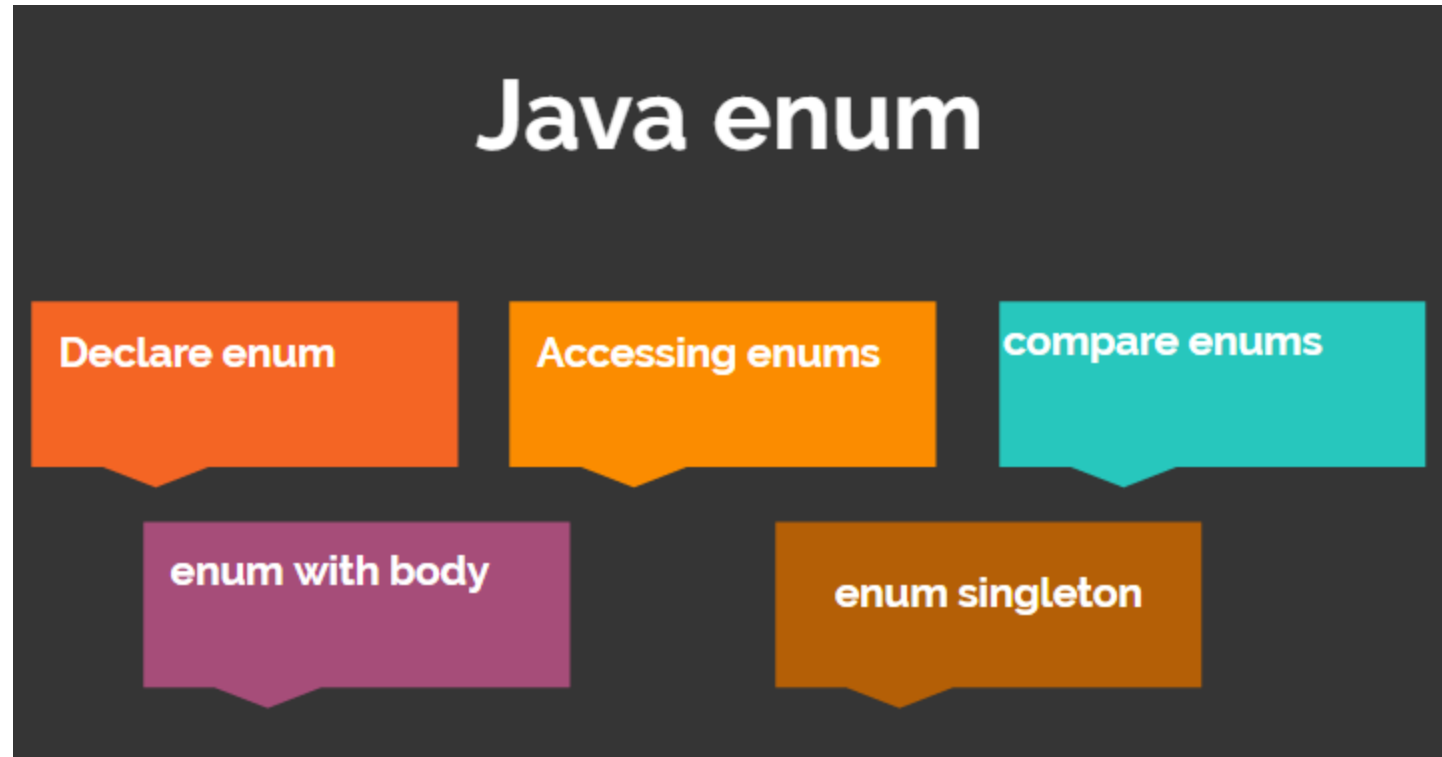
- All enums implicitly extend `java.lang.Enum` class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
- `toString()` method is overridden in `java.lang.Enum` class, which returns enum constant name.
- enum can implement many interfaces.

► `values()`, `ordinal()` and `valueOf()` methods :

- These methods are present inside `java.lang.Enum`.
- `values()` method can be used to return all values present inside enum.
- Order is important in enums. By using `ordinal()` method, each enum constant index can be found, just like array index.
- `valueOf()` method returns the enum constant of the specified string value, if exists.

Enum

- ▶ enum and constructor :
 - enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
 - We can't create enum objects explicitly and hence we can't invoke enum constructor directly.
- ▶ enum and methods :
 - enum can contain concrete methods only i.e. no any abstract method.



Coding Enum

```
enum Ticket{
    screen1("Kabali",200),screen2("baahubali",150),screen3("dangal",150);
    private String movieName;
    private int price;

    /*
     * It does not allow constructor as public
     */
    private Ticket(String movieName,int price) {
        this.price=price;
        this.movieName=movieName;
    }
    public String getMovieName() {
        return movieName;
    }
    public void setMovieName(String movieName) {
        this.movieName = movieName;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }
}
```

```
//Get the movie name of screen1.
System.out.println(Ticket.screen1.getMovieName());
//Get the ticket price of screen1.
System.out.println(Ticket.screen1.getPrice());
//Set the movie name of screen1.
Ticket.screen1.setMovieName("Robot 2");
//Set the ticket price of screen1.
Ticket.screen1.setPrice(200);

//Get the index position of screen1.
System.out.println(Ticket.screen1.ordinal());

//Get the compare index position between screen1 and screen2.
System.out.println(Ticket.screen1.compareTo(Ticket.screen2));

//return all the objects declared in that as an array type.
Ticket[] values=Ticket.screen1.values();

//return all the reference of object named as screen1
Ticket screen1=Ticket.screen1.valueOf("screen1");

//return screen1 object as string type value
String name=Ticket.screen1.name();
```

Generics

- ▶ Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- ▶ Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Advantage of Java Generics

- ▶ Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- ▶ Type casting is not required: There is no need to typecast the object.
- ▶ Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Generics



Generic Types in Java

`public class Pair< T >` | **type declaration**

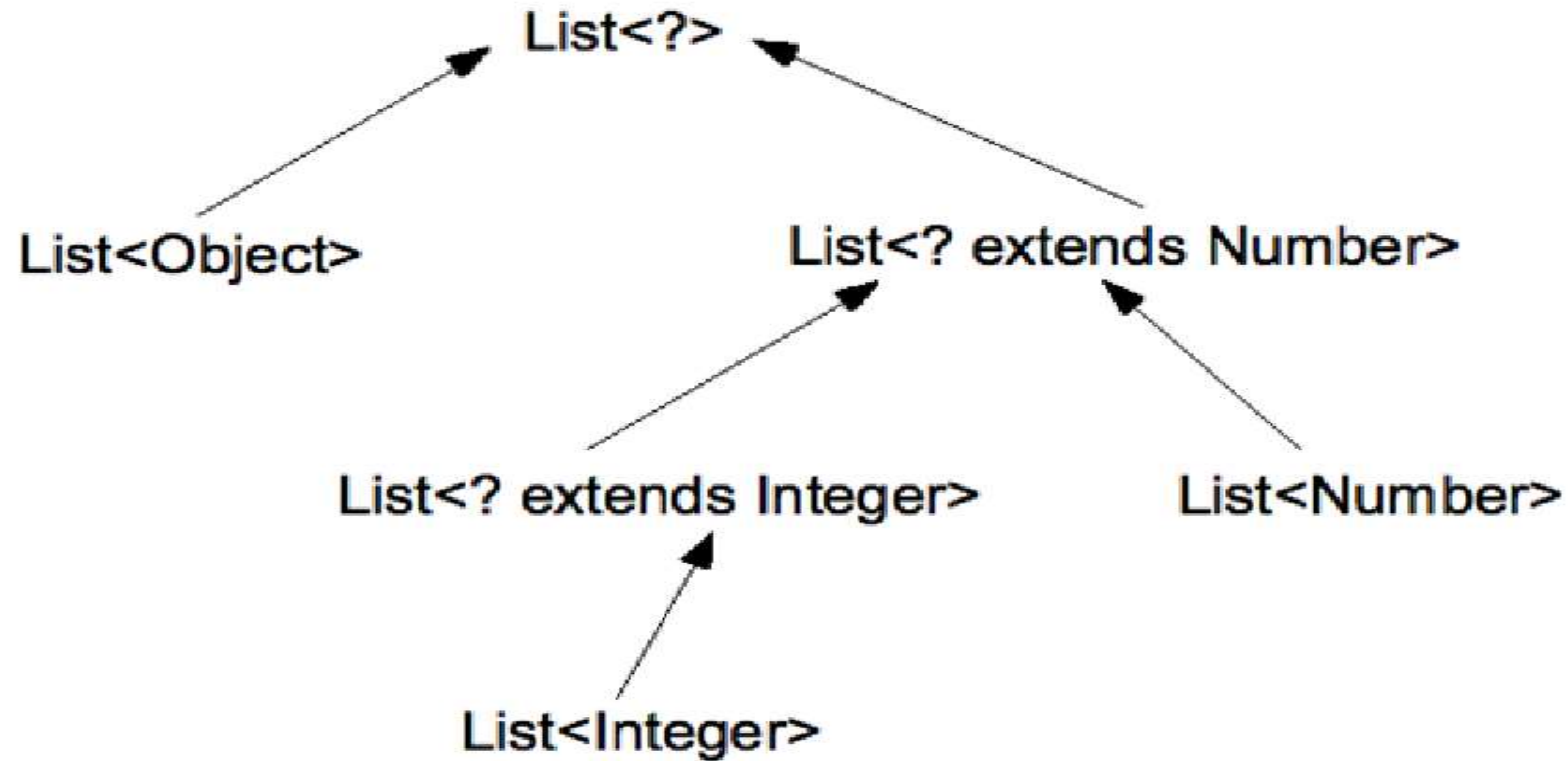
Annotations: **type variable (type parameter)** (pointing to `T`), **formal type parameter list** (pointing to `< T >`), **generic (parameterized) type** (pointing to `Pair< T >`)

`Pair< Integer > intPair;` | **type instantiation**

Annotations: **actual type to map to T** (pointing to `Integer`), **actual type parameter list** (pointing to `< Integer >`)

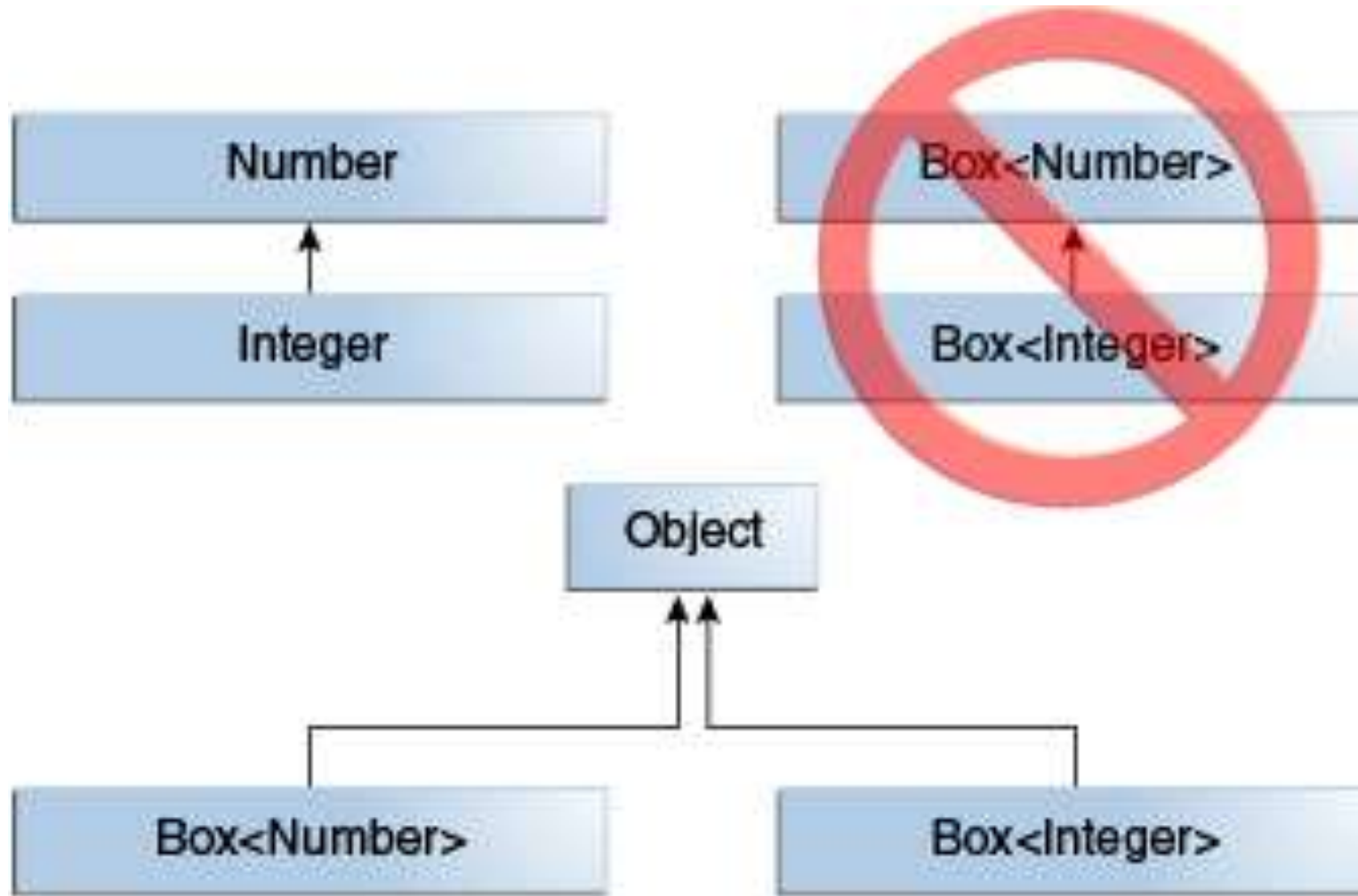
Format for a generic (parameterized) type and instantiation of a generic type

Generics



Taxonomy of the generic *List* types

Generics



Type Erasure

- ▶ Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies type erasure to:
- ▶ Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
- ▶ Insert type casts if necessary to preserve type safety.
- ▶ Generate bridge methods to preserve polymorphism in extended generic types.
- ▶ Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Java Generics

- ▶ Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.
- ▶ The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.
- ▶ Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

Advantage of Java Generics

- ▶ Type-safety : We can hold only a single type of objects in generics. It doesn't allow to store other objects.
- ▶ Type casting is not required: There is no need to typecast the object.
- ▶ **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Before Generics

```
class Box{  
    private Object length;  
  
    public Box(Object length) {  
        this.length = length;  
    }  
  
    public Object getLength() {  
        return length;  
    }  
  
    public void setLength(Object length) {  
        this.length = length;  
    }  
}
```

```
Box box1=new Box(10,20.20,401);  
Integer height=(Integer)box1.getHeight();  
  
Box2<Integer,Integer,Double> box2_1=  
    new Box2<Integer,Integer,Double>(10,10,20.20);  
//Compile time error  
//Integer height2=box2_1.getHeight();  
Double height2=box2_1.getHeight();
```

After Generics

```
class Box<T>{  
    private T length;  
  
    public Box(T length) {  
        this.length = length;  
    }  
    public T getLength() {  
        return length;  
    }  
    public void setLength(T length) {  
        this.length = length;  
    }  
}
```

```
Box<Long> box2_2=  
    new Box<Long>(101);
```

```
Box<Long> box2_3=new Box<Long>(101);
```

Multiple Generic types and Type Bounding

```
//Here V is a Bounded Type Parameters
class Box3<T,S,V extends Number>{
    private T length;
    private S width;
    private V height;

    public Box3(T length, S width, V height) {
        this.length = length;
        this.width = width;
        this.height = height;
    }
}
```

```
//Here height cannot be a String type
//Box3<Long,Double,String> box3_1=
//    new Box3<Long,Double,String>(101,10.343,"20.20");

Box3<Long,Double,Double> box3_1=
    new Box3<Long,Double,Double>(101,10.343,20.20);
Long length=box3_1.getLength();
```


@annotation



Annotations

- ▶ *Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.
- ▶ Annotations are used to provide supplement information about a program.
- ▶ Annotations start with '@'.
- ▶ Annotations do not change action of a compiled program.
- ▶ Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- ▶ Annotations are not pure comments as they can change the way a program is treated by compiler.

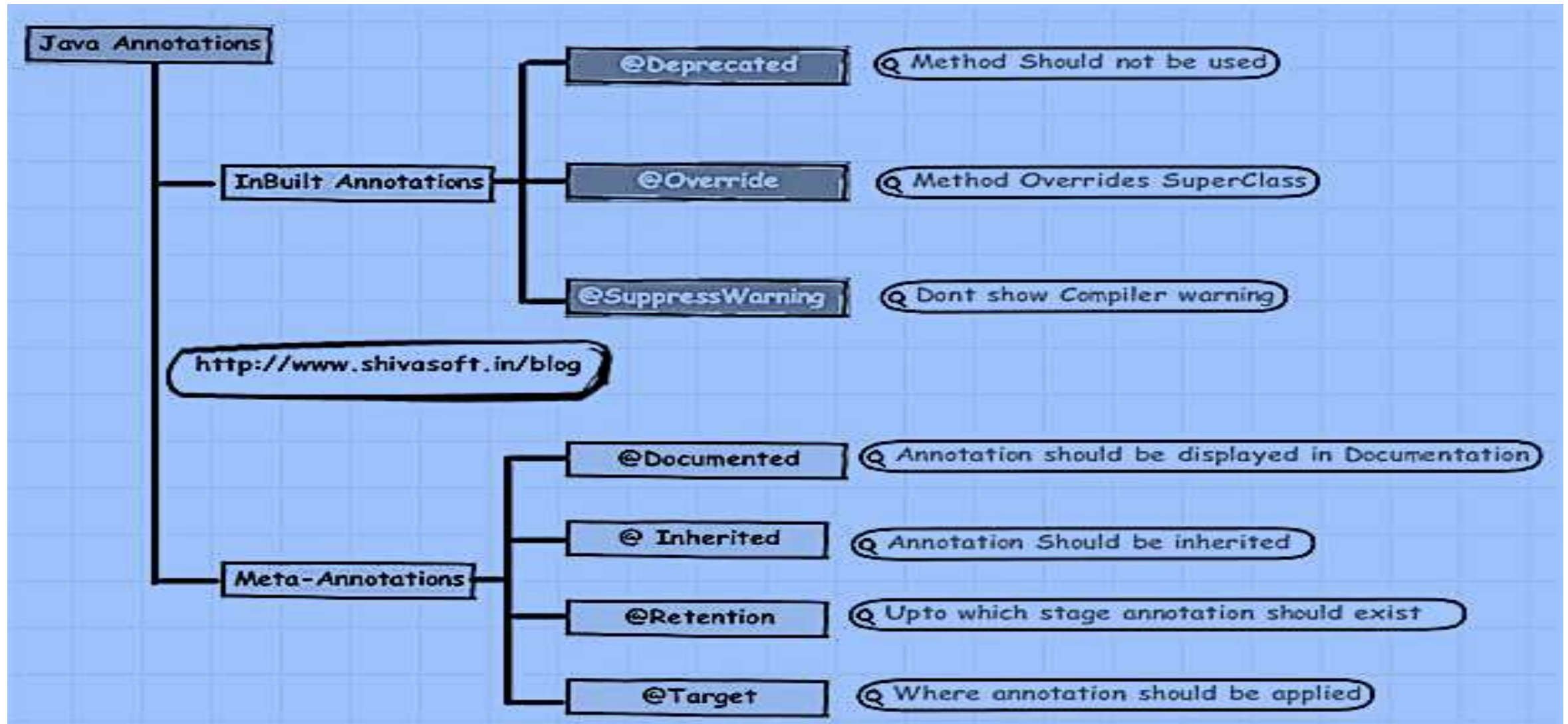
Annotations

- ▶ Built-In Java Annotations used in java code
 - @Override
 - @SuppressWarnings
 - @Deprecated
- ▶ Built-In Java Annotations used in other annotations
 - @Target
 - @Retention
 - @Inherited
 - @Documente

Java Annotations in 5 Steps

1. Annotations are metadata which can be applied on either annotations OR other java element in java source code.
2. Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program.
3. Annotations can be read from source files, class files, or reflectively at run time.
4. There are 10 in-built annotations as of today. 5 of them are meant to be applied on custom annotations and other 5 are meant to be applied on java source code elements.
5. Because annotation types are compiled and stored in byte code files just like classes, the annotations returned by these methods can be queried just like any regular Java object.

Annotations



Annotations used in JUNIT

JUNIT ANNOTATIONS

@Test

@After

@RunWith

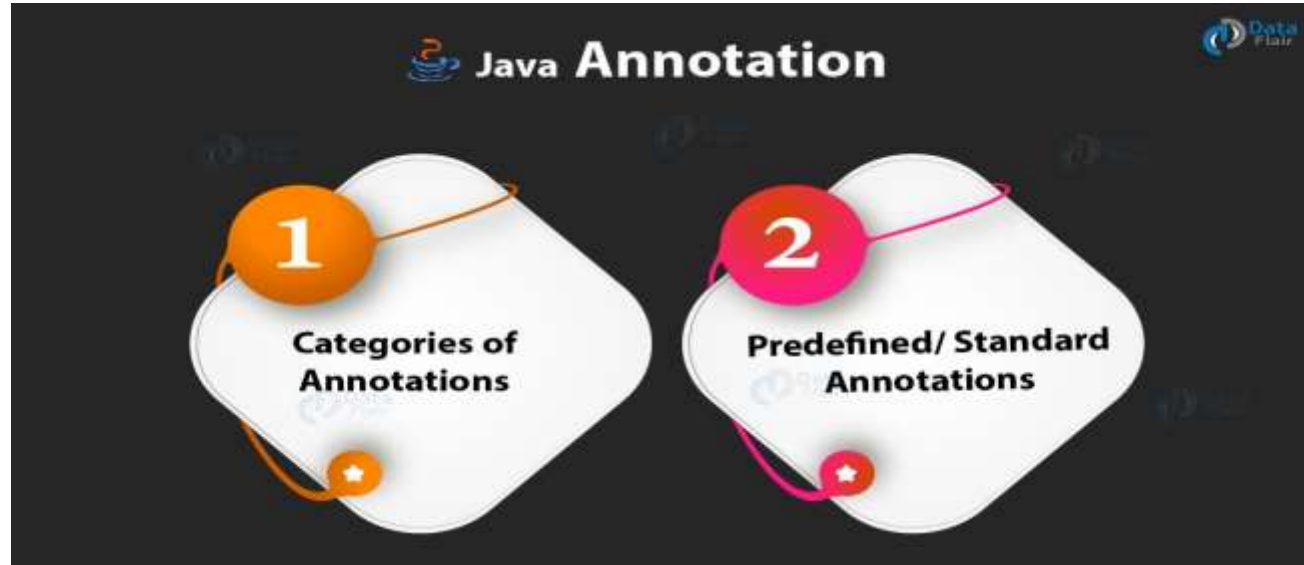
@Parameters

@Before

@Afterclass

@BeforeClass

Annotations used in JUNIT



Annotations

```
class Test1{
    public void checkIt(){
        System.out.println("checkIt method from Test1");
    }
}
class Test2 extends Test1{
    //Override annotation assures that the subclass method is overriding the parent class method.
    //If it is not so, compile time error occurs.
    @Override
    public void checkIt(){
        System.out.println("checkIt method from Test2");
    }
    //SuppressWarnings annotation: is used to suppress warnings issued by the compiler.
    //If you remove the @SuppressWarnings("unchecked") annotation, it will show warning at compile time
    //because we are using non-generic collection.
    @SuppressWarnings("unchecked")
    public static void doIt(String args[]){
        ArrayList list=new ArrayList();
        list.add("sonoo");    list.add("vimal");| list.add("ratan");
        for(Object obj:list)
            System.out.println(obj);
    }
    //Deprecated annotation marks that this method is deprecated so compiler prints warning.
    //It informs user that it may be removed in the future versions. So, it is better not to use such methods.
    //Note: Test.java uses or overrides a deprecated API.
    //Note: Recompile with -Xlint:deprecation for details.
    @Deprecated
    public void testIt(){
        System.out.println("checkIt method from Test2");
    }
}
```


Annotations

```
//This annotation indicates that new annotation should be included into
//java documents generated by java document generator tools.
@Documented
@Inherited
//@Retention annotation is used to specify to what level annotation will be available.
@Retention(RetentionPolicy.RUNTIME)
//@Target tag is used to specify at which type, the annotation is used.
@Target(ElementType.METHOD)
@interface MyAnnotation1{}
```

```
@MyAnnotation1
public static void main(String[] args) {
    Test2 test2=new Test2();
    test2.testIt();
}
```

Java Varargs

Java Varargs

A Varargs parameter means 0 or more arguments can be passed to a function.

Varargs is represented by an ellipsis (3 dots, ...)

Varargs is useful when we are not sure of number of arguments to be passed to a method.

Java Varargs

Arbitrary Number of Arguments

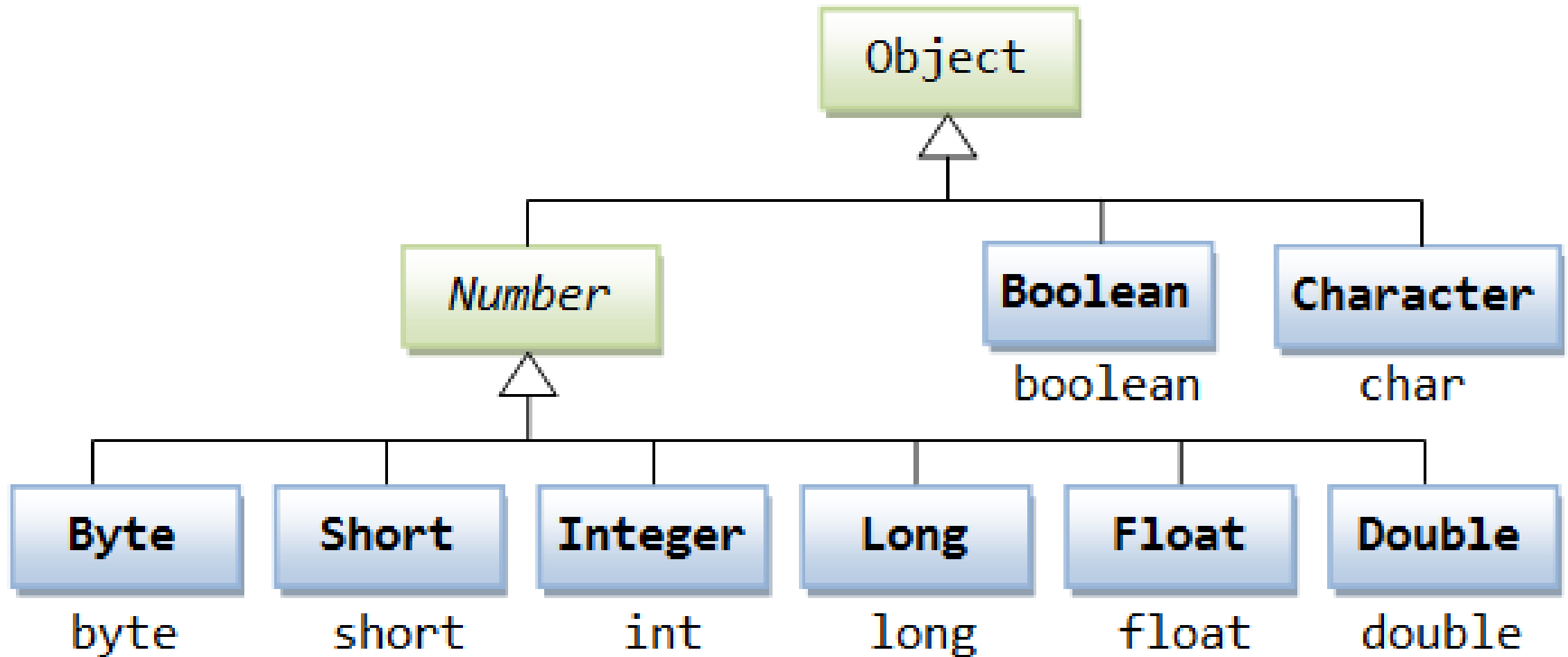
- ✓ We can use a construct called *varargs* to pass an arbitrary number of values to a method. we use varargs when we don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually.
- ✓ To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public static void calculateTotal(int... intVarArgs)
{
}
```

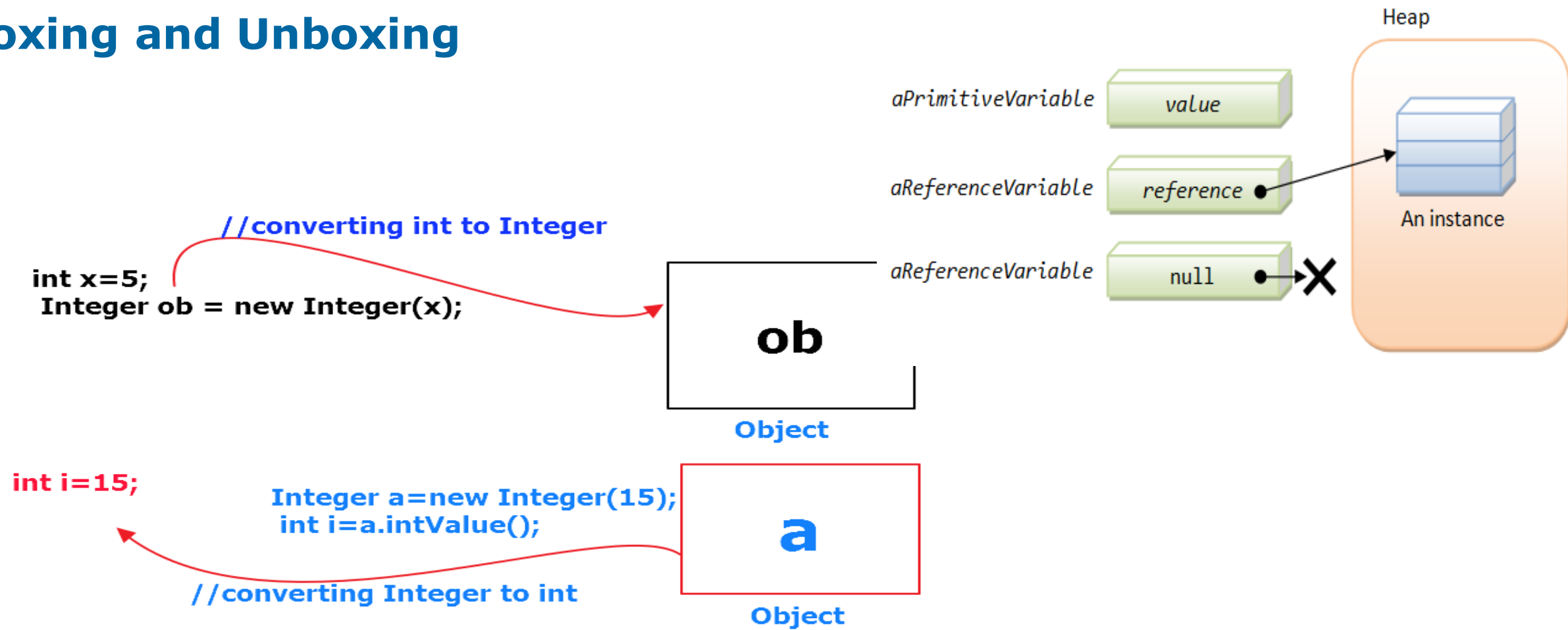
JAVA Varargs

```
public class VarArgsMain {  
    public static int sum(int[] values){  
        int total=0;  
        for(int i:values){  
            total+=i;  
        }  
        return total;  
    }  
    public static int count(int x,int... values){  
        int total=0;  
        for(int i:values){  
            total+=i;  
        }  
        return total;  
    }  
    public static void main(String[] args) {  
        int[] arr={10,20,30,40};  
        sum(arr);  
  
        count(10);//return value is 0  
        count(10,20,30);//return value is 60  
        count(10,20,30,40,50);//return value is 150  
    }  
}
```

Wrapper Classes



Boxing and Unboxing



Autoboxing and Unboxing

Boxing and Unboxing

```
//Boxing use to convert from primitive type to reference type
int check1=10;
Integer test1=Integer.valueOf(check1);

//UNBoxing use to convert reference type to primitive type
Integer test2=Integer.valueOf(20);
int check2=test2.intValue();

//Auto boxing
int check3=30;
Integer test3=check3;

//Auto Unboxing
Integer test4=Integer.valueOf(40);
int check4=test4;
```

Use of Boxing and Unboxing

```
Object object=10;  
object=20.20;  
object=true;  
  
//To use function from Wrapper classes.  
String str="100";  
int check5=Integer.parseInt(str);  
  
int check6=Integer.parseInt(str,2);  
  
int check7=Integer.parseInt(str,8);  
  
int check8=Integer.parseInt(str,16);  
  
String str2=test4.toString();  
  
String str1="100";  
double check9=Double.parseDouble(str1);
```

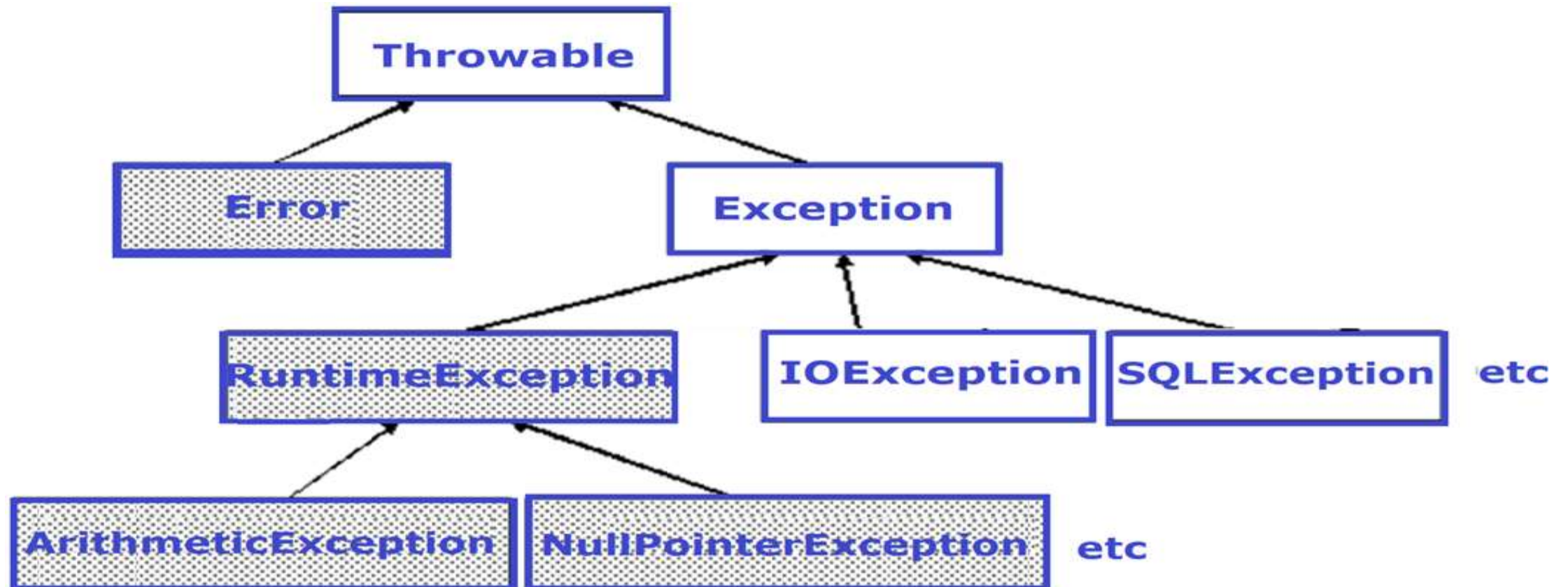

Exception Handling

Exception Handling

► What is an Exception?

- An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:
 - A user has entered invalid data.
 - A file that needs to be opened cannot be found.
 - A network connection has been lost in the middle of communications, or the JVM has run out of memory.
- Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.
- Cause normal program flow to be disrupted

Exception Handling - Class Hierarchy



Exception Handling - Class Hierarchy (Contd...)

- ▶ Throwable class
 - Root class of exception classes
 - Immediate subclasses
 - *Error*
 - *Exception*
- ▶ Exception class
 - Exceptions indicate that a problem occurred but that the problem is not a serious systemic problem.
 - Conditions that user programs can reasonably deal with
 - Usually the result of some flaws in the user program code
 - Examples
 - Division by zero error
 - Array out-of-bounds error

Exception Handling - Class Hierarchy (Contd...)

- ▶ The subclasses of `java.lang.Exception` are divided in two categories :
 - Checked Exception and
 - Unchecked Exception.
- ▶ Checked Exceptions:
 - Classes that extend `Throwable` class except `RuntimeException` and `Error` are Checked Exceptions. They are checked at compile-time by the Java compiler.
 - Compiler checks each method call and method declaration determines whether method throws checked exceptions.
 - If so, the compiler ensures checked exception caught or declared in throws clause.
 - If not caught or declared, compiler error occurs.
 - Examples:
 - `ClassNotFoundException`
 - `IOException`
 - `FileNotFoundException`

Exception Handling - Class Hierarchy (Contd...)

► Unchecked Exception:

- The classes that extend RuntimeException are known as Unchecked Exceptions.
- Unchecked exceptions are those exception objects which don't have to be explicitly caught. Whenever an unchecked exception occurs JVM will handle it automatically.
- Compiler does not check code to see if exception caught or declared rather they are checked at run-time.
- Examples:
 - NullPointerException
 - IndexOutOfBoundsException
 - ArrayIndexOutOfBoundsException

Exception Handling - Class Hierarchy (Contd...)

► Error class

- Used by the Java run-time system to handle errors
- occurring in the run-time environment
- Generally beyond the control of user programs
- This type of abnormal conditions are errors which can come from external sources or agent.
- The instances which are type of `java.lang.Error` cannot be recovered in java program. They have to be corrected by external source.
- Examples
 - `OutOfMemoryError`
 - `AssertionError`
 - `VirtualMachineError`

Exception Handling

- ▶ Exception Handling in java is done using try, catch, throw, throws, finally clause

```
try {  
    // statement that could throw an exception  
}catch (<exception type> e) {  
    // statements that handle the exception  
}catch (<exception type> e) { //e higher in hierarchy  
    // statements that handle the exception  
}finally {  
    // release resources  
}  
//other statements
```

- ▶ At most one catch block executes
- ▶ **finally** block always executes once, whether there's an error or not

Exception Handling

- ▶ What is the problem with the following code snippet?

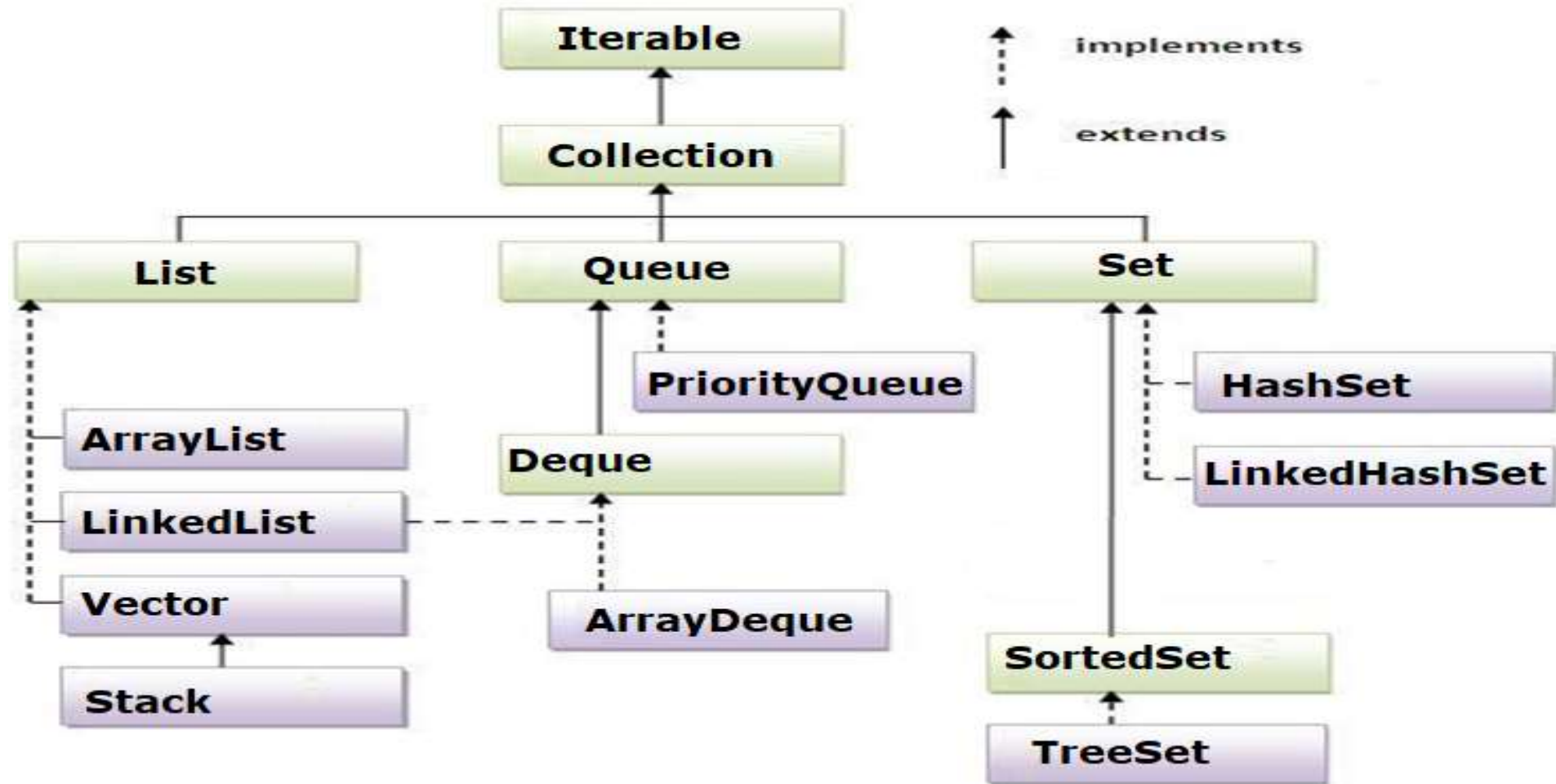
```
class sampleDemo {  
    public static void main(String a[]) {  
        int d = 0;  
        int a = 10 /d;  
        System.out.println("Rest of the code.....");  
    }  
}
```

Exception Handling

► Example: With Exception Handling

```
class exceptionDemo {  
    public static void main(String a[]) {  
        try {  
            int d = 0;  
            int a = 10 /d;  
        } catch(ArithmeticException ae) {  
            System.out.println(ae);  
        }  
        System.out.println("Rest of the code.....");  
    }  
}
```

Exception Handling



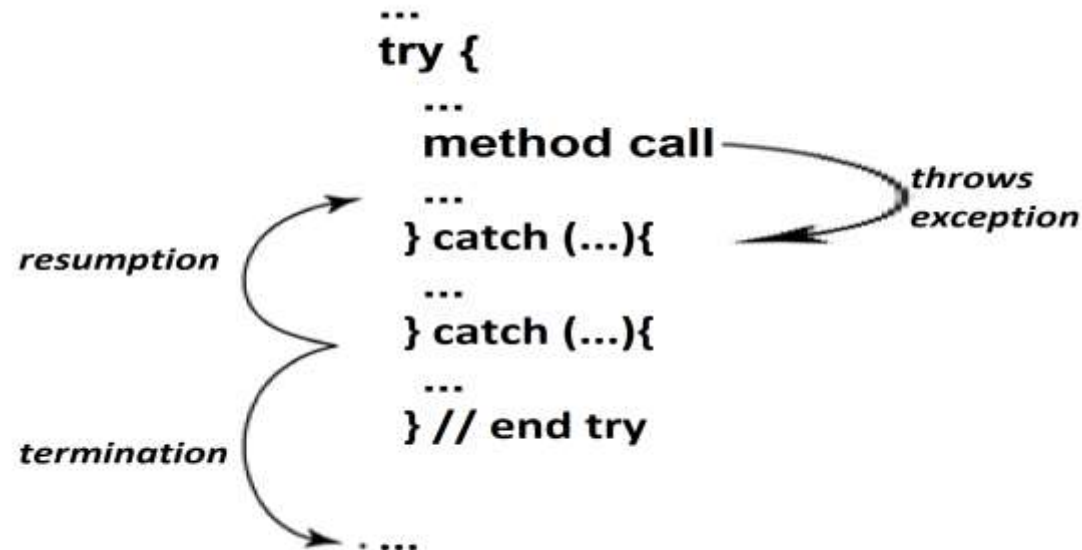
Exception Handling

- ▶ Why Exception Handling ?
 - Improves readability and maintainability
 - Error handling code is separated from the normal program
 - Adding more handlers requires more catch clauses without affecting the original program flow
 - Easy to say where the exception will be handled
 - Exceptions propagate the call stack at runtime
 - Caller can have its own error handling routines for some abnormal condition

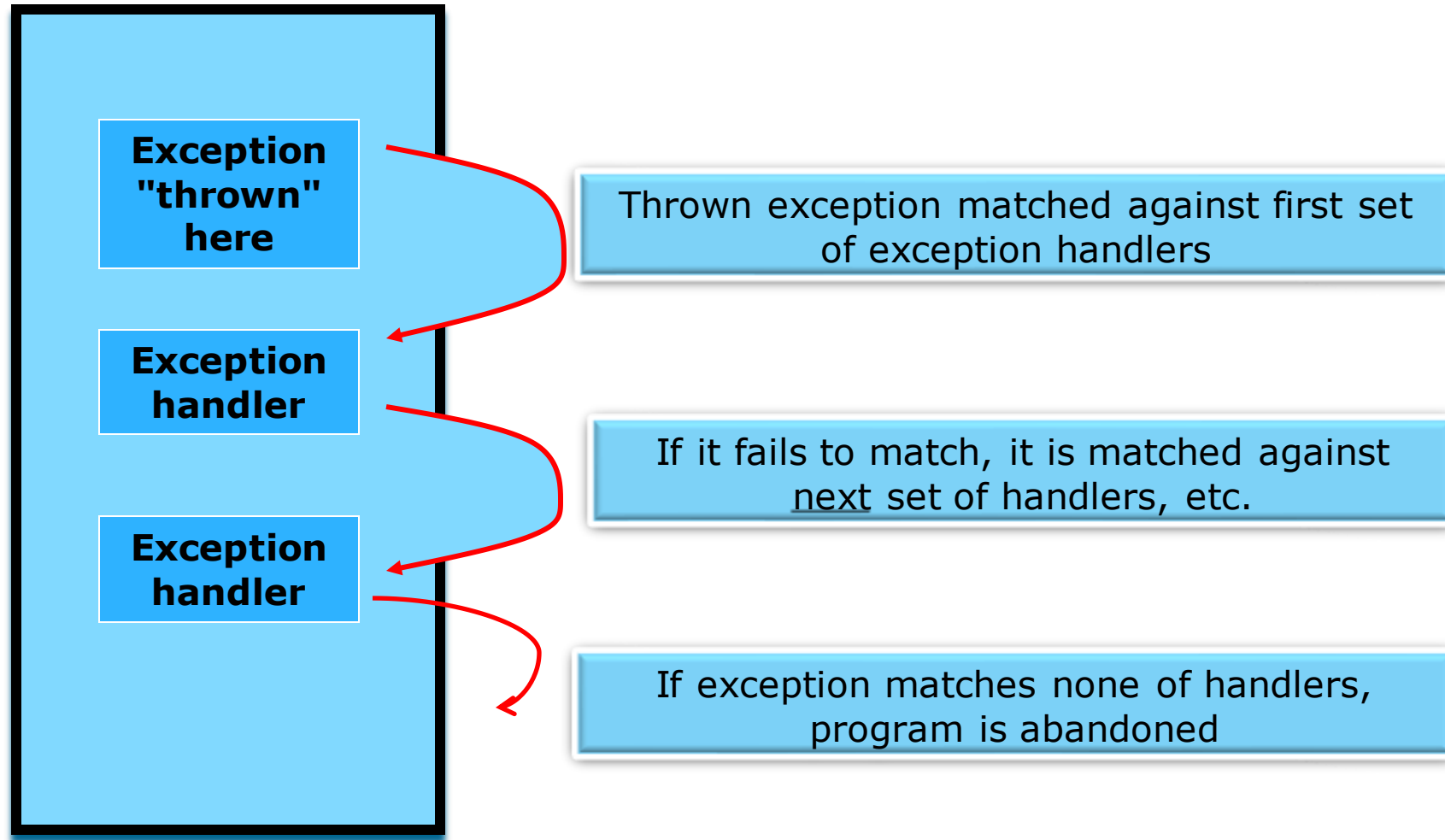
Exception Handling

► Multiple Catch

- It is possible that a statement might throw more than one kind of exception
 - We can list a sequence of catch blocks, one for each possible exception
 - There is an object hierarchy for exceptions - since the first one that matches is used and the others skipped, put a derived class first and its base class later or else it will result in a compiler error.



Exception Handling



Exception Handling

- ▶ Although try and catch provide a terrific mechanism for trapping and handling exceptions, the problem is

“how to clean up after an exception occurs”

- ▶ Because execution transfers out of the try block as soon as an exception is thrown, the cleanup code cannot be put at the bottom of the try block and expect it to be executed if an exception occurs.



Exception Handling

► Finally Clause

- A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not.
 - Even if there is a return statement in the try block, the finally block executes right after the return statement is encountered, and before the return executes
 - Will not execute if the application exits early from a try block via method `System.exit`

Exception Handling

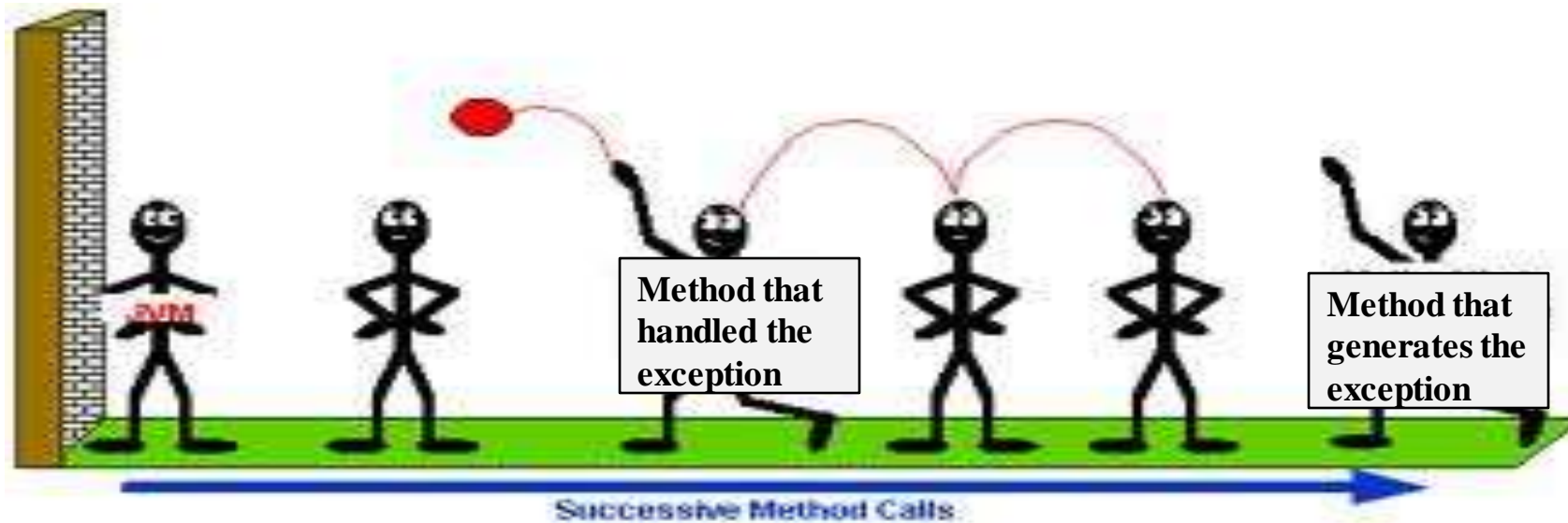
► Finally Clause(Contd..)

– Example:

```
class demo {  
    public static void main(String a[]) {  
        try {  
            int d = 0;  
            int a = 42 /d;  
        } catch(ArithmeticException ae) {  
            System.out.println(ae);  
        } finally {  
            System.out.println("Always Executed");  
        }  
    }  
}
```

Exception Handling – Propagating an Exception

- ▶ Before you can catch an exception, some code somewhere must throw one.
- ▶ Any code can throw an exception: our code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment.
- ▶ Regardless of what throws the exception, it's always thrown with the **throw** statement.



Exception Handling – Propagating an Exception

► Throw Clause:

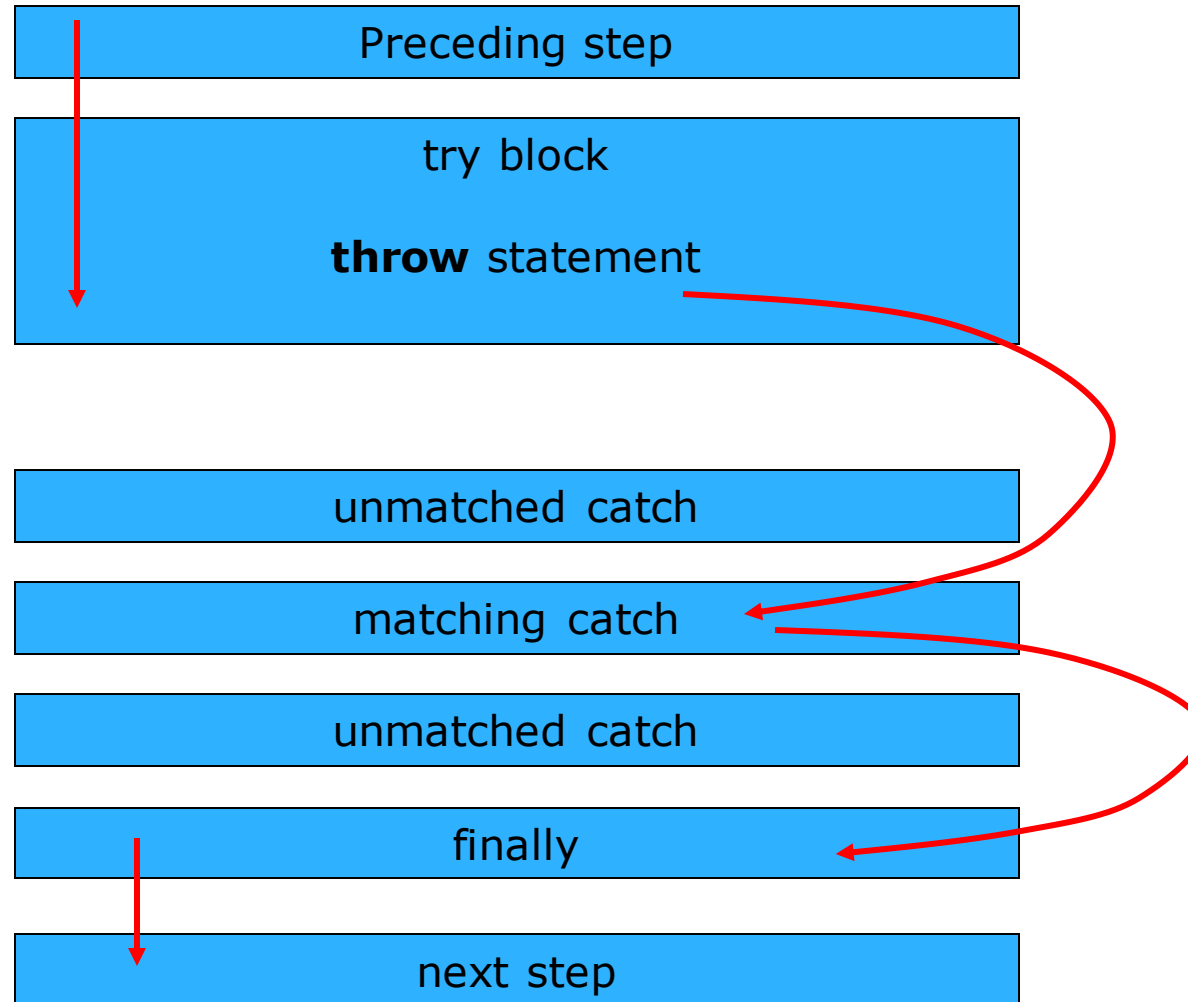
- You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.
- The throw statement requires a single argument: a throwable object.
- Throwable objects are instances of any subclass of the Throwable class.

Exception Handling – Propagating an Exception

► Example:

```
public int divideIntegers(int number, int divisor) {  
    try{  
        if(divisor==0)  
            throw new ArithmeticException("divisor cannot be zero");  
        else  
            return number/divisor;  
    }catch(ArithmeticException ae){ }
```

Exception Handling – Propagating an Exception



Exception Handling - Question

- ▶ Consider a scenario where you are making some general purpose class. This class may contain several methods that may be error prone. So in this case it's better to declare them instead of handling.

“How will you declare an exception is thrown”

Exception Handling

- ▶ If method is capable of throwing an exception or If a method does not handle a checked exception, then calling programs needs to be informed about it. This is done by **throws**
- ▶ **Throws Clause:**
 - The throws keyword appears at the end of a method's signature followed by a comma separated list of Exceptions.
 - Throws clause is not commonly used for Exceptions of type Error, RuntimeException, or its subclasses

Exception Handling

► Throws Clause(Contd..)

– Example

```
public int divideIntegers(int number, int divisor) throws ArithmeticException{  
    if(divisor==0)  
        throw new ArithmeticException("divisor cannot be zero");  
    else  
        return number/divisor;  
}
```

- Now, wherever this method is used either handle the thrown exception or re throw them.
- Re-throwing an exception is known as **Chained exception**.

Exception Handling – User Defined Exception

- ▶ You can create your own exceptions in Java. For creating user-defined exceptions we must ensure that:
 - All exceptions must be a child of Throwable.
 - If you want to write a checked exception we need to extend the Exception class.
 - If you want to write a runtime exception, you need to extend the RuntimeException class.

Exception Handling – User Defined Exception

► Example:

```
class AgeException extends Exception {  
    AgeException(String message) {  
        super(message);  
    }  
}
```

► A method that makes use of the above exception class:

```
public int setAge(int age) throws AgeException{  
    if (age<18)  
        throw new AgeException("Age should be more than 18");  
    else  
        this.age=age;
```

Assertion

Assertions

- ▶ An assertion is a Java statement in the program specifying whether the current state of the computation is as expected
- ▶ Assertions can be used to assure program correctness and avoid logic errors.
- ▶ Must form a **boolean** expression that
 - When evaluated to **true** nothing happens
 - The program state is ok
 - When evaluated to **false** throws **AssertionError**
- ▶ Can be disabled during runtime without program modification or recompilation
- ▶ Two forms
 - **assert condition;**
 - **assert condition: expression;**

Assertions

► Example:

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i == 10;  
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;  
    }  
}
```

Assertions

- ▶ By default, the assertions are disabled at runtime. To enable it, use the switch `-enableassertions`, or `-ea` for short, as follows:
 - **java -ea AssertionDemo**
- ▶ Assertions can be selectively enabled or disabled at class level or package level. The disable switch is `-disableassertions` or `-da` for short.
 - Example: The following command enables assertions in package package1 and disables assertions in class Class1.

java -ea:package1 -da:Class1 AssertionDemo

Exception Handling vs Assertions

► Exception Handling Vs Assertions

- Assertion should not be used to replace exception handling. Exception handling deals with unusual circumstances during program execution. Assertions are to assure the correctness of the program.
- Exception handling addresses robustness and assertion addresses correctness.
- Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks.
- Assertions are checked at runtime and can be turned on or off at startup time.

Summary

- ▶ Exception Handling :
 - It is done using try, catch, throw, throws, finally.
 - For normal execution:
 - try block executes, then finally block executes, then other statements execute
 - When an error is caught and the catch block throws an exception or returns:
 - try block is interrupted
 - catch block executes (until throw or return statement)
 - finally block executes
 - When error is caught and catch block doesn't throw an exception or return:
 - try block is interrupted
 - catch block executes
 - finally block executes
 - other statements execute

Summary

- ▶ Exception
 - When an error occurs that is not caught:
 - try block is interrupted
 - finally block executes
- ▶ Assertions:
 - Checks for the correctness of the program
 - Can be enabled or disabled as per programmer's choice.



Questions

- ▶ What can be the possible cause of NoClassDefFoundError?

NoClassDefFoundError occurs when a class was found during compilation but could not be located in the classpath while executing the program.

Questions

- ▶ How will you re-throw an exception which has been caught in a catch block?

Throw

Questions

- ▶ Consider a scenario where you are using a third-party function that is specified to return a double value between 0 and 1, but we'd like to guarantee that is the case. How will you do it?

Using Assertions

Collections

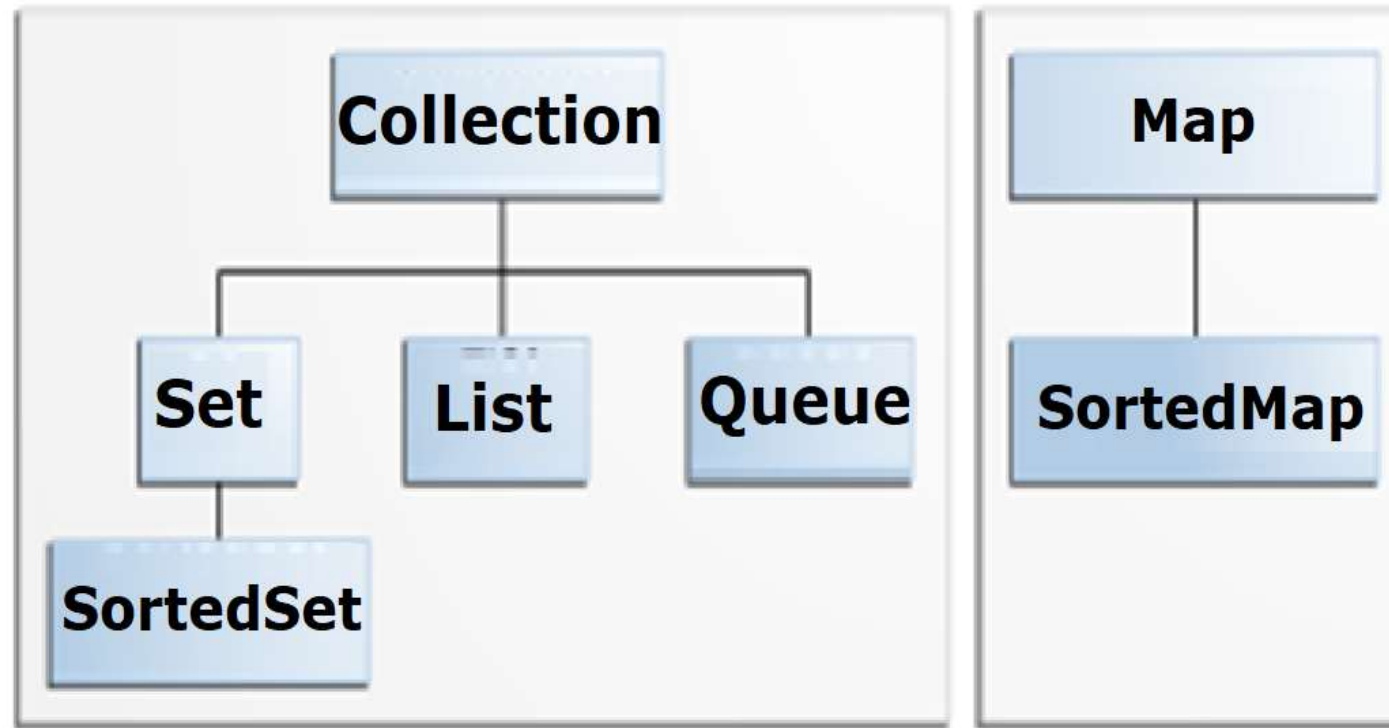
Collections

- ▶ What are Collections?
 - A Collection is a group of objects represented as a single unit.
- ▶ Collections Framework
 - Collections framework provide a set of standard utility classes to manage collections.
 - Defined within **java.util** package.
 - Collections Framework consists of three parts:
 - Core Interfaces:
 - Concrete Implementation
 - These are the concrete implementations of the collection interfaces.
 - In essence, they are reusable data structures.
 - Algorithms:
 - These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Collections

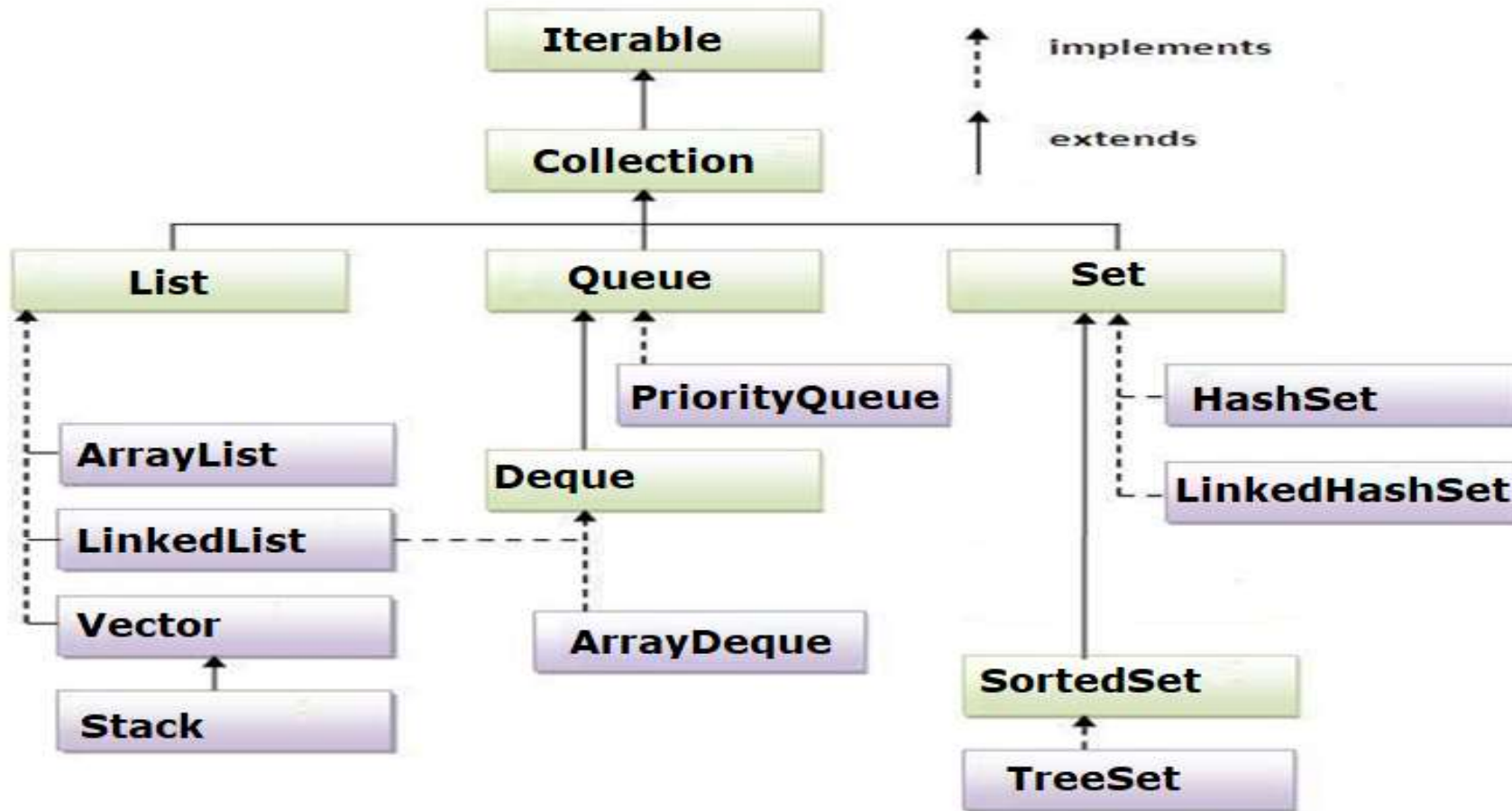
► Core Interfaces:

- These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation.
- There are two groups of interfaces Collection and Map



Collections

► Concrete Implementations



Iterable Interface:

- ▶ The Iterable interface (`java.lang.Iterable`) is one of the root interfaces of the Java collection classes.
- ▶ The Collection interface extends Iterable, so all subtypes of Collection also implement the Iterable interface.

Collection Interface:

- ▶ Java does not come with a usable implementation of the Collection interface, so you will have to use one of the listed subtypes.
- ▶ The Collection interface just defines a set of methods (behaviour) that each of these Collection subtypes share.
 - Collection extends the Iterable interface, which only requires one method, which supplies an object used to iterate through the collection
- ▶ Regardless of what Collection subtype you are using there are a few standard methods
 - To add and remove elements from a Collection.
 - `boolean add(Object element);`
 - `boolean remove(Object element);`

Collection Interface: (Contd...)

- ▶ To add and remove collection sub-types from a Collection.
 - `boolean addAll(Collection c);`
 - `boolean removeAll(Collection c);`
 - `boolean retainAll(Collection c);` - Instead of removing all the elements found in the parameter Collection, it keeps all these elements, and removes all other elements.
- ▶ To check if a Collection contains one or more certain elements.
 - `Boolean contains(Object element)` and
 - `boolean containsAll(Collection c);`
- ▶ To check the size of a collection
 - `int size();`
- ▶ To iterate all elements of a collection
 - `Iterator iterator();`

Collection Interface: (Contd...)

► Traversing a Collection:

- for-each Construct

```
for (Object o : collection)  
    System.out.println(o);
```

- Iterators

```
Iterator it = collection.iterator()
```

List Interface

- ▶ Elements are ordered and may contain duplicates
- ▶ Have indexable elements (using zero-based indexing).
- ▶ Elements are kept ordered sequence (which often depends on the order that we added them).
- ▶ There are 4 main **List implementations**:
 - **Vector**
 - a general kind of list with many useful accessing/modifying/searching methods
 - a synchronized class
 - **ArrayList**
 - also general like Vectors
 - an unsynchronized class (faster than Vectors)
 - It is re-sizable array implementation
 - **LinkedList**
 - methods for double-ended access, no direct access from the middle
 - **Stack**
 - accessible from one end only (the top).

Set Interface

- ▶ It represents set of objects, meaning each element can only exists once in a Set.
- ▶ It contains only methods inherited from Collection interface
- ▶ The **Set Implementations** are:
 - **HashSet:**
 - Stores its elements in a hash table.
 - This class permits the **null** element
 - **LinkedHashSet**
 - implements both the Hash table and linked list implementation of the **Set** interface.
 - This implementation differs from **HashSet** that it maintains a doubly-linked list.
 - The orders of its elements are based on the order in which they were inserted.
 - **TreeSet**
 - useful when you need to extract elements from a collection in a sorted manner.
 - The elements added to a TreeSet are sortable in an order.

SortedSet Interface:

- ▶ Extends the **Set** interface.
- ▶ Internally maintains its elements in sorted order.
- ▶ The order of the sorting is either the natural sorting order of the elements or the order determined by a Comparator .
- ▶ Only one implementation of the SortedSet interface – the **TreeSet** class.

Queue Interface:

- ▶ Represents an ordered list of objects just like a List.
- ▶ Ordering of elements in a FIFO (first-in-first-out) manner.
- ▶ Designed to have elements inserted at the end of the queue, and elements removed from the beginning of the queue.
- ▶ The Queue implementations are:
 - LinkedList
 - PriorityQueue
- ▶ We can peek at the element at the head of the queue without taking the element out of the queue using the `element()` method.

Map Interface:

- ▶ Not a subtype of the Collection interface
- ▶ Represents a mapping between a key and a value.
- ▶ Also referred to as *associative array* or *a dictionary*
- ▶ Has its own set of methods
 - Methods for adding and deleting
 - **put(Object key, Object value)**
 - **remove (Object key)**
 - Methods for extraction objects
 - **get (Object key)**
 - Methods to retrieve the keys, the values, and (key, value) pairs
 - **keySet() // returns a Set**
 - **values() // returns a Collection,**
 - **entrySet() // returns a set**

Map Interface: (Contd...)

► Most common **Map implementations** are:

- **HashMap**

- The implementation is based on a hash table except that it is unsynchronized and permits **null**.
- No ordering on (key, value) pairs.

- **TreeMap**

- useful when you need to traverse the keys from a collection in a sorted manner.
- (key, value) pairs are ordered on the key.

- **LinkedHashMap**

- implemented using both Hash table and linked list implementation of the **Map** interface.
- implementation differs from **HashMap** that maintains a doubly-linked list running through all of its entries in it.
- orders of its elements are based on the insertion-order.

SortedMap Interface:

- ▶ SortedMap interface extends the **Map** interface.
- ▶ Special Map interface for maintaining elements in a sorted order.
- ▶ Similar to a **SortedSet** except, the sort is done on the map keys.
- ▶ In addition to methods of the Map interface, it provides two methods shown as:
 - **firstKey()**:
 - returns the first (lowest) value currently in the map
 - **lastKey()**:
 - returns the last (highest) value currently in the map

Collection Benefits

- ▶ Reduces programming effort
- ▶ Increases program speed and quality
- ▶ Allows interoperability among unrelated APIs
- ▶ Reduces effort to learn and to use new APIs
- ▶ Reduces effort to design new APIs
- ▶ Fosters software reuse

Multithreading

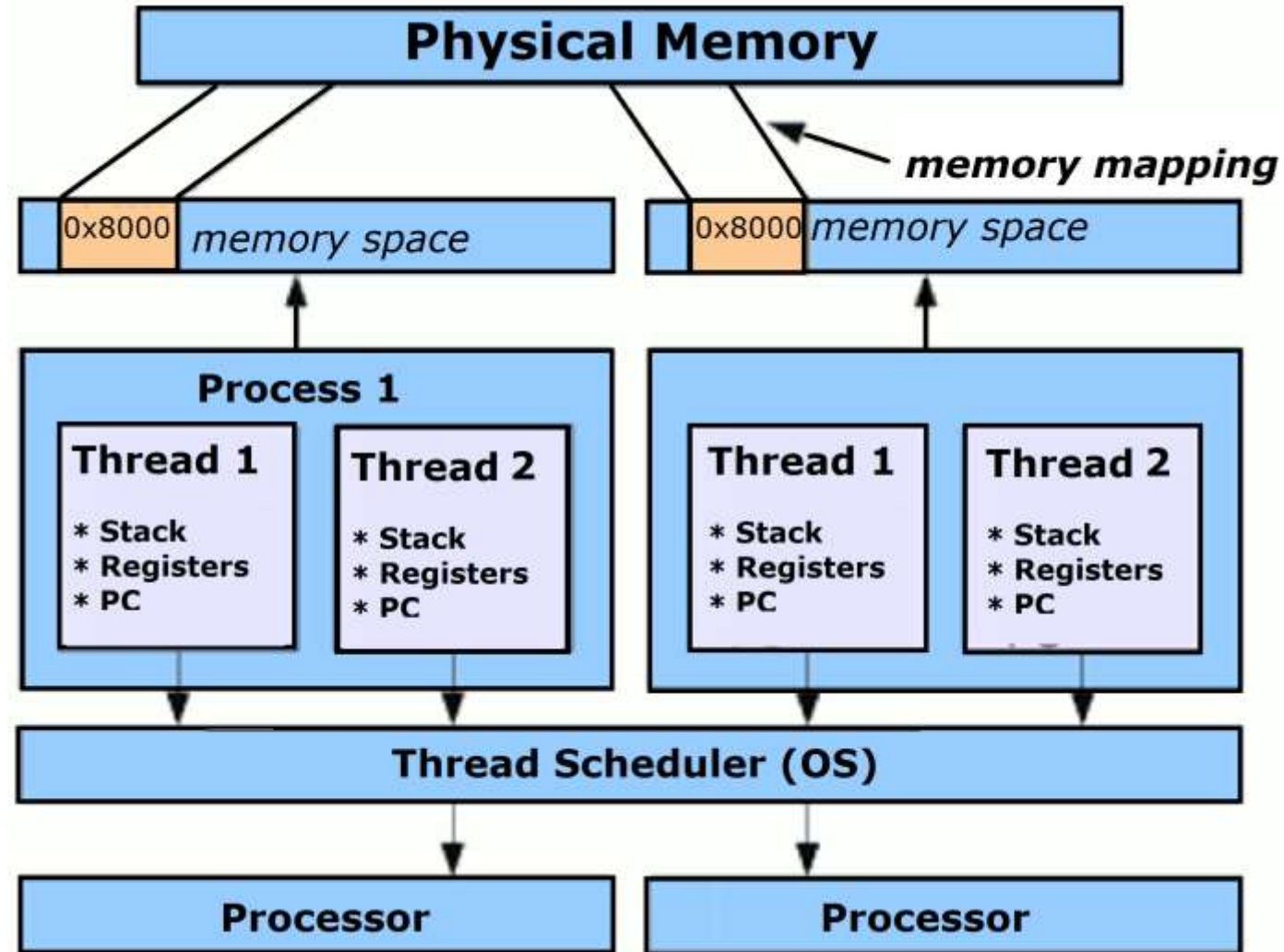
Multithreading: Thread Basics

- What are Threads?
 - A Thread defines a separate path of execution.
 - Threading is a facility to allow multiple activities to coexist within a single process.
 - We can think of them as tasks that belong to a program and that can run "simultaneously".
 - Also called *lightweight processes*.
 - Threads exist within a process — every process has at least one.
 - Used for multi-tasking.

Thread Based Multi-tasking Vs Process Based Multi-tasking

Thread Based	Process Based
Light-weight	Heavy weight
Threads exist within a process — every process has at least one.	A process has a self-contained execution environment.
Threads share the process's resources, including memory and open files.	A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Multithreading How Threads Work?



Multithreading

- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

"How"

- ▶ A **multithreaded program** has multiple threads of control
 - OS runs one thread a short period of time, then switches to another, and so on
 - To user, threads appear to run simultaneously. An illusion.
 - Nonetheless, makes it easier for user to interact with program

Multithreading

- ▶ We just discussed that every process has at least one thread.

So “ which is that thread ”

The Main Thread

- ▶ Starts running when a Java program starts up
- ▶ Help spawn other child threads.
- ▶ The last thread to finish execution.
 - When the main thread stops, program terminates
 - If the main thread finishes before a child thread has completed, then the Java run-time system may hang.

The Main Thread

“ Can the main Thread be controlled ”

The Main Thread

- Main Thread can be controlled by obtaining a reference to it.
 - Using **static Thread currentThread()**
 - a public static member of Thread class
 - Returns a reference to the thread in which it is called.
 - Once a reference to the main thread is obtained, it can be controlled just like any other thread

Creating new Threads

► Creating new Threads

– Write a class for thread

- Either a class that extends the class java.lang.Thread

`class MyThreadClass extends Thread {...}`

- Or a class that implements the interface java.lang.Runnable

`class MyRunnableClass implements Runnable {}`

– Specify what the new thread should do in the method void run() – entry point for the thread

`class MyThreadClass extends Thread`

`{`

`public void run(){`

`// codes for new thread to run`

`}`

`}`

Creating new Threads (Contd...)

- ▶ Create an object of the thread class

```
MyThreadClass t = new MyThreadClass();  
Thread t = new Thread( new MyRunnableClass() );
```

- ▶ Start a new thread by calling the start method

```
t.start()
```

- ▶ Notes:

- Don't call run directly. "run" does not create new thread
- "start" does thread setup and calls "run"

Multithreading

- Now that we have seen both the approaches of creating threads

“Which one is the best ?”



Thread vs Runnable

▶ common difference

- When you extends Thread class, after that you can't extend any other class which you required. (As you know, Java does not allow inheriting more than one class).
- When you implements Runnable, you can save a space for your class to extend any other class in future or now.

▶ significant difference

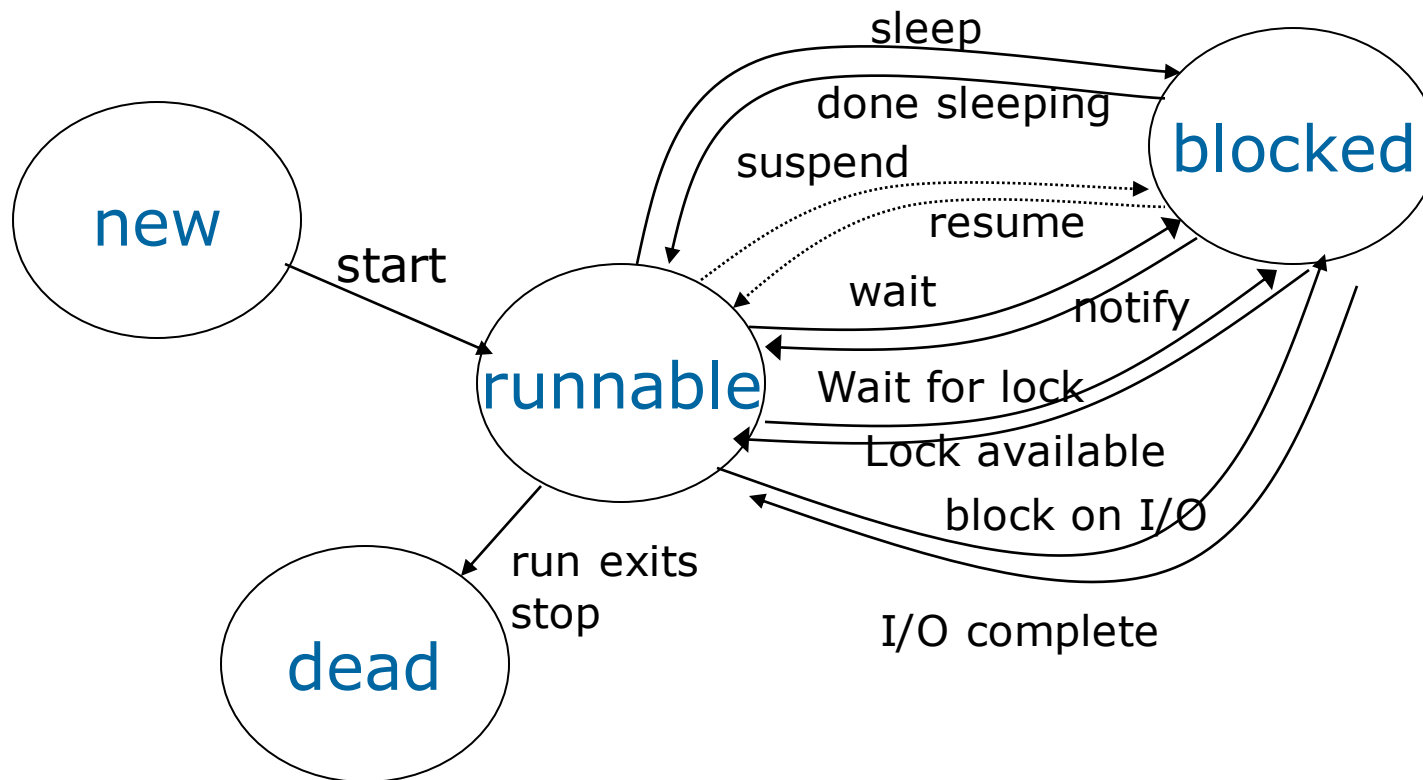
- When you extends Thread class, each of your thread creates unique object and associate with it.
- When you implements Runnable, it shares the same object to multiple threads.

Thread vs Runnable

Thread	Runnable
When you extends Thread class, after that you can't extend any other class which you required. This limits how you can reuse your application logic.	Whereas when you implements Runnable, you can save a space for your class to extend any other class in future or now.
When you extends Thread class, each of your thread creates unique object and associate with it	when you implements Runnable, it shares the same object to multiple threads.
Once a thread is started and completed its work, it's subject to garbage collection.	An instance of Runnable task can be resubmitted or retried multiple times, though usually new tasks are instantiated for each submission to ease state management.
Extending Threads do not give the program much flexibility	Implementing Runnable makes your class more flexible. You can run it in a thread, or pass it to some kind of executor service, or just pass it around as a task within a single threaded application (maybe to be run at a later time, but within the same thread).

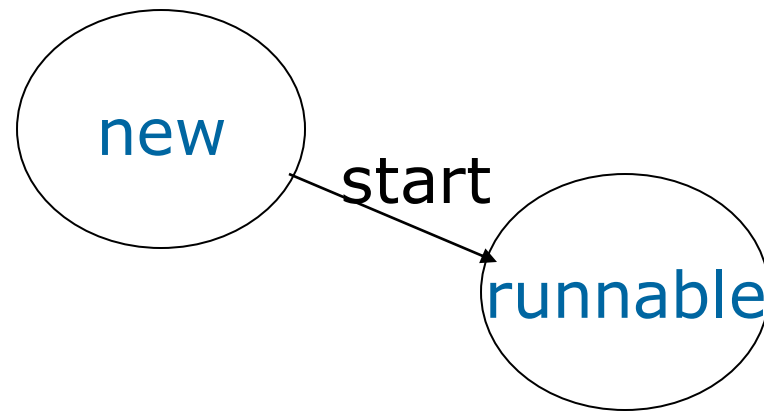
Thread States

- ▶ Thread can be in 4 states new, runnable, blocked, dead



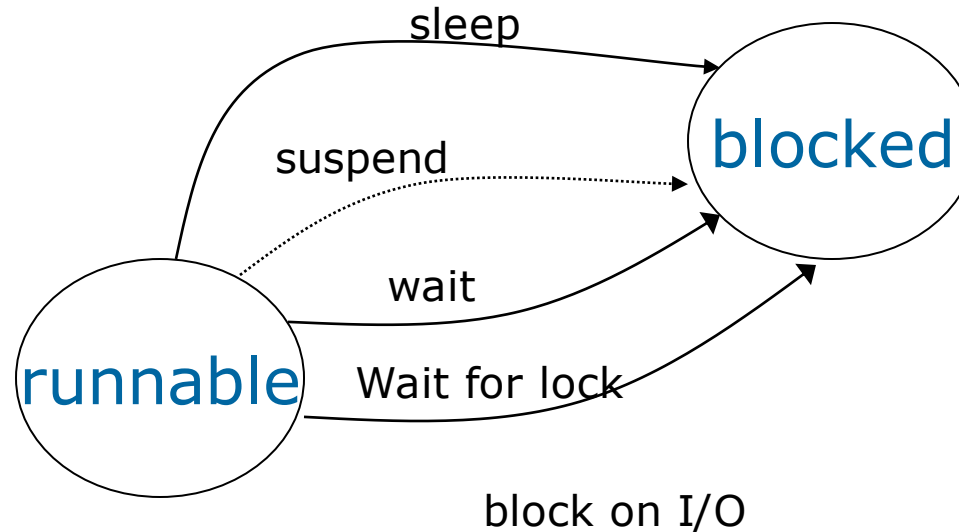
Thread: New Runnable State

- ▶ When a thread has just been created using the new operator, it is in the new state.
- ▶ Once start method is invoked (which calls the run method), the thread becomes runnable.
 - A runnable thread might not be running.
 - There can be many runnable threads. But only one of them can be running at any time point.
 - OS decides which thread to run.



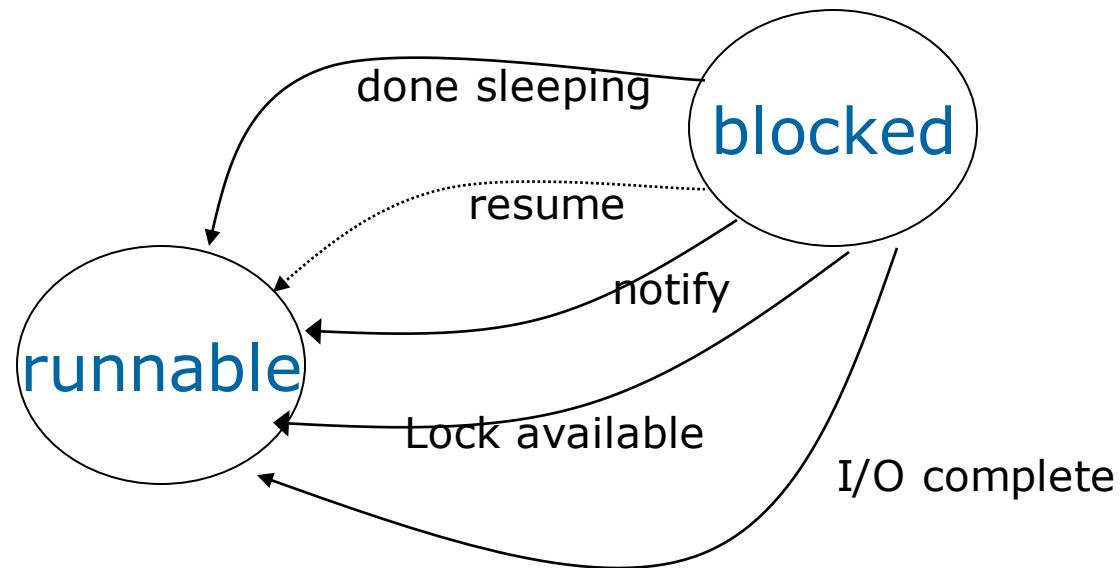
Thread: Runnable -> Blocked State

- ▶ A runnable thread enters the blocked state when
 1. The thread is currently running and method `Thread.sleep` is called
 2. `suspend` method of the thread is called. (deprecated)
 3. The thread calls the `wait` method.
 4. The thread tries to lock an object locked by another thread.
 5. The thread calls an operation that is blocked on i/o.



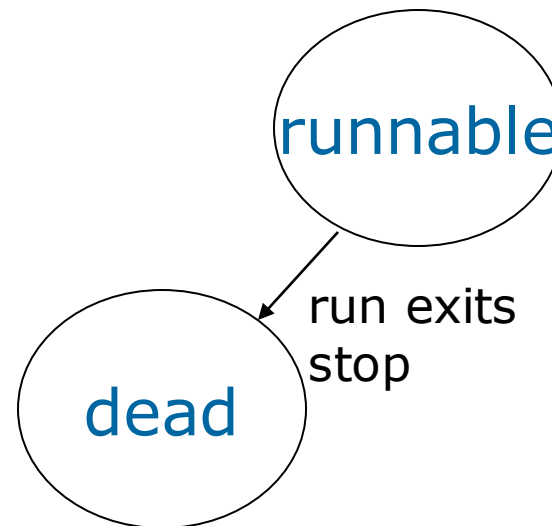
Thread: Blocked → Runnable State

- A blocked reenters runnable state when
 1. It has slept the specified amount of time.
 2. resume method of the thread is called. (deprecated)
 3. Another method calls notify or notifyAll
 4. Object lock released by other thread
 5. I/O completed.



Thread: Runnable → Dead State

- A runnable thread enters the dead state when
 1. Its run method exits. Natural death.
 2. stop method of the thread is called. (deprecated)
 3. An exception is thrown but not caught.



Thread:

“How can we know if a Thread is dead or Alive?”

Thread States:

- Method `isAlive` allows you to find out whether a thread is alive or dead.
 - This method returns `true` if the thread is `Runnable` or `Blocked`,
 - `false` if the thread is still new and not yet `Runnable` or if the thread is dead
- No way to find out whether an alive thread is `Running`, `Runnable`, or `Blocked`.

Other Thread Class Method

- ▶ `getPriority()`: Obtains thread priority
- ▶ `sleep()`: suspends the thread for some time
- ▶ `suspend()/resume()`: suspends a thread & resumes
- ▶ `interrupt()` : interrupts a thread
- ▶ `interrupted()` : true if current thread has been interrupted and false otherwise
- ▶ `isInterrupted()` : determines if a particular thread is interrupted
- ▶ `stop()` : stops a thread by throwing a `ThreadDeath` object which is a subclass of error.
- ▶ `join()`: waits for the thread to which the message is sent to die before the current thread can proceed
- ▶ `Yield()`: Causes the currently executing thread object to temporarily pause and allow other threads to execute

Mutithreading Question

- ▶ Execute the Demo UnSynchronizedThreadDemo.java and examine the result

“ How can we control access to the shared resource?”

Thread Synchronization

- ▶ Thread Synchronization:
 - When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
 - The process by which this is achieved is called *thread synchronization*.
 - Implemented using *synchronized* keyword
- ▶ The *synchronized* keyword can be applied as
 - synchronized block

```
public void method() {  
    synchronized( this ) { .... }  
}
```
 - synchronized method

```
public synchronized void method() {.... }
```

Thread Synchronization (Contd...)

- ▶ Synchronized keyword provides the following functionality:
 - Provides locking which ensures mutual exclusive access of shared resource and prevent data race.
 - It involves locking and unlocking. Before entering into synchronized method or block thread needs to acquire the lock at this point it reads data from main memory than cache and when it release the lock it flushes write operation into main memory which eliminates memory inconsistency errors.
- ▶ Hence When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

Thread Synchronization (Contd...)

- Synchronized Method Vs Synchronized Block
 - synchronized block it will lock a specific object whereas synchronized method it will lock all the objects.
- Where to use them?
 - Suppose if we want to invoke a critical method which is in a class whose access is not available then synchronized block is used. Otherwise synchronized method can be used.
 - Synchronized methods are used when we are sure all instance will work on the same set of data through the same function Synchronized block is used when we use code which we cannot modify ourselves like third party jars etc

Thread Q

- ▶ At any time, there might be many runnable threads. But only one of them is actually running.
- ▶ The thread scheduler decides which runnable thread to run.

When does the thread scheduler kick in and pick a thread to run?

How does the thread scheduler select among the runnable threads?

Thread Scheduling

- ▶ Multiple threads compete for time on the CPU. When multiple threads want to execute, it is up to the underlying operating system to determine which of those threads are placed on a CPU.
- ▶ Java programs can influence that decision in some ways, but the decision is ultimately up to the operating system.
- ▶ The threads scheduling algorithm always lets the highest priority runnable thread run. If there are several runnable high-priority threads, the CPU is allocated to all of them, one at a time, in a round-robin fashion.

Thread Scheduling Features

- ▶ The JVM schedules using a preemptive , priority based scheduling algorithm.
- ▶ All Java threads have a priority and the thread with the highest priority is scheduled to run by the JVM.
- ▶ In case two threads have the same priority a *time-slicing mechanism* is followed.
- ▶ The Java runtime does not implement (and therefore does not guarantee) time-slicing. However, some systems on which you can run Java do support time-slicing. Your Java programs should not rely time-slicing as it may produce different results on different systems.

Thread Scheduling (Contd...)

- ▶ Two different implementations:
 - Pre-emptive Scheduling
 - Non Pre-emptive Scheduling
- ▶ Non Pre-emptive” implementation :
 - A running Java thread will continue to run until
 - It calls yield method, or
 - It ceases to be runnable (dead or blocked), or
 - Another thread with higher priority moves out of blocked state
- ▶ Then the thread scheduler kicks in and picks another thread with the highest priority to run

Thread Scheduling (Contd...)

► Sleep Vs Yield

- There is a big difference
 - Calling sleep put the current running thread into the blocked state
 - Calling yield does not put the calling thread, t1, into the blocked state
 - It merely let the scheduler kick in and pick another method to run.
 - It might happen that the t1 is select to run again. This happens when t1 has a higher priority than all other runnable threads.

Thread Scheduling (Contd...)

► “Pre-emptive Scheduling” implementation :

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- In theory, higher-priority threads get more CPU time than lower-priority threads.
- The job of Java scheduler is to keep a highest priority thread running at all times
- If time slicing is available, it ensures that several equally high - priority threads execute for a quantum in a round - robin fashion

Thread Scheduling (Contd...)

▶ Thread Priorities:

- Priority of individual threads Can be increased or decreased using ***setPriority***
- Java have 10 priority levels (constants of **Thread** class)
MIN_PRIORITY = 1;
NORMAL_PRIORITY = 5;
MAX_PRIORITY = 10
- A thread inherits priority from its parent thread, the one the creates it.

▶ Note

- Some OS has fewer. E.g. Windows NT has 7.
- JVM maps Java priority levels to priority level of the underlying OS.

Thread Scheduling Q

- ▶ Sometimes one might want to pause (or delay) the execution of a thread until some condition is met. And this condition only occurs if another thread does such and such.

How do the threads communicate?

Inter Thread Communication

- ▶ It is all about making synchronized threads communicate with each other.
- ▶ Java offers three possible solutions. The following three methods are inherited from the Object class:
 - public final void wait():
 - forces a thread to wait until some other thread calls either notify() or notifyAll() methods.
 - It may throw an InterruptedException.
 - public final void notify():
 - wakes up the thread that invoked the wait() method on the same object.
 - public final void notifyAll():
 - wakes up all of the threads that called the wait() method on the same object.

Summary

- ▶ The Java language includes a powerful threading facility built into the language. You can use the threading facility to:
 - Increase the responsiveness of GUI applications
 - Take advantage of multiprocessor systems
 - Simplify program logic when there are multiple independent entities
 - Perform blocking I/O without blocking the entire program
- ▶ When you use multiple threads, we must synchronize the threads so that a lock can be acquired on the shared resource.
- ▶ The threads can interact among themselves using methods like wait, notify and notify all.



Questions

- ▶ How can you specify which thread is notified with the wait/notify protocol?

The wait/notify protocol does not offer a method of specifying which thread will be notified.

Questions

► Can we synchronize constructor of a Java Class?

As per Java Language Specification, constructors cannot be synchronized because other threads cannot see the object being created before the thread creating it has finished it.

Questions

- Can two threads call two different synchronized instance methods of an Object?

No. If a object has synchronized instance methods then the Object itself is used a lock object for controlling the synchronization. Therefore all other instance methods need to wait until previous method call is completed.

Questions

- What state does a thread enter when it terminates its processing?

When a thread terminates its processing, it enters the dead state.

Questions

- ▶ Specify at least 3 ways in which a thread can enter the waiting state
-
1. Invoking its sleep() method,
 2. By blocking on I/O
 3. By unsuccessfully attempting to acquire an object's lock
 4. By invoking an object's wait() method.
 5. It can also enter the waiting state by invoking its (deprecated) suspend() method..

10

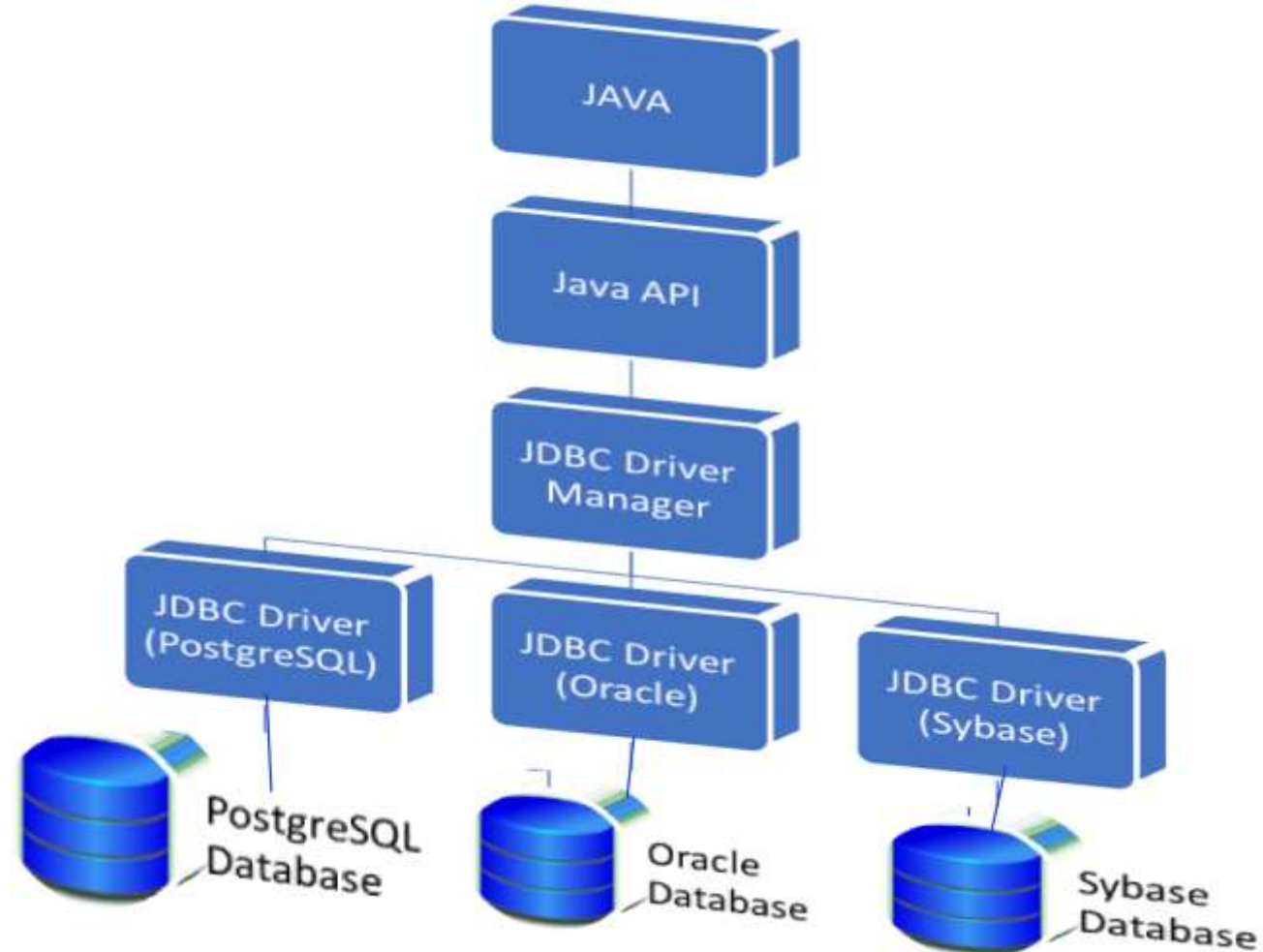
Java Database Connectivity

What is JDBC?

- ▶ Defines how a java program can communicate with a database.
- ▶ JavaSoft worked with D/B tool vendors to provide DBMS independent mechanism to write client side applications
- ▶ The result is JDBC API
 - JDBC API is designed to allow developers to create database front ends without having to continually rewrite the code
 - An API that is D/B independent and uniform across databases
 - It has through a set of interface that are implemented by the driver.
 - Driver is responsible for converting a standard JDBC call to a native call.
- ▶ JDBC API has two major packages
 - java.sql and
 - javax.sql.
 - This new JDBC API moves Java applications into the world of heavy-duty database computing.

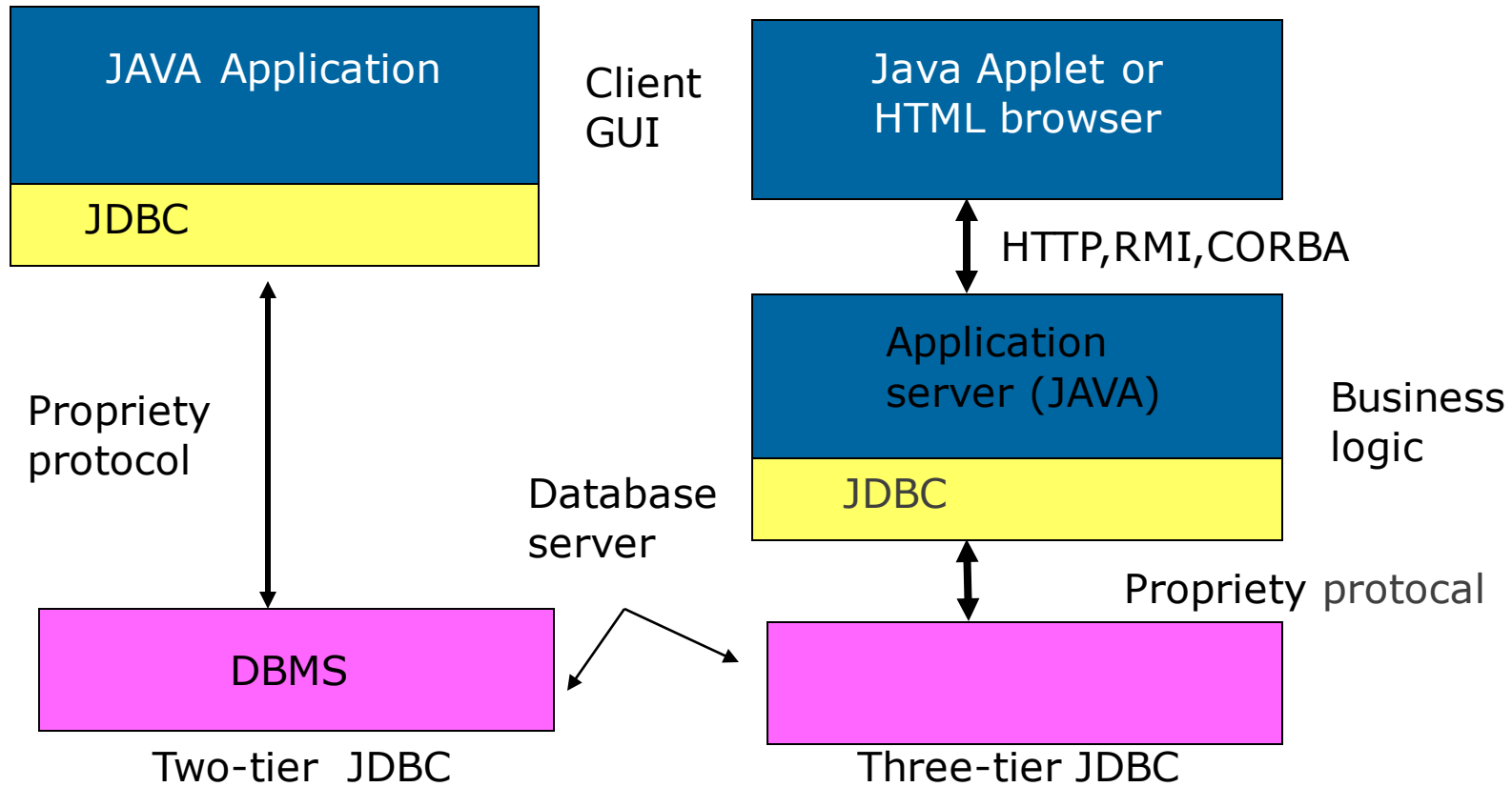
JDBC Architecture

- ▶ JDBC architecture consist of different layers and drivers which are capable of working with any database.



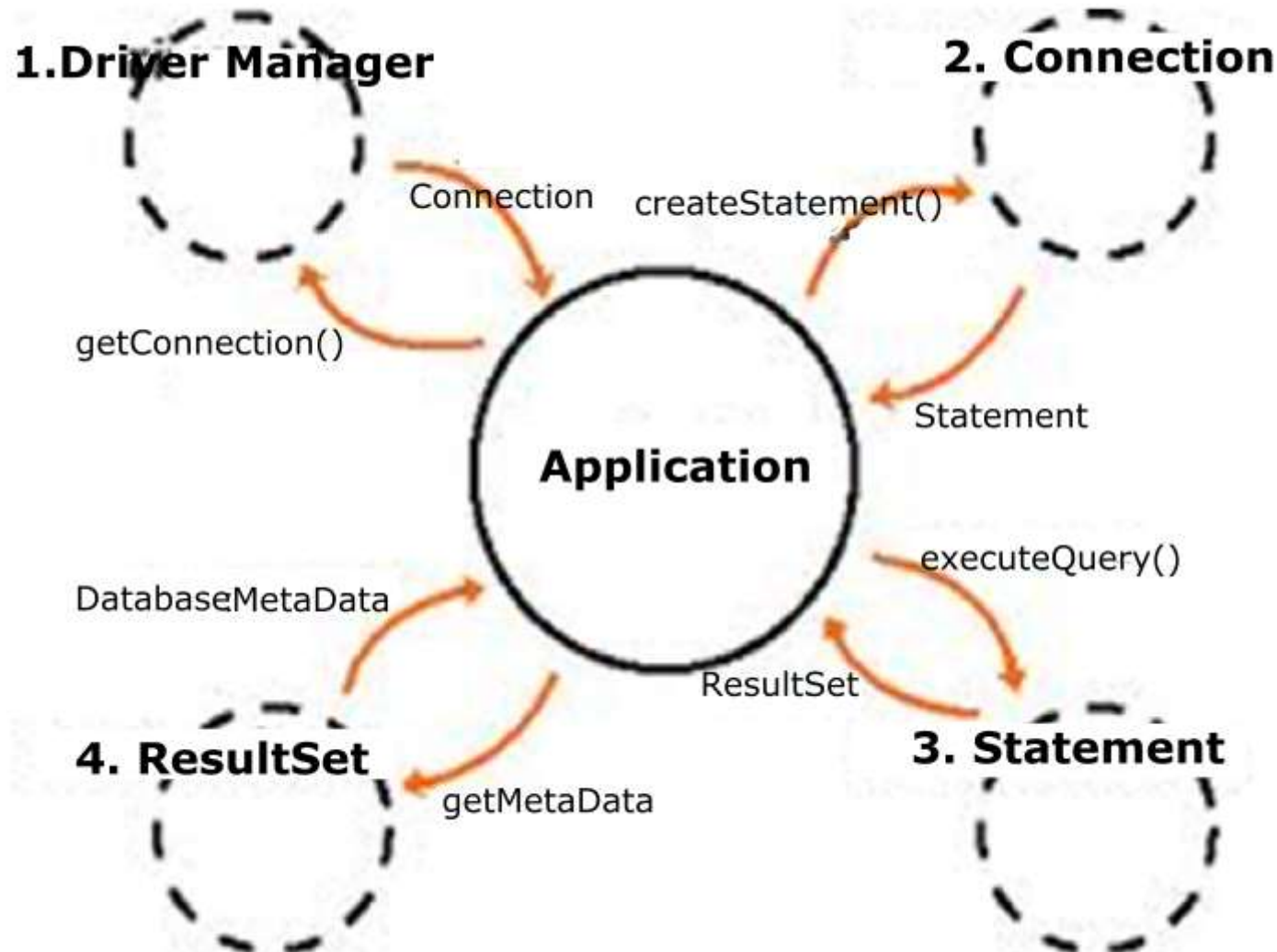
Java Database Connectivity

- ▶ JDBC API supports both two-tier and three-tier models for database access



JDBC API

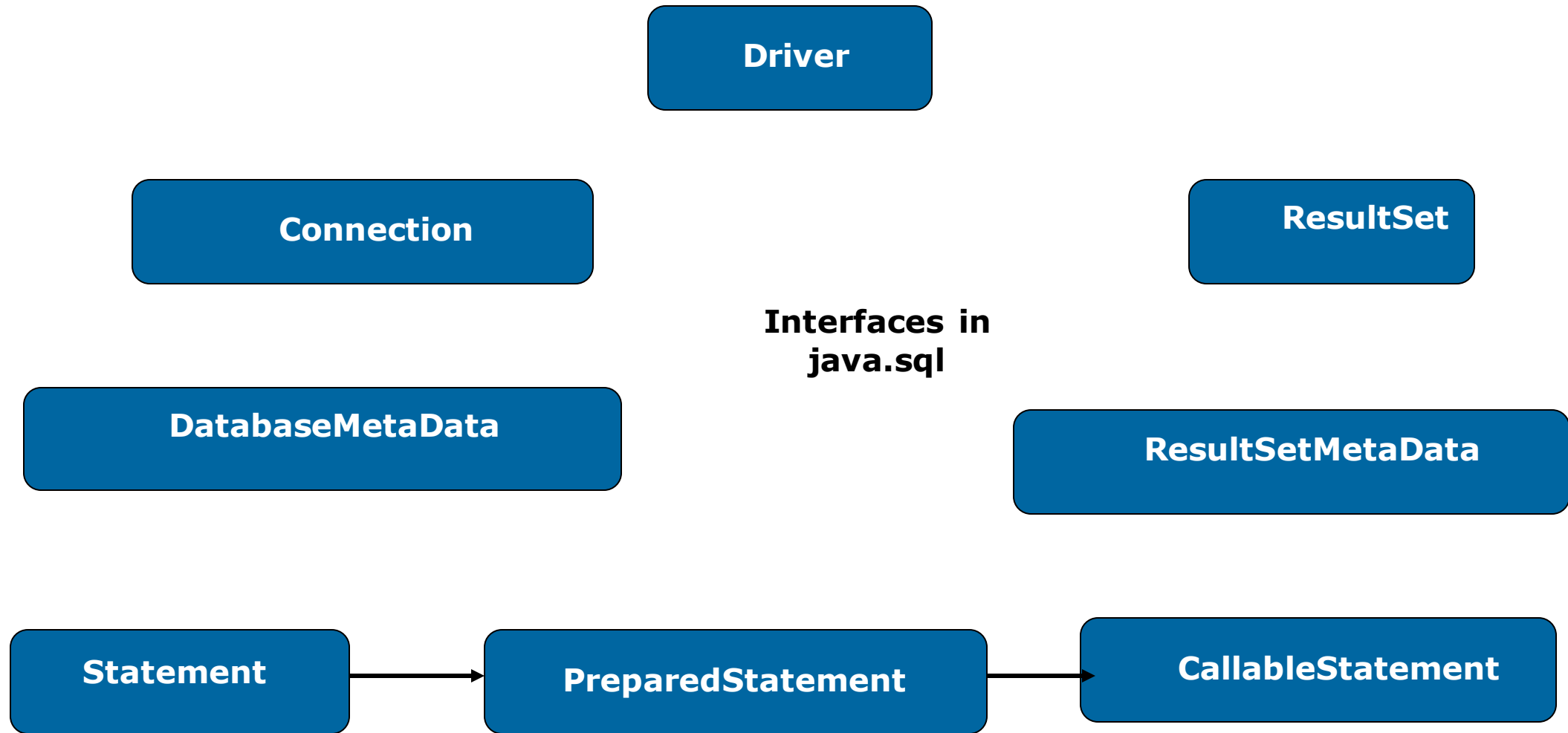
- ▶ JDBC API consist of different classes and Interfaces to help interact with the Database



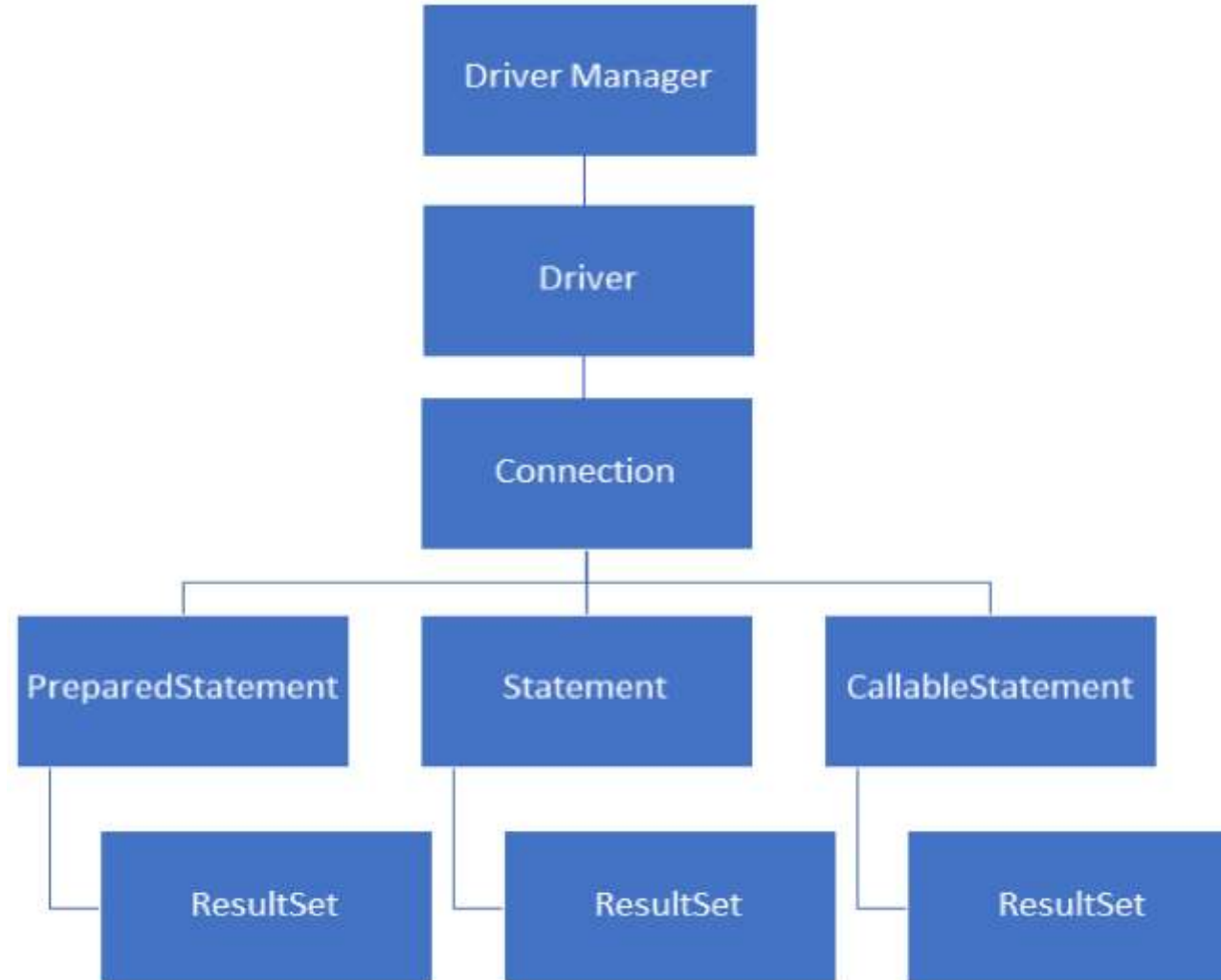
JDBC API

- ▶ JDBC has four primary pieces, used for each database access phase:
 - DriverManager:
 - the DriverManager class loads and configures a database driver on the database
 - Connection:
 - The Connection class performs connection and authentication to a database
 - Statement / PreparedStatement:
 - The Statement and PreparedStatement classes send SQL statements to the database engine for preprocessing and eventually execution
 - ResultSet:
 - the ResultSet class allows for the inspection of results from executions

JDBC API



JDBC Component



Dealing with Database

- ▶ Load or register the driver with `Class.forName()`.
- ▶ Establish Connection with the database with Connection Interface.
- ▶ Create Statement objects for Queries.
- ▶ Execute the statements which may or may not return `ResultSet`.
- ▶ Manipulate the `ResultSet`.
- ▶ Close the Statements and Connection objects.

Dealing with Database

- ▶ Load or register the driver with `Class.forName()`.
- ▶ Establish Connection with the database with Connection Interface.
- ▶ Create Statement objects for Queries.
- ▶ Execute the statements which may or may not return `ResultSet`.
- ▶ Manipulate the `ResultSet`.
- ▶ Close the Statements and Connection objects.

Dealing with Database (Contd...)

- Step 1: Load or register the driver with `Class.forName()`.
 - A JDBC driver implements the interfaces and classes used to connect to databases and send queries for a particular DBMS vendor.

`Class.forName()` throws `ClassNotFoundException`

- `sun.jdbc.odbc.JdbcOdbcDriver`
- `jdbc.driver.oracle.OracleDriver`
- `jdbc:sqlserver://`

For Example:

```
try {  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
} catch (ClassNotFoundException e) {  
    System.out.println("Exception : " + e);  
}
```

Types of Drivers

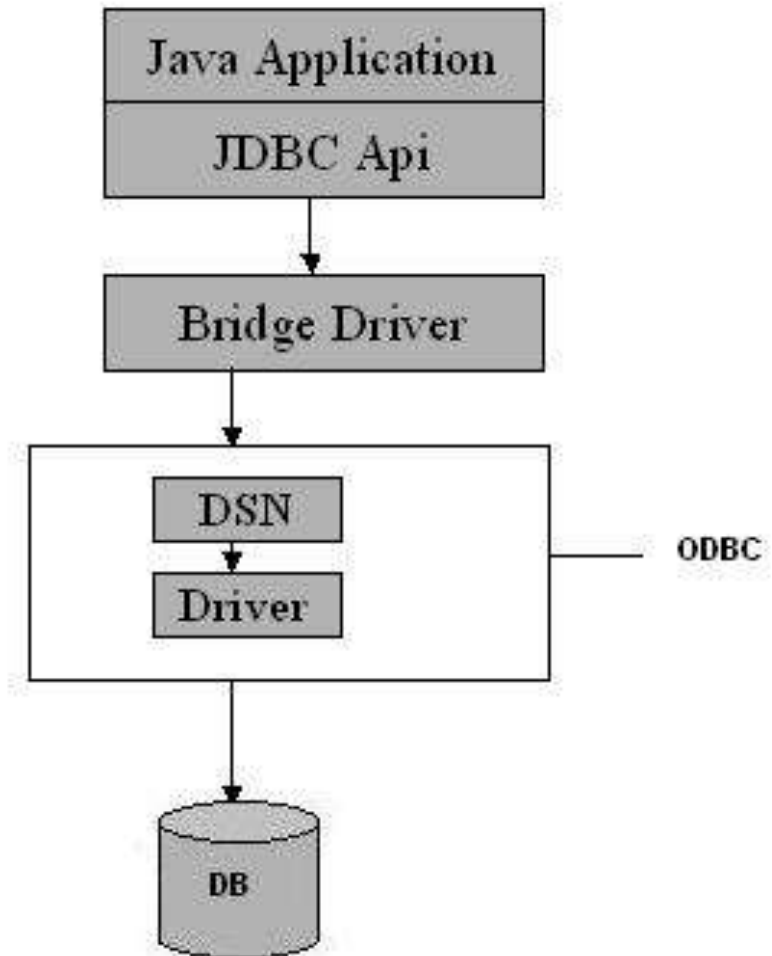
Driver Type	Specification
Type I	Jdbc-Odbc Bridge Driver
Type II	Partly Java, partly native code implementing Vendor specific API
Type III	Pure Java driver requesting either to type I or II as another layer(Middleware)
Type IV	: Pure Java requesting directly to the database

Dealing with Database (Contd...)

- ▶ Step 1: Load or register the driver with `Class.forName()(Contd..)`.
- ▶ Types of Drivers
 - JDBC-ODBC Bridge driver
 - The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API.
 - The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.
- ▶ Advantage
 - The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.
- ▶ Disadvantages
 - 1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
 - 2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
 - 3. The client system requires the ODBC Installation to use the driver.
 - 4. Not good for the Web.

Dealing with Database (Contd...)

- ▶ Step 1: Loading the Driver(Contd..):
- ▶ JDBC-ODBC Bridge driver

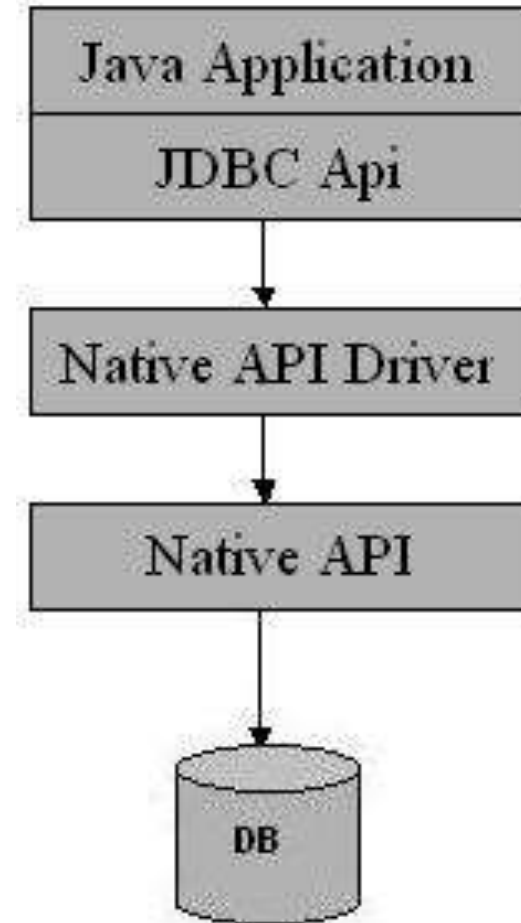


Dealing with Database (Contd...)

- ▶ Step 1: Loading the Driver(Contd..):
 - JDBC Driver :Native-API/partly Java driver
 - The distinctive characteristic of type 2 JDBC drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database.
 - Example: Oracle will have oracle Native API.
- ▶ Advantage
 - They offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less and also it uses Native API which is Database specific.
- ▶ Disadvantage
 - 1. Native API must be installed in the Client System and hence cannot be used for the Internet.
 - 2. It's not written in Java Language which forms a portability issue.
 - 3. If we change the Database we have to change the Native API as it is specific to a database
 - 4. Usually not thread safe. And is mostly obsolete now.

Dealing with Database (Contd...)

- ▶ Step 1: Loading the Driver(Contd..):
 - JDBC Driver :Native-API/partly Java driver



Dealing with Database (Contd...)

- ▶ : Loading the Driver(Contd..):
- ▶ Step 1: Loading the Driver(Contd..):
- ▶ Type3 : All Java/Net-protocol driver
 - Type 3 database requests are passed through the network to the middle-tier server which then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.
- ▶ Advantage
 - This driver is server-based, so there is no need for any vendor database library to be present on client machines.
 - This driver is fully written in Java and hence Portable. It is suitable for the web.
 - The net protocol can be designed to make the client JDBC driver very small and fast to load.
 - The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
 - This driver is very flexible allows access to multiple databases using one driver.
 - They are the most efficient amongst all driver types.

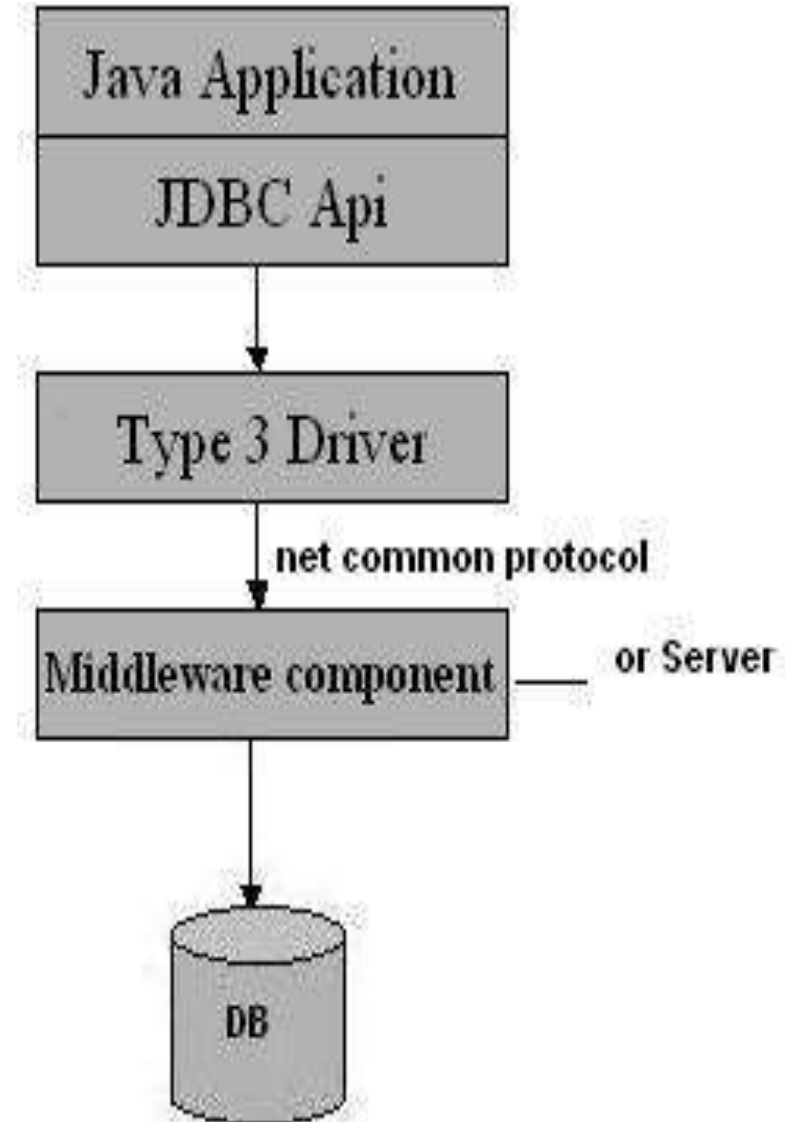
Dealing with Database (Contd...)

► Disadvantage

- It requires another server application to install and maintain.
- Traversing the recordset may take longer, since the data comes through the backend server.

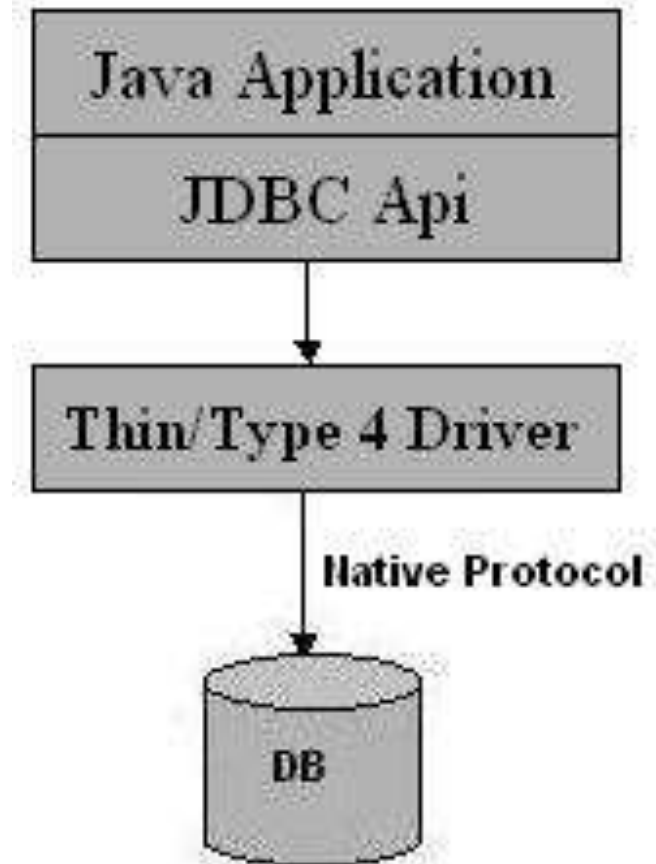
Dealing with Database (Contd...)

- ▶ Type3 : All Java/Net-protocol driver(Contd..)



Dealing with Database (Contd...)

- ▶ Type4: Native-protocol/all-Java driver



Dealing with Database (Contd...)

- ▶ Type4: Native-protocol/all-Java driver
 - The Type 4 uses java networking libraries to communicate directly with the database server.
- ▶ Advantage
 - 1. They are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
 - 2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
 - 3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
 - Disadvantage
- ▶ With type 4 drivers, the user needs a different driver for each database.

Dealing with Database (Contd...)

- ▶ Step 2: Establish Connection with the database with Connection Interface.
 - A JDBC URL has the following syntax

String url= jdbc:<subprotocol>:<subname>

// for odbc

String url= "jdbc:odbc:employee" ;

// for jdbc – Oracle Driver

String url= "jdbc:oracle:thin:@tech:1521:ORCL" ;

**Connection con = DriverManager.getConnection
("url","myLogin", "myPassword")**

// for jdbc – SQL Driver

jdbc:sqlserver://localhost;databaseName=biblio;integratedSecurity=true;

Dealing with Database (Contd...)

► Step 2: Establish Connection with the database with Connection Interface(Contd..)

– **DriverManager:**

- DriverManager is considered the backbone of JDBC architecture. It manages the JDBC drivers that are installed on the system.
- Its getConnection() method is used to establish a connection to a database.
- A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned.
- An application can have one or more connections with a single database, or it can have many connections with different databases.
- A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

Dealing with Database (Contd...)

- ▶ Step 3: Create Statement objects for Queries.
 - Once a connection is obtained we can interact with the database. Connection interface defines methods for interacting with the database via the established connection.
 - To execute SQL statements, you need to instantiate a Statement object from your connection object by using the `createStatement()` method.
 - `Statement stmt = conn.createStatement();`

Dealing with Database (Contd...)

- ▶ Step 3: Create Statement objects for Queries(Contd..).
 - Three kinds of Statements
 - **Statement:** Execute simple sql queries without parameters.
`Statement createStatement()`
 - **Prepared Statement:** Execute precompiled sql queries with or without parameters.
`PreparedStatement prepareStatement(String sql)`
 - **Callable Statement:** Execute a call to a database stored procedure.
`CallableStatement prepareCall(String sql)`

Dealing with Database (Contd...)

- ▶ Step 3: Create Statement objects for Queries(Contd..).
 - Three kinds of Statements
 - **Statement:** Execute simple sql queries without parameters.
`Statement createStatement()`
 - **Prepared Statement:** Execute precompiled sql queries with or without parameters.
`PreparedStatement prepareStatement(String sql)`
 - **Callable Statement:** Execute a call to a database stored procedure.
`CallableStatement prepareCall(String sql)`

Dealing with Database (Contd...)

- ▶ Step 4: Execute the statements which may or may not return ResultSet.
 - The Statement class has three methods for executing statements:
 - **executeQuery():** For a SELECT statement, the method to use is executeQuery .
 - ResultSet executeQuery(String sql) throws SQLException

ResultSet rs = stmt.executeQuery("SELECT name, age FROM student");

- **executeUpdate():** For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate.

Dealing with Database (Contd...)

- ▶ Step 4: Execute the statements which may or may not return ResultSet.
 - ResultSet executeUpdate() throws SQLException

```
Statement stmt = con.createStatement();  
stmt.executeUpdate("INSERT INTO student " + "VALUES (1, 'Smith', ' Bombay' , 20,  
7646234 ) );
```

- **execute()** : executes an SQL statement that is written as String object.

Dealing with Database (Contd...)

- ▶ Step 5: Manipulate the ResultSet:
 - **ResultSet** provides access to a table of data generated by executing a Statement.
 - The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data.
 - Retrieving data from Resultset
 - **boolean next() throws SQLException**
 - **void close () throws SQLException**
 - **XXX getXXX(int index) throws SQLException**

Dealing with Database (Contd...)

- ▶ Step 5: Manipulate the ResultSet(Contd..):
 - Example:

```
String query = " SELECT name, age FROM student ";  
ResultSet rs = stmt.executeQuery(query);  
while (rs.next()) {  
    String s = rs.getString("name");  
    BigDecimal n = rs.getBigDecimal("age");  
    System.out.println(s + "    " + n);  
}
```

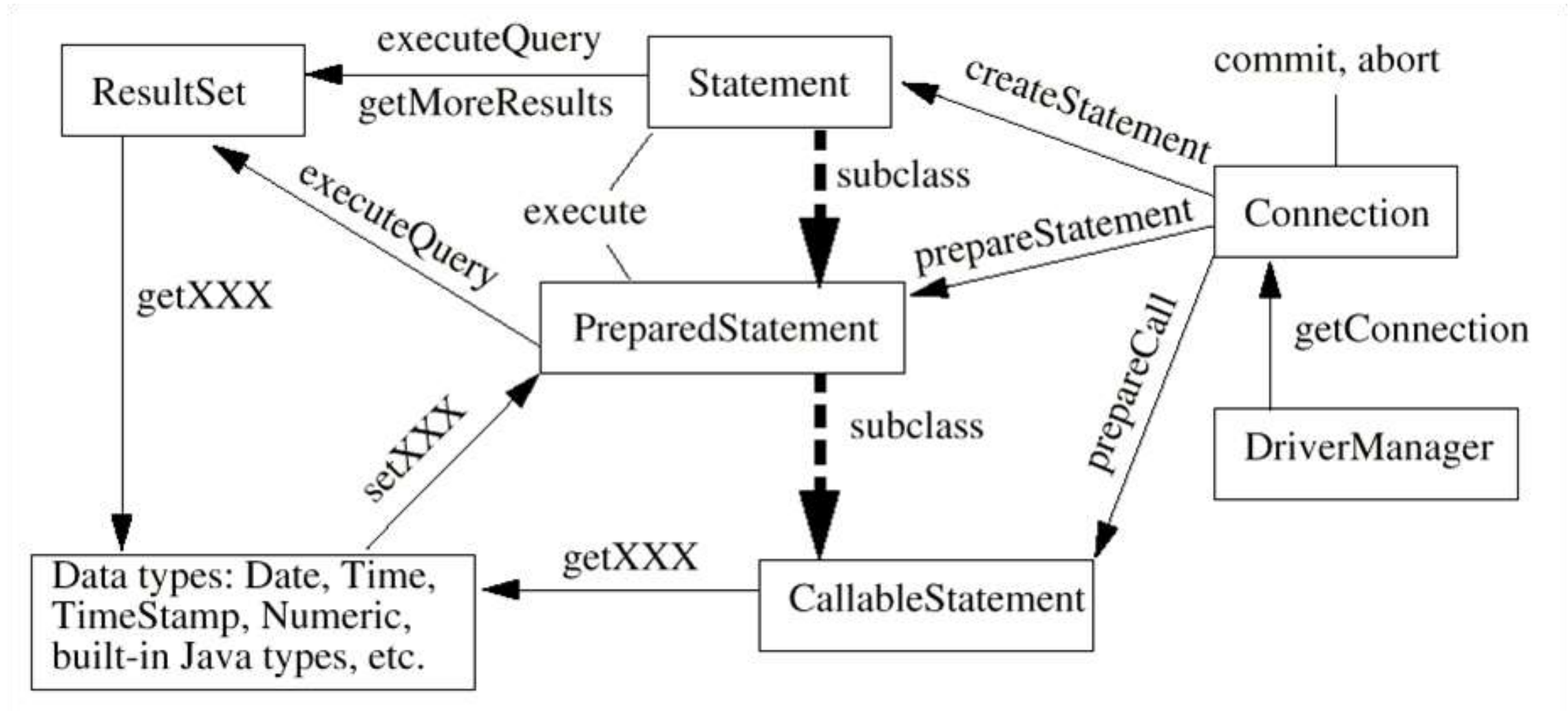
Dealing with Database (Contd...)

- Step 6: Closing the Connections
 - close() is used with different interfaces

`stmt.close()`

`conn.close()`

Dealing with Database (Contd...)



Transaction Management

- ▶ Transactions are not explicitly opened and closed, Instead, the connection has a state called AutoCommit mode
- ▶ if AutoCommit is true, then every statement is automatically committed. Default case: true
- ▶ `Connection.setAutoCommit(boolean)`
- ▶ if AutoCommit is false, then every statement is added to an ongoing transaction
- ▶ You must explicitly commit or rollback the transaction using
- ▶ `Connection.commit()` and
- ▶ `Connection.rollback()`

Dealing with Database (Contd...)

► Step 4: Using Prepared Statement..:

– Example:

```
» PreparedStatement prep = con.prepareStatement( "UPDATE"+  
»      " course values set fees = ? WHERE c-id = ?");  
» prep.setInt(2, 2);  
» prep.setBigDecimal(1, 8000.00);  
» prep.executeUpdate ( );  
» prep.setInt(1, 4);  
» prep.setBigDecimal(2, 9000.00);prep.executeUpdate ( );
```

Dealing with Database (Contd...)

► Step 4: Using Callable Statement..:

– Example:

- » `String sql="execute getEmployees ? ";`
- » `CallableStatement call=con.prepareCall(sql);`

- » `call.registerOutParameter(1,Types.INTEGER);`
- » `call.execute();`
- » `int val=call.getInt(1);`
- » `System.out.println("There are " +val + " employees");`

Dealing with Database (Contd...)

► Step 4: Using Transactions

– Example:

```
» con.setAutoCommit(boolean commit)
» con.commit()
» con.rollback()
» try {
»   con.setAutoCommit(false);
»   // perform transactions
»   con.commit()
»   con.setAutoCommit(true);
» } catch (SQLException e) {
»     con.rollback(); }
```

Dealing with Database (Contd...)

- ▶ We have used ResultSet to fetch the data and convert it into String or int and so on.
- ▶ “But how can you know what is the datatype of the field being fetched ?”

ResultSet MetaData

- ▶ The simple meaning of metadata is data about data.
- ▶ There are two metadata available in the JDBC API –
 - ResultSetMetaData
 - DatabaseMetaData.
- ▶ ResultSetMetaData:
 - It is used to make descriptive information about **ResultSet** object, like; number of columns, name of columns and datatype of columns.
 - It does not provide any information regarding database and how many rows are available in the ResultSet object. Whether ResultSet is read only, updatable or scrollable.
 - To create object of ResultSetMetadata by calling getMetaData() method from **ResultSet** object.

ResultSet MetaData (Contd...)

► Syntax :

```
ResultSetMetaData rsmd=res.getMeataData();  
// res is a valid object of ResultSet object
```

► The following are common methods in ResultSetMetadata interface

- int getColumnCount()
- string getColumnName()
- int getColumnType()
- string getTableName()

ResultSet MetaData (Contd...)

► DatabaseMetaData:

- This interface provide comprehensive information about the database. It contains the information about Database Management System (DBMS) and all the objects in the database, like; all the tables, cataloge name, view, stored procedure etc.
- To create DatabaseMetaData object by calling getMetaData() from **Connection** interface.

– Syntax:

```
DatabaseMetaData dmd=con.getMetaData();  
// con is an object of Connection interface
```

JDBC Summary

- ▶ JDBC providers are prerequisites for data sources, which supply applications with the physical connections to a database.
 - » 1. Load the driver
 - use `Class.forName` otherwise
 - » 2. Define the Connection URL
 - `jdbc:vendor:blah` (vendor gives exact format)
 - » 3. Establish the Connection
 - `DriverManager.getConnection`
 - » 4. Create a Statement object
 - `connection.createStatement`
 - » 5. Execute a query
 - `statement.executeQuery`
 - » 6. Process the results
 - Loop using `resultSet.next()` and Call `getString`, `getInt`, etc.
 - » 7. Close the connection



Questions

- ▶ “If we are using PreparedStatement the execution time will be less. ”
- ▶ Is it true?
 - » **Yes, the statement is true. This is because PreparedStatement object contains SQL statement that has been precompiled. Thus, the DBMS does not have to recompile the SQL statement and prepare an execution plan - it simply runs the statement.**

Questions

- ▶ What Class.forName will do while loading drivers?
 - » **It is used to create an instance of a driver and register it with the Driver Manager. When you have loaded a driver, it is available for making a connection with a DBMS.**

Questions

- ▶ If you need a Statement object to execute many times Which of the Statement object will you prefer?

» **PreparedStatement**

Questions

- ▶ Can we reuse a Statement or must we create a new one for each query?
 - » **Yes the Statement object be reused. We can use the same Statement for any number of queries.**

Questions

- ▶ What do you understand by connection timeout interval ?
 - » **Every database connection has the timeout interval set (depends on database configuration). This is the max inactivity interval for any connection. If not statement is executed within this time the underlying database would invalidate the connection.**

Java Features

Lambda Expressions

What is a lambda expression?

- ▶ In mathematics and computing generally, a lambda expression is a function
- ▶ Lambda expressions in Java introduce the idea of functions into the language
- ▶ In conventional Java terms lambdas can be understood as a kind of anonymous method with a more compact syntax that also allows the omission of modifiers, return type, and in some cases parameter types as well

Lambda Expressions

Why Lambdas in Java Now?

- ▶ **Concise syntax**
 - More succinct and clear than anonymous inner classes
- ▶ **Deficiencies with anonymous inner classes**
 - Bulky, confusion re “this” and naming in general, no non-final vars, hard to optimize
- ▶ **Convenient for new streams library**
 - `shapes.forEach(s -> s.setColor(Color.RED));`
- ▶ **Step toward true functional programming**
 - Real functions and mutable local vars perhaps in future

Lambda Expressions

Advantages: Concise and Expressive

- ▶ Old
 - `button.addActionListener(new ActionListener()`
 - `{`
 - `@Override`
 - `public void actionPerformed(ActionEvent e) {`
 - `doSomething (e);`
 - `}`
 - `});`
- ▶ New
- ▶ `button.addActionListener(e -> doSomething(e));`

Lambda Expressions

Advantages: Support New Way of Thinking

- ▶ Encourage functional programming
 - When functional programming approach is used, many classes of problems are easier to solve and result in code that is clearer to read and simpler to maintain
- ▶ Support streams
 - Streams are wrappers around data sources (arrays, collections, etc.) that use lambdas, support map/reduce, use lazy evaluation, and can be made parallel automatically by compiler

Lambda Expressions

Most Basic Form

- ▶ You write what looks like a function
 - `Arrays.sort(testStrings, (s1, s2) -> s1.length() - s2.length());`
 - `taskList.execute(() -> downloadSomeFile());`
 - `someButton.addActionListener(event -> handleButton());`
- ▶ You get an instance of a class that implements the interface that was expected in that place
 - The expected type must be an interface that has *exactly one* (abstract) method
 - Called “Functional Interface” or “Single Abstract Method (SAM) Interface”

Lambda Expressions

Where Can Lambdas Be Used?

- ▶ Find any variable or parameter that expects an interface that has one method
- ▶ Code that uses interface is the same
 - `public void someMethod(InterfaceRef ref) { ... }`
- ▶ Code that calls interface can supply lambda
 - `String result = someMethod(s -> s.toUpperCase() + "!")`

Lambda Expressions

Simplest Form: Syntax Summary

- ▶ Replace this
 - `new SomeInterface()`
 - `{`
 - `@Override`
 - ▶ `public SomeType someMethod(args) { body }`
 - ▶ `}`
- ▶ With this
 - `(args) -> body`

Lambda Expressions

Simplest Form: Example

► Old style

- `Arrays.sort(testStrings, new Comparator<String>())`
- `{`
 - `@Override`
 - `public int compare(String s1, String s2)`
 - `{`
 - `return(s1.length() - s2.length());`
 - `}`
- `});`

► New style

- `Arrays.sort(testStrings,`
- `(String s1, String s2) -> { return(s1.length() - s2.length()); }`
- `);`

Lambda Expressions

Sorting Strings by Length

► Old version

- `String[] testStrings = {"one", "two", "three", "four"}`
 - `Arrays.sort(testStrings, new Comparator<String>())`
 - `{`
 - `@Override`
 - `public int compare(String s1, String s2)`
 - `{`
 - `return(s1.length() - s2.length());`
 - `}`
 - `});`

► New version

- `Arrays.sort(testStrings,`
- `(String s1, String s2) -> { return(s1.length() - s2.length()); });`

Lambda Expressions

Type Inferencing

- ▶ **Types in argument list can usually be omitted**
- ▶ **Basic lambda**
 - (Type1 var1, Type2 var2 ...) -> { method body }
- ▶ **Lambda with type inferencing**
 - (var1, var2 ...) -> { method body }

Lambda Expressions

Syntax Summary

- ▶ Replace this
 - `new SomeInterface()`
 - `{`
 - `@Override`
 - `public SomeType someMethod(T1 v1, T2 v2)`
 - `{`
 - `//body`
 - `}`
 - `}`
- ▶ With this
- ▶ `(v1, v2) -> { body }`

Lambda Expressions

EXAMPLE

- ▶ Old style
- ▶ `Arrays.sort(testStrings, new Comparator<String>())`
- ▶ `{`
 - `@Override`
 - `public int compare(String s1, String s2)`
 - `{`
 - `return(s1.length() - s2.length());`
 - `}`
- ▶ `});`
- ▶ New style
- ▶ `Arrays.sort(testStrings, (s1, s2) -> { return(s1.length() - s2.length()); });`

Lambda Expressions

Sorting Strings by Length

► Old version

```
– String[] testStrings = {"one", "two", "three", "four"};
– ...
– Arrays.sort(testStrings, new Comparator<String>()
– {
    • @Override
    • public int compare(String s1, String s2)
    • {
    •     return(s1.length() - s2.length());
    • }
– });
```

► New version

```
► Arrays.sort(testStrings,
► (s1, s2) -> { return(s1.length() - s2.length()); });
```

Lambda Expressions

Expression for Body

- ▶ For body, use expression instead of block
 - Value of expression will be the return value, with no explicit “return” needed
- ▶ Previous version
 - `(var1, var2 ...) -> { return(something); }`
- ▶ Lambda with expression for body
 - `(var1, var2 ...) -> something`
- ▶

Lambda Expressions

Omitting Parens

- ▶ **If method takes single arg, parens optional**
 - No type should be used: you must let Java infer the type
 - But omitting types is normal practice anyhow
- ▶ **Previous version**
 - (varName) -> someResult()
- ▶ **Lambda with parentheses omitted**
 - varName -> someResult()

Lambda Expressions

Omitting Parens : example

► Old style

- `button1.addActionListener(new ActionListener()`
- `{`
- `@Override`
- `public void actionPerformed(ActionEvent event)`
- `{`
- `setBackground(Color.BLUE);`
- `}`
- `});`
- **New style**
- `button1.addActionListener(event -> setBackground(Color.BLUE));`

Pipelines and Streams

Streams in a Nutshell

- ▶ **Name is confusing for newbies**
 - Little relationship to IO streams
- ▶ **Streams have more convenient methods than Lists**
 - `forEach`, `filter`, `map`, `reduce`, `min`, `sorted`, `distinct`, `limit`, etc.
- ▶ **Streams have cool properties that Lists lack**
 - Making streams more powerful, faster, and more memory efficient
 - The three coolest properties
 - Lazy evaluation
 - Automatic parallelization
 - Infinite (unbounded) streams
- ▶ **Streams do not store data**
 - They are just programmatic wrappers around existing data sources

Pipelines and Streams

Characteristics of Streams

▶ **Not data structures**

- Streams have *no storage*.
- They carry values from source through a pipeline of operation
- They also never modify the underlying data structure

▶ **Designed for lambdas**

- All Stream operations take lambdas as arguments

▶ **Do not support indexed access**

- You can ask for the first element, but not the second or third or last element. But, see next bullet

▶ **Can easily be output as arrays or Lists**

- Simple syntax to build an array or List from a Stream

▶ **Lazy**

- Many Stream operations are postponed until it is known how much data is eventually needed
- E.g., if you do a 10-second-per-item operation on a 100 element list, then select the first entry, it takes 10 seconds, not 1000 seconds

Pipelines and Streams

Characteristics of Streams

► **Parallelizable**

- If you designate a Stream as parallel, then operations on it will automatically be done concurrently, without having to write explicit multi-threading code

► **Can be unbounded**

- Unlike with collections, you can designate a generator function, and clients can consume entries as long as they want, with values being generated on the fly

Pipelines and Streams

Making Streams: Overview

► Stream

- Streams are not collections: they do not manage their own data. Instead, they are wrappers around existing data structures
- Three most common ways to make Stream
 - `someList.stream()`
 - `Stream.of(arrayOfObjects)`
 - `Stream.of(val1, val2, ...)`

Default Methods

Overview

- ▶ Java provides a facility to create default methods inside the interface.
- ▶ Methods which are defined inside the interface and tagged with default are known as default methods.
- ▶ These methods are non-abstract methods.
- ▶ Static Methods inside Java 8 Interface
 - You can also define static methods inside the interface. Static methods are used to define utility methods

Date and Time API

The java.time package: Classes

- » [java.time.LocalDate class](#)
- » [java.time.LocalTime class](#)
- » [java.time.LocalDateTime class](#)
- » [java.time.MonthDay class](#)
- » [java.time.OffsetTime class](#)
- » [java.time.OffsetDateTime class](#)
- » [java.time.Clock class](#)
- » [java.time.ZonedDateTime class](#)
- » [java.time.ZoneId class](#)
- » [java.time.ZoneOffset class](#)
- » [java.time.Year class](#)
- » [java.time.YearMonth class](#)
- » [java.time.Period class](#)
- » [java.time.Duration class](#)

Nashorn JavaScript Engine

- ▶ Nashorn is a JavaScript engine.
- ▶ It is used to execute JavaScript code dynamically at JVM (Java Virtual Machine).
- ▶ Java provides a command-line tool `jjs` which is used to execute JavaScript code.
- ▶ Example
 - Create a file `hello.js`.

```
var hello = function(){  
    print("Hello Nashorn");  
};  
hello();
```
 - Write command **`jjs hello.js`** in terminal

Nashorn JavaScript Engine

Executing JavaScript file in Java Code

- ▶ You can execute JavaScript file directly from your Java file

```
import javax.script.*;
import java.io.*;
public class NashornExample {
    public static void main(String[] args) throws Exception{
        // Creating script engine
        ScriptEngine ee = new ScriptEngineManager().getEngineByName("Nashorn");
        // Reading Nashorn file
        ee.eval(new FileReader("js/hello.js"));
    }
}
```

Thank you

For more information please contact:

T+ 33 1 98765432

M+ 33 6 44445678

maninsha.mane@atos.net