

Object Oriented Programming using JAVA

Dr. Purushothama B R
Computer Science and Engineering
National Institute of Technology Goa

History

- Java is a general purpose, Object oriented programming language
- It is developed by Sun Microsystems of USA in 1991.
- Originally it was called Oak by James Gosling
- James Gosling is one of the inventor of the Java
- Java was designed for the development of software for consumer electronics devices like TV's , VCR's, toasters, etc.

Java History

- Java was modelled around C and C++ but removed a number of features of C and C++ that were considered as source of problems.
- Java teams goal was to make Java simple, reliable, portable and powerful language.

Java Milestones

- 1990 Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
- 1991 After exploring the possibility of using the most popular object-oriented language C++, the team announced a new language named "Oak".
- 1992 The team, known as Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
- 1993 The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
- 1994 The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.

Milestones

- 1995 Oak was renamed “Java”, due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
- 1996 Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
- 1997 Sun releases Java Development Kit 1.1 (JDK 1.1).
- 1998 Sun releases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2).
- 1999 Sun releases Java 2 Platform, Standard Edition (J2SE) and Enterprise Edition (J2EE).
- 2000 J2SE with SDK 1.3 was released.
- 2002 J2SE with SDK 1.4 was released.
- 2004 J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

Java 2 Features

- Compiled and Interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Familiar, Simple and Small
- Multithreaded and Interactive
- High Performance
- Dynamic and Extensible

Java and C

- Java does not include the C unique statement keywords **sizeof**, and **typedef**.
- Java does not contain the data types **struct** and **union**.
- Java does not define the type modifiers keywords **auto**, **extern**, **register**, **signed**, and **unsigned**.
- Java does not support an explicit pointer type.
- Java does not have a preprocessor and therefore we cannot use **# define**, **# include**, and **# ifdef** statements.
- Java requires that the functions with no arguments must be declared with empty parenthesis and not with the **void** keyword as done in C.
- Java adds new operators such as **instanceof** and **>>>**.
- Java adds labelled **break** and **continue** statements.
- Java adds many features required for object-oriented programming.

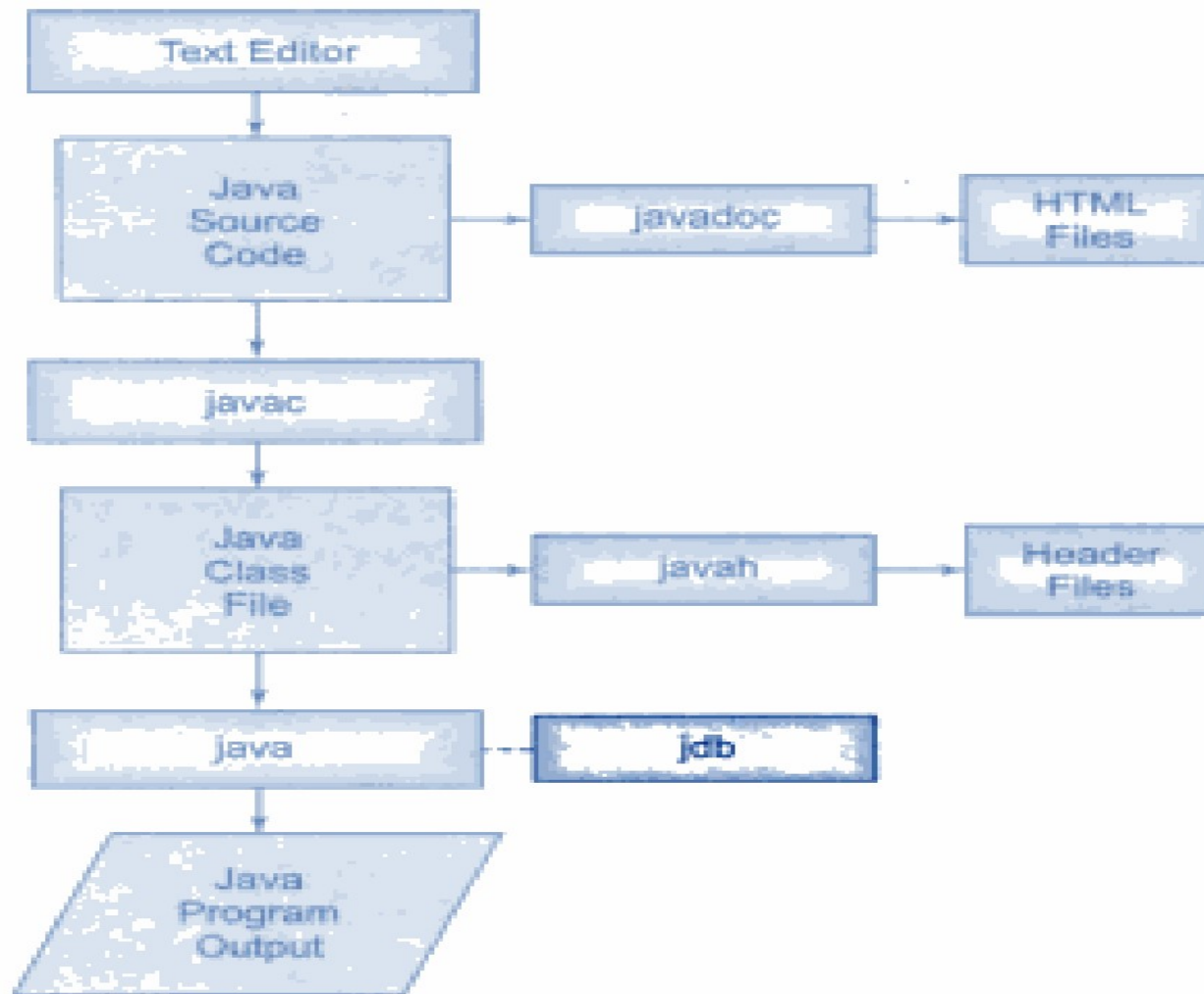
Java and C++

- Java does not support operator overloading.
- Java does not have template classes as in C++.
- Java does not support multiple inheritance of classes. This is accomplished using a new feature called “interface”.
- Java does not support global variables. Every variable and method is declared within a class and forms part of that class.
- Java does not use pointers.
- Java has replaced the destructor function with a `finalize()` function.
- There are no header files in Java.

Application Programming Interface

- **Language Support Package:** A collection of classes and methods required for implementing basic features of Java.
- **Utilities Package:** A collection of classes to provide utility functions such as date and time functions.
- **Input/Output Package:** A collection of classes required for input/output manipulation.
- **Networking Package:** A collection of classes for communicating with other computers via Internet.
- **AWT Package:** The Abstract Window Tool Kit package contains classes that implements platform-independent graphical user interface.
- **Applet Package:** This includes a set of classes that allows us to create Java applets.

Process of Executing and Running Java Programs



Simple Java Program

```
class SampleOne
{
    public static void main (String args[ ])
    {
        System.out.println("Java is better than C++.");
    }
}
```

Another Java Program

```
/*
 * More Java statements
 * This code computes square root
 */
import java.lang.Math;
class SquareRoot
{
    public static void main(String args[ ])
    {
        double x = 5 : // Declaration and initialization
        double y:      // Simple declaration
        y = Math.sqrt(x) ;
        System.out.println("y = " + y);
    }
}
```

Program containing multiple classes

```
class Room
{
    float length;
    float breadth;
    void getdata(float a, float b)
    {
        length = a;
        breadth = b;
    }
}

class RoomArea
{
    public static void main (String args[ ])
    {
        float area;
        Room room1 = new Room( ); // Creates an object room1
        room1.getdata(14, 10);    // Assigns values to length and breadth
        area = room1.length * room1.breadth;
        System.out.println ("Area =" + area);
    }
}
```

Java Program Structure

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

Creating a Program

```
class Test
{
    public static void main (String args[ ])
    {
        System.out.println("Hello!");
        System.out.println("Welcome to the world of Java.");
        System.out.println("Let us learn Java.");
    }
}
```

Compiling the Program

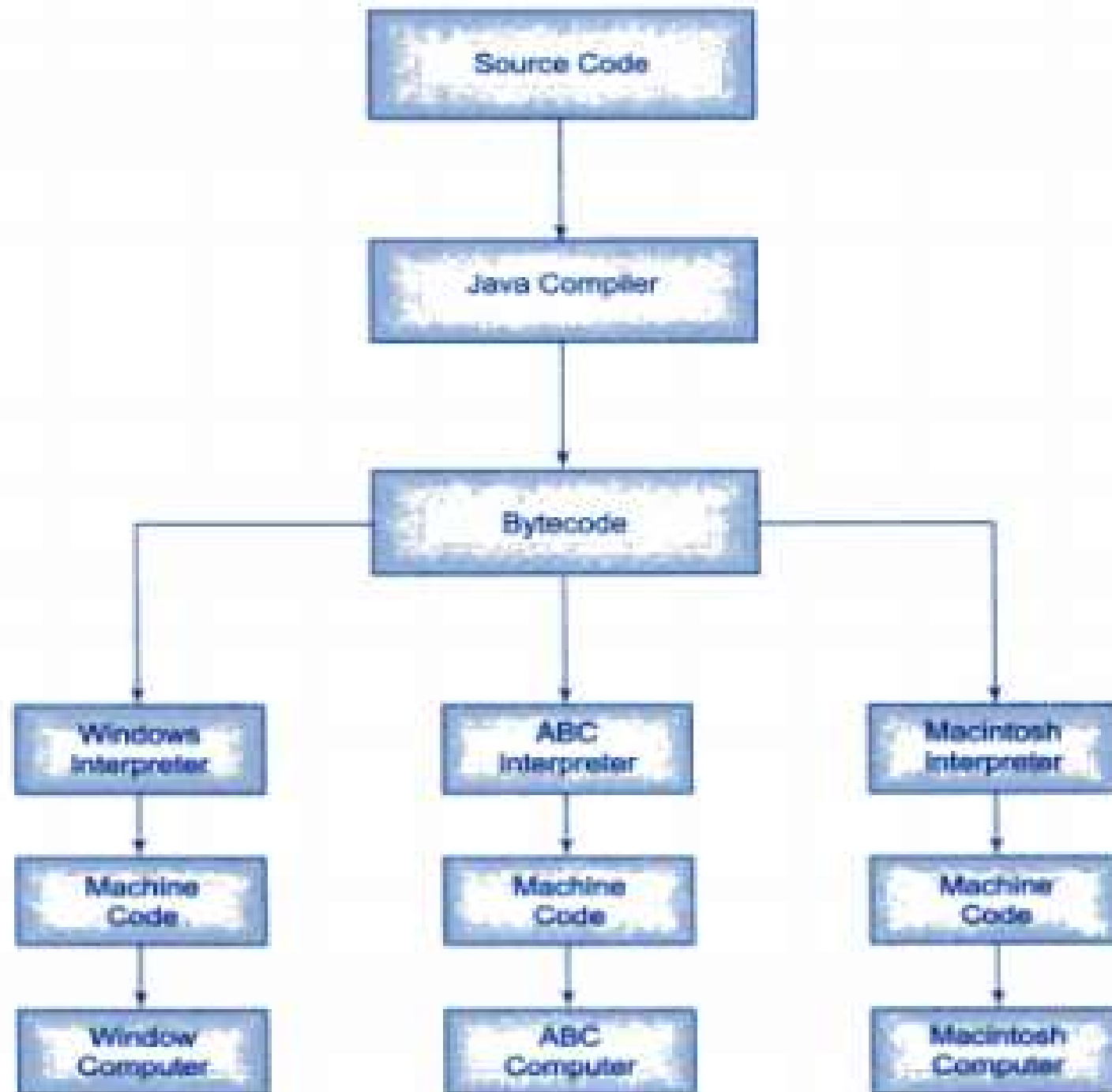
- To compile
 - `javac Test.java`
- If everything is fine, compiler creates `Test.class`
- This contains the Bytecodes of the program
- Compiler automatically names the bytecode file as `<classname>.class`

Running the Program

- To run the program
 - `java Test`

Machine Neutral

- The compiler converts the source file into bytecode files
- These bytecode files are machine independent
- They can be run on any machine
- A program compiled on an IBM Machine will run on Macintosh machine
- Java Interpreter reads bytecode files and translates them to machine code for the specific machine
- **The interpreter is specifically written for each type of machine.**



Java Virtual Machine

- How does Java achieve architecture neutrality?
 - Java produces intermediate code known as **bytecode** for a machine that does not exist
 - This machine is called **Java Virtual Machine**
 - It only exists inside compute memory
 - It is a simulated computer within a computer that does all the major functions of real computer

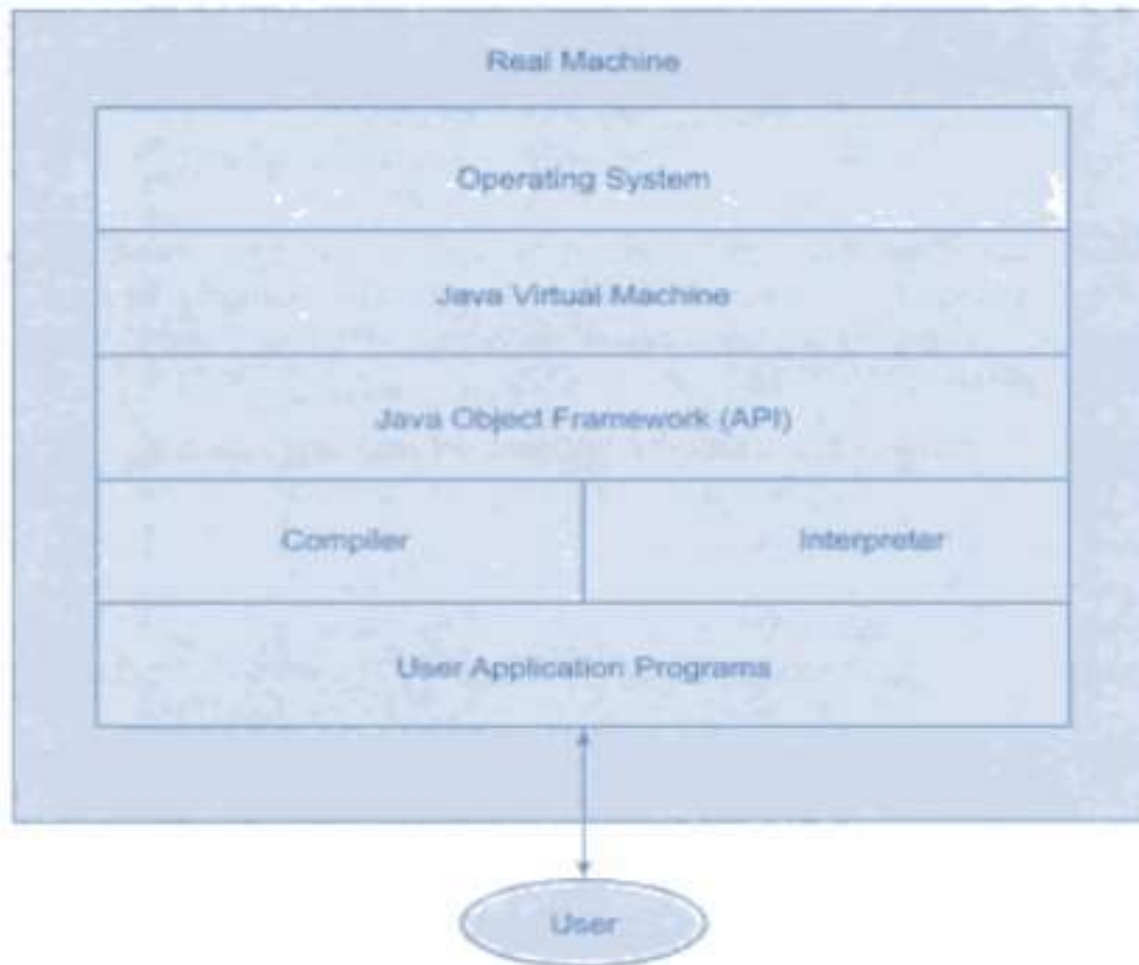
Process of Compilation



Process of Converting Bytecode to Machine Code



Layers of Interactions for Java Programs



Java Development Kit (JDK)

- `appletviewer` (for viewing Java applets)
- `javac` (Java compiler)
- `java` (Java interpreter)
- `javap` (Java disassembler)
- `javah` (for C header files)
- `javadoc` (for creating HTML documents)
- `jdb` (Java debugger)

Description

<i>Tool</i>	<i>Description</i>
appletviewer	Enables us to run Java applets (without actually using a Java-compatible browser).
java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files.
javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand.
javadoc	Creates HTML-format documentation from Java source code files.
javah	Produces header files for use with native methods.
javap	Java disassembler, which enables us to convert bytecode files into a program description.
jdb	Java debugger, which helps us to find errors in our programs.

Object Oriented Programming using JAVA

Dr. Purushothama B R
Computer Science and Engineering
National Institute of Technology Goa

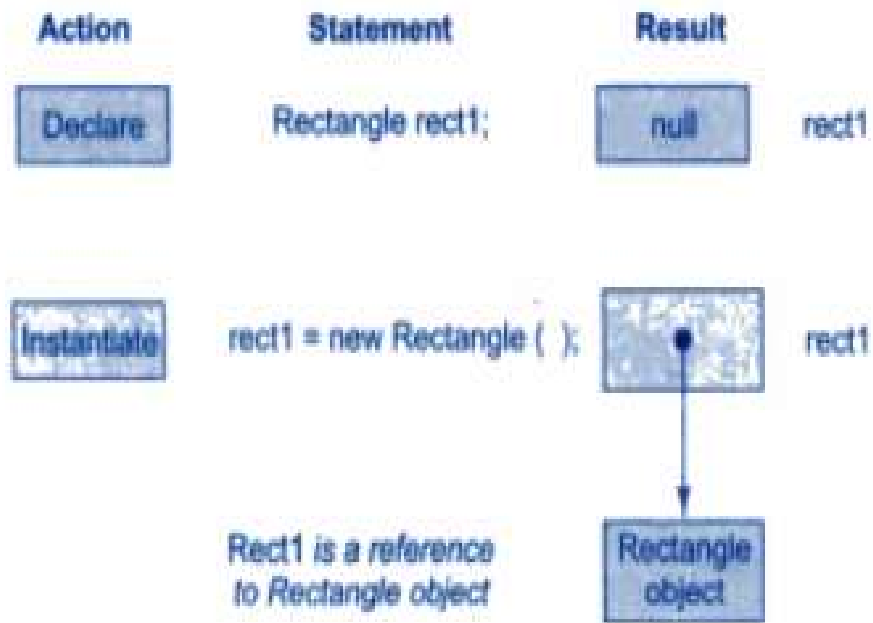
Classes and Objects

```
class Rectangle
{
    int length;
    int width;
    void getData (int x, int y) // Method declaration
    {
        length = x;
        width  = y;
    }
}
```

Creating Objects

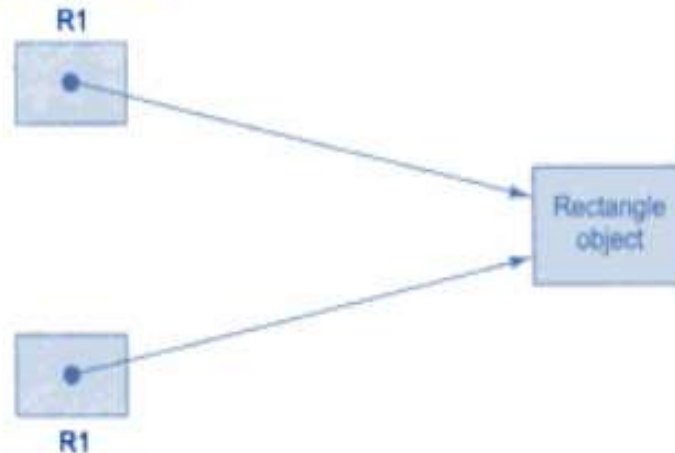
```
Rectangle rect1;           // declare the object  
rect1 = new Rectangle( );  // instantiate the object
```

Scenario



Object Assignment

```
Rectangle R1 = new Rectangle ( );  
Rectangle R2 = R1;
```



Both R1 and R2 refer to the same object.

Accessing Class Members

```
objectname.variablename = value;  
objectname.methodname(parameter-list);
```

Constructors

```
class Rectangle
{
    int length, width;
    Rectangle (int x, int y)                // Defining constructor
    {
        length = x;
        width = y;
    }
    int rectArea( )
    {
        return (length * width);
    }
}
class RectangleArea
{
    public static void main (string args[ ])
    {
        Rectangle rect1 = new Rectangle(15, 10); // Calling constructor
        int areal = rect1.rectArea( );
        System.out.println("Areal = "+ areal);
    }
}
```


Methods Overloading

```
class Room
{
    float length;
    float breadth;
    Room(float x, float y)           // constructor1
    {
        length = x;
        breadth = y;
    }
    Room(float x)                   // constructor2
    {
        length = breadth = x;
    }
    int area( )
    {
        return (length * breadth);
    }
}
```

Static Members

- Suppose we want to define a member that is common to all the objects and accessed without using a particular object
- Such members can be declared as static
- They belong to the class as a whole rather than the objects created from the class
- Example
 - `static int count`
 - `static int max (int x, int y);`

Static methods

- Like static variables, static methods can be called without using objects
- Java class libraries contain large number of class methods
- For example,
 - The Math class of Java library defined many static methods to perform math operations
 - `float x= Math.sqrt(25.0)`

Defining and Using static members

```
class Mathoperation
{
    static float mul(float x, float y)
    {
        return x*y;
    }
    static float divide (float x, float y)
    {
        return x/y;
    }
}
class MathApplication
{
    public void static main(string args[ ])
    {
        float a = MathOperation.mul(4.0,5.0);
        float b = MathOperation.divide(a,2.0);
        System.out.println("b = "+ b);
    }
}
```

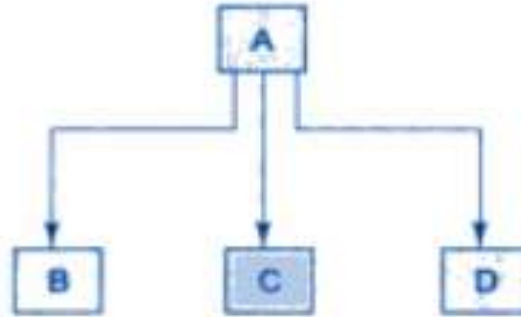
Inheritance

```
class subclassname extends superclassname
{
    variables declaration;
    methods declaration;
}
```

Several Types



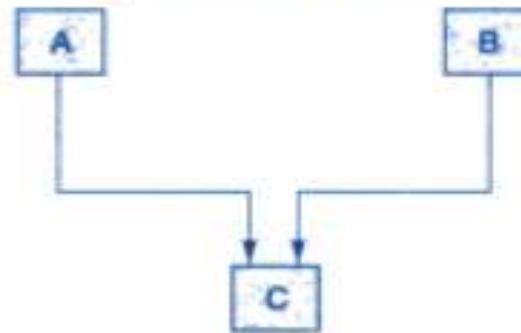
(a) Single inheritance



(b) Hierarchical inheritance



(c) Multilevel inheritance



(d) Multiple inheritance

Subclass constructors

- **super** may only be used within a subclass constructor method
- The call to superclass constructor must appear as the first statement within the subclass constructor
- The parameters in the **super** call must match the order and type of the instance variable declared in the superclass.

```

class Room
{
    int length;
    int breadth;
    Room(int x, int y)
    {
        length = x;
        breadth = y;
    }
    int area( )
    {
        return (length * breadth);
    }
}
class Bedroom extends Room                // Inheriting Room
{
    int height;
    Bedroom(int x, int y, int z)
    {
        super(x, y)                        // pass values to superclass
        height = z;
    }
    int volume( )
    {
        return (length * breadth * height);
    }
}
class InherTest
{
    public static void main(String args[ ])
    {
        Bedroom room1 = new Bedroom(14,12,10);
        int areal = room1.area( );          // superclass method
        int volumel = room1.volume( );      // baseclass method
        System.out.println("Areal = "+ areal);
        System.out.println("Volume = "+ volume);
    }
}

```


Overriding Methods

```
class Super
{
    int x;
    Super(int x)
    {
        this.x = x;
    }
    void display( )           // method defined
    {
        System.out.println("Super x = " + x);
    }
}
```

Overriding Methods

```
Class Sub extends Super
{
    int y ;
    sub (int x, int y)
    {
        super(x);
        this.y = y;
    }
    void display( )                // method defined again
    {
        System.out.println("Super x = " + x);
        System.out.println("Sub y = " + y);
    }
}
class OverrideTest
{
    public static void main(String args[ ])
    {
        Sub s1 = new Sub(100,200);
        s1.display( );
    }
}
```

Final Variables and Methods

- All methods and variables can be overridden by default in subclasses
- If we wish to prevent the subclasses from overriding the members of the superclass
- We can declare them as final using the keyword **final**
 - **final int SIZE =100;**
 - **final void showStatus(){.....}**

Final classes

- We may like to prevent a class being further subclassed for security reason
- A class that cannot be subclassed is called **final class**.

```
final class Aclass {.....}  
final class Bclass extends Someclass {.....}
```

Visibility Control: Public

- Any variable or method is visible to entire class in which it is defined
- What if we want to make it visible to all the classes outside this class?
 - Simply declare the variable or method as **public**

```
public int number;  
public void sum( ) {.....}
```

Friendly access

- In many examples, we have not used public modifier
- Yet, they were still accessible in other classes of the program
- When no access specifier is specified, the member defaults to a limited version of public accessibility known as “friendly level” access.

Difference of Friendly and Public Access

- Public modifier makes all the fields visible in all the classes,
 - Regardless of their packages
- While the friendly access makes fields visible only in the same package, but not in other packages

Protected Access

- Protected access modifier
 - Makes the fields visible not only to all classes and subclasses in the same package
 - Also to subclasses in other packages
- Non-subclasses in other packages cannot access protected member

Visibility of fields in classes

<div>Access modifier →</div> <div>Access location ↓</div>	<i>public</i>	<i>protected</i>	<i>friendly (default)</i>	<i>private protected</i>	<i>private</i>
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Rules of Thumb

1. Use **public** if the field is to be visible everywhere.
2. Use **protected** if the field is to be visible everywhere in the current package and also subclasses in other packages.
3. Use "default" if the field is to be visible everywhere in the current package only.
4. Use **private protected** if the field is to be visible only in subclasses, regardless of packages.
5. Use **private** if the field is *not* to be visible anywhere except in its own class.

Object Oriented Programming using JAVA

Dr. Purushothama B R
Computer Science and Engineering
National Institute of Technology Goa

Arrays

- Creating an array
 - Declaring an array
 - Creating memory locations
 - Putting values into memory locations
- Declaration of arrays

Form 1

```
type arrayname[ ]:
```

Form 2

```
type [ ] arrayname:
```

Arrays

Examples:

```
int  
float  
int[ ]  
float[ ]
```

```
number[ ]:  
average[ ]:  
counter:  
marks:
```

Note: The array size is not entered in the declaration

Creating arrays

- After declaring, we need to create it in the memory
 - Java allows us to create arrays using **new operator only**

```
arrayname = new type[size];
```

Creating arrays

Examples:

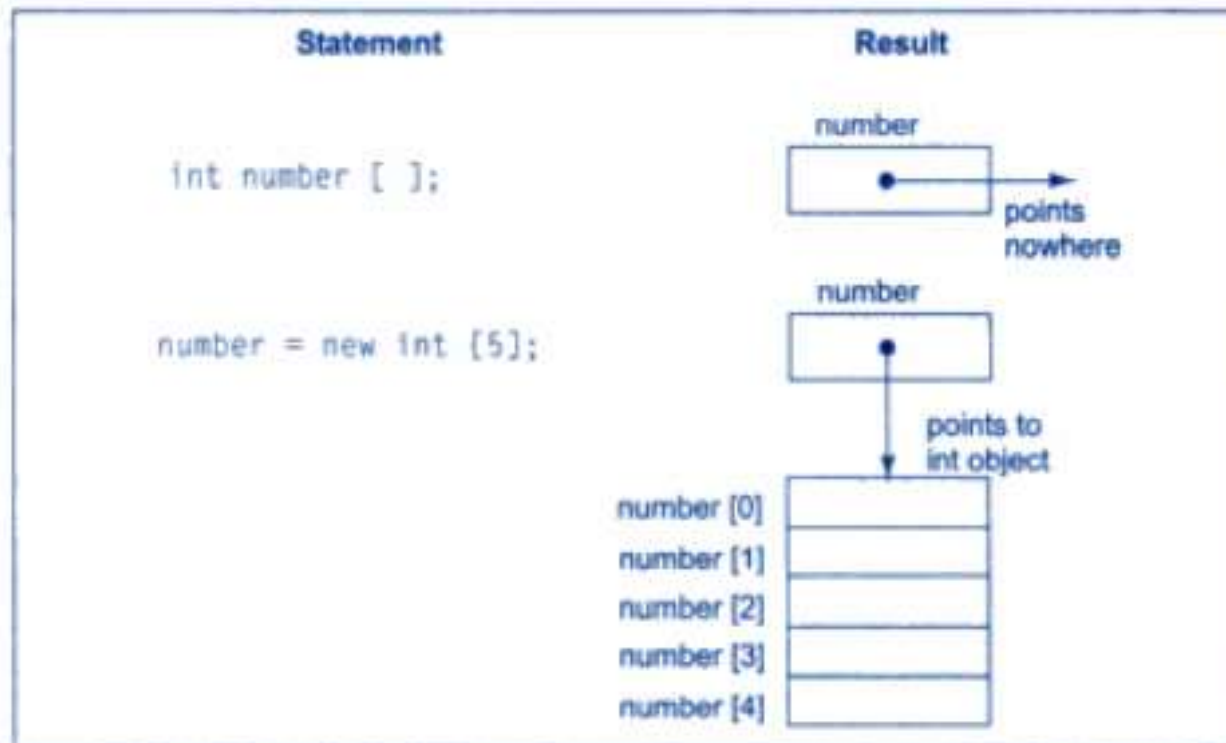
```
number = new int[5];  
average = new float[10];
```

Creating Arrays

- You can combine two steps into one

```
int number[ ] = new int[5];
```


Creation of array in memory



Two dimensional arrays

```
int myArray[ ][ ]:  
myArray = new int[3][4];
```

OR

```
int myArray[ ][ ] = new int[3][4];
```

Strings

- String manipulation is the most common part of many Java programs
- Strings represent a sequence of characters
- The easiest way to represent sequence of characters in Java is by using character array

Example:

```
char charArray[ ] = new char[4];  
charArray[0]      = 'J';  
charArray[1]      = 'a';  
charArray[2]      = 'v';  
charArray[3]      = 'a';
```

Strings

- Although character arrays have advantage of being able to query their length,
 - They themselves are not good enough to support the range of operations we may like to perform on strings
- For example,
 - Copying one character array into another might require lot of book keeping effort
- Java is equipped to handle these situations efficiently.

String and StringBuffer class

- In Java, strings are class objects
- These are implemented using two classes
 - String
 - StringBuffer
- A Java string is an instantiated object of the String class
- Java strings, as compared to C strings are more reliable and predictable

String and StringBuffer class

- This is basically due to C's lack of bound checking.
- A Java string is not a character array and is not NULL terminated

String creation

- Strings may be declared and created as below

```
String stringName;  
stringName = new String ("string");
```

Example:

```
String firstName;  
firstName = new String("Anil");
```

String creation

- Two statements can be combined into one

```
String firstName = new String ("Anil");
```


Operations

- It is possible to get the length of the string using the length method of the String class

```
int m = firstName.length( );
```

- Java Strings can be concatenated using '+' operator

```
String fullName = name1 + name2;  
String city1    = "New" + "Delhi";
```

String arrays

- The following will create itemArray of size 3 to hold string constants

```
String itemArray[ ] = new String[3];
```

Most commonly used String Class methods

<i>Method Call</i>	<i>Task performed</i>
<code>s2 = s1.toLowerCase;</code>	Converts the string s1 to all lowercase
<code>s2 = s1.toUpperCase;</code>	Converts the string s1 to all Uppercase
<code>s2 = s1.replace('x', 'y');</code>	Replace all appearances of x with y
<code>s2 = s1.trim();</code>	Remove white spaces at the beginning and end of the string s1
<code>s1.equals(s2)</code>	Returns 'true' if s1 is equal to s2
<code>s1.equalsIgnoreCase(s2)</code>	Returns 'true' if s1 = s2, ignoring the case of characters
<code>s1.length()</code>	Gives the length of s1
<code>s1.charAt(n)</code>	Gives nth character of s1
<code>s1.compareTo(s2)</code>	Returns negative if s1 < s2, positive if s1 > s2, and zero if s1 is equal s2
<code>s1.concat(s2)</code>	Concatenates s1 and s2
<code>s1.substring(n)</code>	Gives substring starting from n th character
<code>s1.substring(n, m)</code>	Gives substring starting from n th character up to m th (not including m th)
<code>String.valueOf(p)</code>	Creates a string object of the parameter p (simple type or object)
<code>p.toString()</code>	Creates a string representation of the object p
<code>s1.indexOf('x')</code>	Gives the position of the first occurrence of 'x' in the string s1
<code>s1.indexOf('x', n)</code>	Gives the position of 'x' that occurs after nth position in the string s1
<code>String.valueOf(Variable)</code>	Converts the parameter value to string representation

StringBuffer Class

- StringBuffer is a peer class of String
- String creates string of fixed length
- StringBuffer creates strings of flexible length
 - That can be modified in terms of both length and content
- We can insert characters and substrings in the middle of the string OR
- Append another string to the end.

Commonly used StringBuffer

<i>Method</i>	<i>Task</i>
<code>s1.setCharAt(n, 'x')</code>	Modifies the nth character to x
<code>s1.append(s2)</code>	Appends the string s2 to s1 at the end
<code>s1.insert(n, s2)</code>	Inserts the string s2 at the position n of the string s1
<code>s1.setLength(n)</code>	Sets the length of the string s1 to n. If <code>n < s1.length()</code> s1 is truncated. If <code>n > s1.length()</code> zeros are added to s1

Vectors

- Java supports the concept of variable arguments to methods (Exercise!!!)
- This feature can also be achieved in Java through the use of Vector class
- This class is contained in java.util package
- This class is used to create generic dynamic array known as vector
 - That can hold objects of any type and any number.

Vectors

- The objects do not have to be homogeneous.
- Arrays can be easily implemented as vectors
- Vectors are created like arrays as follows

```
Vector intVect = new Vector( ); // declaring without size  
Vector list    = new Vector(3); // declaring with size
```

- Vectors can be declared without specifying any size explicitly.
- A vector without size can accommodate an unknown number of items.

Vectors

- Remember, an array must always have its size specified
- Advantages of vectors over arrays

1. It is convenient to use vectors to store objects.
2. A vector can be used to store a list of objects that may vary in size.
3. We can add and delete objects from the list as and when required.

Important Vector class Methods

<i>Method Call</i>	<i>Task performed</i>
<code>list.addElement(item)</code>	Adds the item specified to the list at the end
<code>list.elementAt(10)</code>	Gives the name of the 10th object
<code>list.size()</code>	Gives the number of objects present
<code>list.removeElement(item)</code>	Removes the specified item from the list
<code>list.removeElementAt(n)</code>	Removes the item stored in the nth position of the list
<code>list.removeAllElements()</code>	Removes all the elements in the list
<code>list.copyInto(array)</code>	Copies all items from list to array
<code>list.insertElementAt (item, n)</code>	Inserts the item at nth position

Major Constraint

- The major constraint is using vectors is
 - We cannot directly store simple data type in a vector
 - We can only store objects
- So, there is a need to convert simple types to objects
- This can be done using wrapper classes.

Wrapper Classes

- Vectors cannot handle primitive data types like int, float, char, and double.
- Primitive data types may be converted into object types by using wrapper classes
- These classes are contained in java.lang package

Wrapper classes for converting simple types

<i>Simple Type</i>	<i>Wrapper Class</i>
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

Wrapper classes

- Wrapper classes have number of unique methods to handle
 - primitive types and objects

Converting primitive numbers to Object numbers using constructor methods

<i>Constructor Calling</i>	<i>Conversion Action</i>
Integer IntVal = new Integer(i);	Primitive integer to Integer object
Float FloatVal = new Float(f);	Primitive float to Float object
Double DoubleVal = new Double(d);	Primitive double to Double object
Long LongVal = new Long(l);	Primitive long to Long object

Converting object numbers to primitive numbers using `typeValue()` methods

<i>Method Calling</i>	<i>Conversion Action</i>
<code>int i = IntVal.intValue();</code>	Object to primitive integer
<code>float f = FloatVal.floatValue();</code>	Object to primitive float
<code>long l = LongVal.longValue();</code>	Object to primitive long
<code>double d = DoubleVal.doubleValue();</code>	Object to primitive double

Converting numbers to strings using toString() method

<i>Method Calling</i>	<i>Conversion Action</i>
<code>str = Integer.toString(i)</code>	Primitive integer to string
<code>str = Float.toString(f);</code>	Primitive float to string
<code>str = Double.toString(d);</code>	Primitive double to string
<code>str = Long.toString(l);</code>	Primitive long to string

Converting String Objects to Numeric objects using the static method Valueof() method

Method Calling	Conversion Action
DoubleVal = Double.Valueof(str);	Converts string to Double object
FloatVal = Float.ValueOf(str);	Converts string to Float object
IntVal = Integer.Valueof(str);	Converts string to Integer object
LongVal = Long.ValueOf(str);	Converts string to Long object

Converting numeric strings to primitive numbers using parsing methods

<i>Method Calling</i>	<i>Conversion Action</i>
<code>int i = Integer.parseInt(str);</code>	Converts string to primitive integer
<code>long i = Long.parseLong(str);</code>	Converts string to primitive long

Interfaces and Multiple Inheritance

- Java does not support multiple inheritance
- That is
 - Classes in Java cannot have more than one superclass
- Definitions like below are not permitted in Java

```
class A extends B extends C
{
    .....
    .....
}
```

Multiple inheritance

- However, designers of Java could not overlook the importance of multiple inheritance
- A large number of real life applications require the use of multiple inheritance
- Java provides an alternate approach known as interfaces to support the concept of multiple inheritance.

Interfaces

- Although a java class cannot be a subclass of more than one superclass
 - It can implement more than one interface.
- Interface is basically a kind of class
- Like classes interfaces contains
 - Methods and variables
- But with a major difference.

Major Difference

- Major difference is
 - Interfaces define only abstract methods and final fields.
- Interfaces do not specify any code to implement these methods and
- Data fields contain only constants.
- It is the responsibility of the class that implements an interface to define the code for implementation of these methods.

General form of Interface definition

```
interface InterfaceName
{
    variables declaration;
    methods declaration;
}
```

Variables and Method declaration

- The variables are declared as follows

```
static final type VariableName = Value;
```

- Method declaration will contain only a list of methods without any body statements

```
return-type methodName (parameter_list);
```


Example

```
interface Item
{
    static final int code = 1001;
    static final String name = "Fan";
    void display ( ) ;
}
```

Another Example

```
interface Area
{
    final static float pi = 3.142F;
    float compute (float x, float y);
    void show ( );
}
```

Extending interfaces

- Like classes interfaces can be extended

```
interface name2 extends name1
{
    body of name2
}
```

Example

```
interface ItemConstants
{
    int code = 1001;
    string name = "Fan";
}
interface Item extends ItemConstants
{
    void display ( );
}
```

Combine several interfaces to single interface

```
interface ItemConstants
{
    int code = 1001;
    String name = "Fan";
}
interface ItemMethods
{
    void display( );
}
interface Item extends ItemConstants, ItemMethods
{
    .....
    .....
}
```

Implementing Interfaces

```
class classname implements interfacename  
{  
    body of classname  
}
```

More general Form

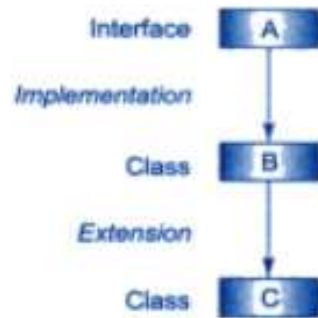
```
class classname extends superclass  
    implements interface1, interface2, .....  
{  
    body of classname  
}
```

```

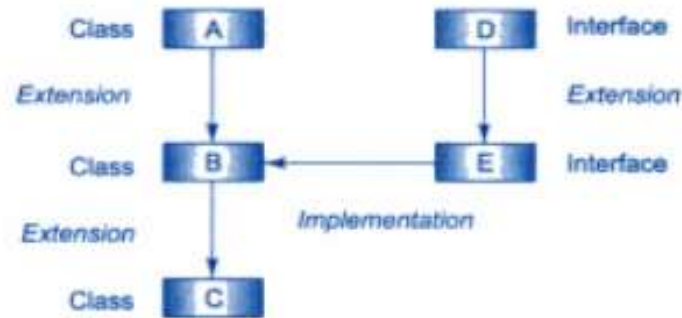
interface Area// Interface defined
{
    final static float pi = 3.14F;
    float compute (float x, float y);
}
class Rectangle implements Area
{
    public float compute (float x, float y)
    {
        return (x*y);
    }
}
class Circle implements Area // Another implementation
{
    public float compute (float x, float y)
    {
        return (pi*x*x);
    }
}
class InterfaceTest
{
    public static void main(String args[ ])
    {
        Rectangle rect = new Rectangle( );
        Circle cir = new Circle( );
        Area area; // Interface object
        area = rect; // area refers to rect object
        System.out.println("Area of Rectangle = "
            + area.compute(10, 20));
        area = cir; // area refers to cir object
        System.out.println("Area of Circle = "
            + area.compute(10, 0));
    }
}

```

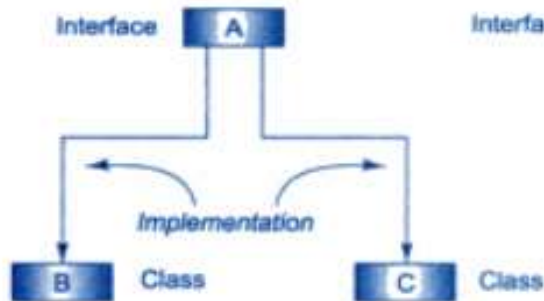

Various forms of Interface Implementation



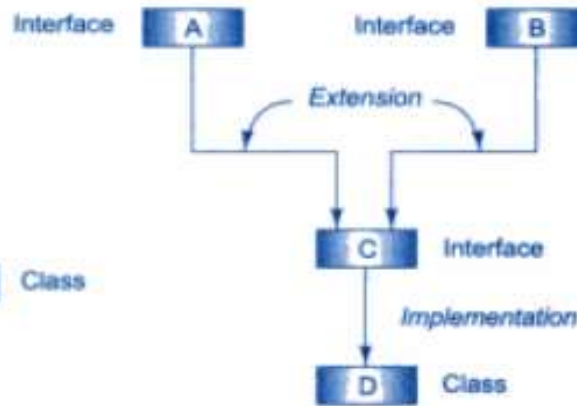
(a)



(b)



(c)



(d)

Thank you

Object Oriented Programming using JAVA

Dr. Purushothama B R
Computer Science and Engineering
National Institute of Technology Goa

Packages: Putting classes together

- Main feature of OOP is to ability to reuse the code already created.
- One way to achieve this is
 - Extending the classes and implementing the interfaces
 - This is limited to reusing the classes within a program
- What if we need
 - To use the classes from other programs without physically copying them into the program under development?

Packages

- This can be accomplished using packages
 - This is a concept similar to class libraries in other languages
- Another way of achieving the reusability in Java therefore, is to use packages.
- Packages are Java's way of
 - Grouping variety of classes and/or interfaces together.
 - The grouping is usually done according to functionalities
 - Packages act like “containers” for classes.

Benefits of Packages

- The classes contained in the packages of the other programs can be easily reused.
- In packages,
 - Classes can be unique compared with classes in other packages
 - Two classes in two different packages can have the same name
 - They may be referred by their fully qualified name
 - Comprising the package name and the class name

Benefits of Packages

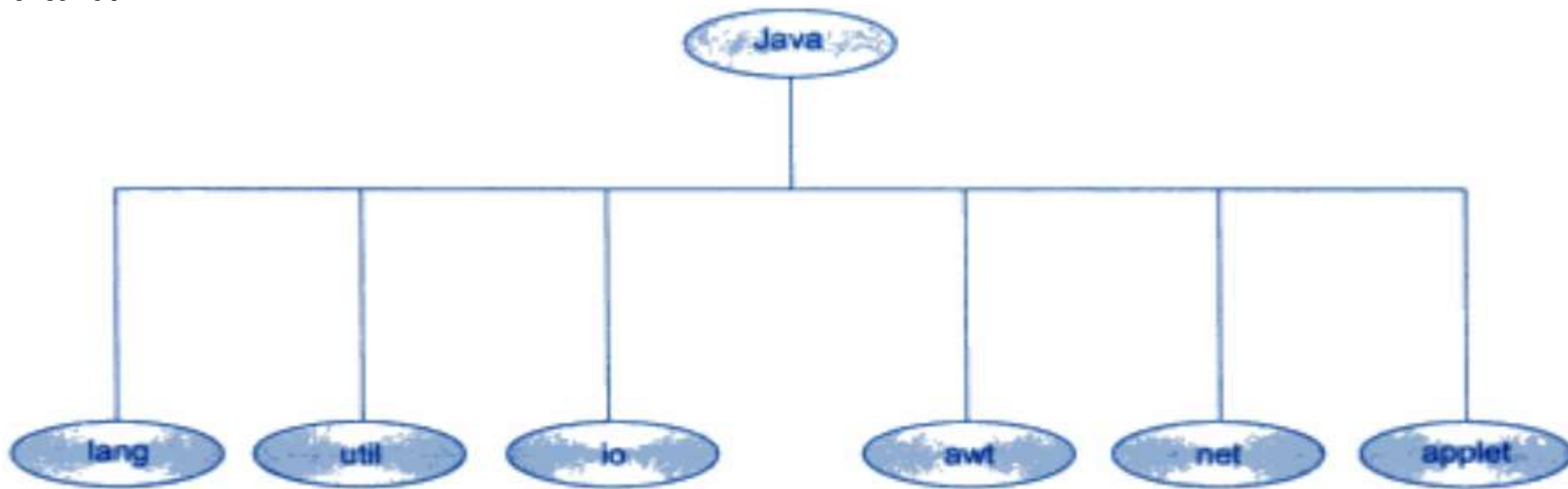
- Packages provide a way to “hide” classes
 - Thus preventing other programs or packages from accessing classes that are meant for internal use only.
- Packages also provide a way of separating “design” from “coding”.

Two categories of Packages

- Java API packages
- User defined packages

Java API packages

- Java API provides a large number of classes grouped into different packages according to functionality.
- Most of the time we use the packages available with the Java API.



Frequently used API packages

Java System Packages and Their classes

`java.lang`

Language support classes. These are classes that Java compiler itself uses and therefore they are automatically imported. They include classes for primitive types, strings, math functions, threads and exceptions.

Java System Packages and Their classes

`java.util`

Language utility classes such as vectors, hash tables, random numbers, date, etc.

Java System Packages and Their classes

`java.io`

Input/output support classes. They provide facilities for the input and output of data.

Java System Packages and Their classes

`java.awt`

Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.

Java System Packages and Their classes

`java.net`

Classes for networking. They include classes for communicating with local computers as well as with internet servers.

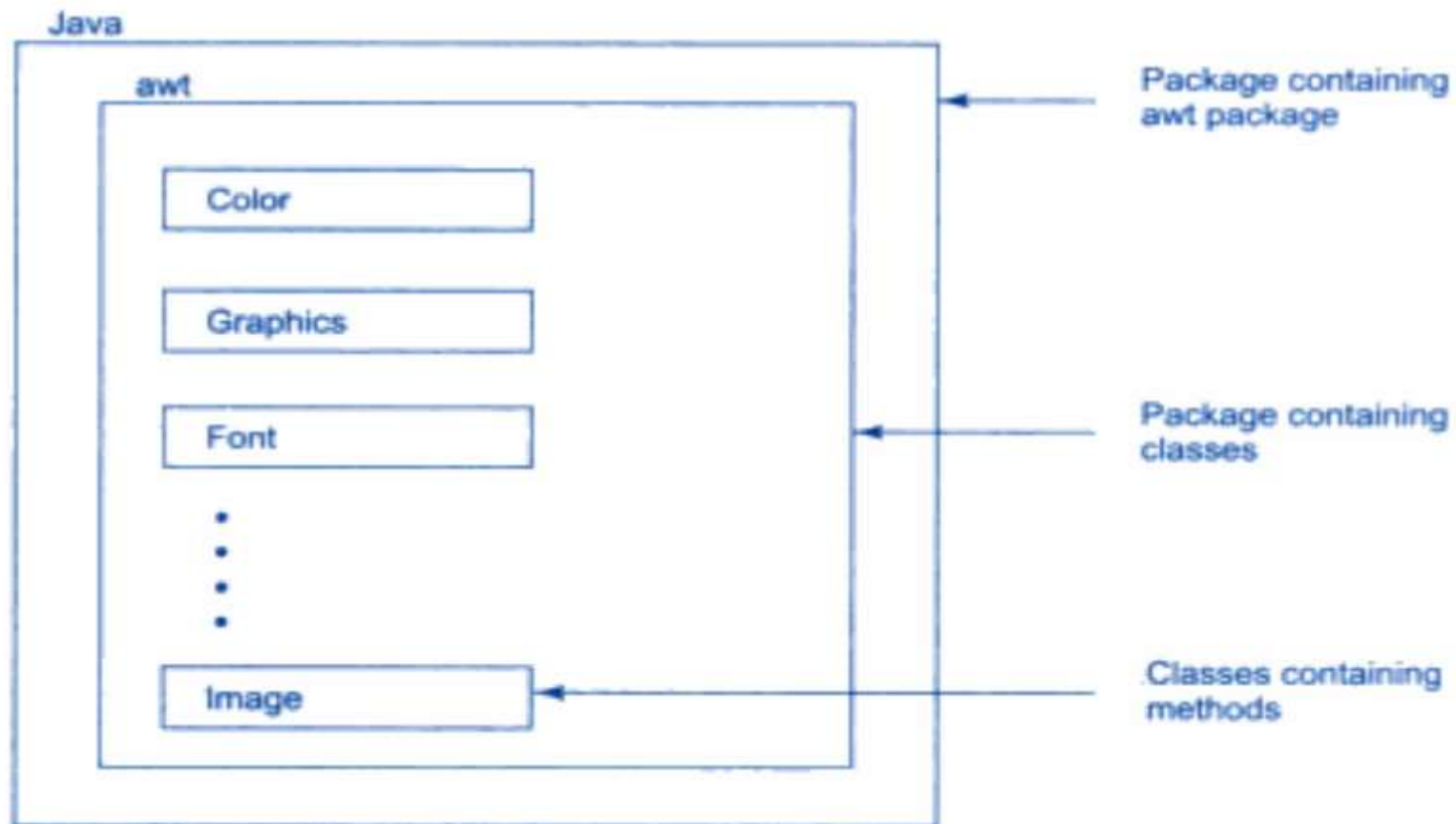
Java System Packages and Their classes

`java.applet`

Classes for creating and implementing applets.

Using system packages

- The packages are organized in a hierarchical structure



Ways of accessing the classes stored in a package

- First approach
 - Use fully qualified class name of the class we want to use.
 - This is done by using the package name containing the class and then
 - Appending the class name to it using the dot operator
- For example
 - To refer to the Color in the awt package

java.awt.Color

Discussion

- This approach is best and easiest
 - If we need to access the class only once or
 - when we need not have to access any other classes of the package

But??

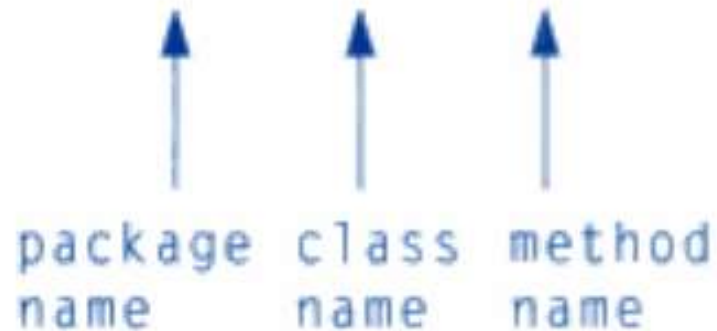
- In many situations,
 - We might want to use a class in number of places in the program or
 - We may like to use many of the classes contained in a package.
- How to achieve this?

To use classes in a package

```
import packagename.classname:  
or  
import packagename.*
```

Naming convention

```
double y = java.lang.Math.sqrt(x);
```



package name class name method name

Caution

- Every package name must be unique to make the best use of packages
- Duplicate names will cause run-time errors
- Since multiple users work on Internet
 - Duplicate names are unavoidable
- Java designer have recognized this problem
 - And suggested a package naming convention that ensures uniqueness
 - Suggests to use the domain names as prefix to preferred package names
 - For Example,
cityname.organisation-name.mypackage

Creating packages

- To create our own packages
 - We must **first declare** the name of the package using **package keyword** followed by package name.
 - This **must be the first statement** in the Java source file (except for comments and whitespaces)
 - Then we define a class, just as we normally define a class.

Creating packages

```
package firstPackage;           // package declaration
public class FirstClass         // class definition
{
    .....
    ..... (body of class)
    .....
}
```


Creating Packages

- The above program is saved as a file `FirstClass.java`
- And located in a directory named `firstPackage`
- When the source files are compiled, Java will create a `.class` file and
 - Store it in the same directory.
- Note that the `.class` files must be located in a directory that has the same name as the package.
- This directory should be a subdirectory of the directory where classes that will import the package are located.

Steps for creating package

1. Declare the package at the beginning of a file using the form

```
package packagename;
```

2. Define the class that is to be put in the package and declare it **public**.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates **.class** file in the subdirectory.

Package Hierarchy

- Java also supports the concept of package hierarchy.
- This is done by specifying the multiple names in a package statement, separated by dots
- For example,
 - `package firstPackage.secondPackage;`
- Note that this package must be stored in a subdirectory named `firstPackage/secondPackage`.

Accessing a Package

- General form of import statement for searching a class is as follows.

```
import package1 [.package2] [.packag3].classname;
```

- Example,

```
import firstPackage.secondPackage.MyClass;
```

Using a package

```
package package1:  
public class ClassA  
{  
    public void displayA( )  
    {  
        System.out.println("Class A");  
    }  
}
```

Using Package

```
import package1. ClassA;  
class PackageTest1  
{  
    public static void main(String args[ ] )  
    {  
        ClassA objectA = new ClassA( ) ;  
        objectA.displayA( );  
    }  
}
```

Another package

```
package package2;  
public class ClassB  
{  
    protected int m = 10  
    public void displayB( )  
    {  
        System.out.println("Class B");  
        System.out.println("m = " + m);  
    }  
}
```

```
import package1.ClassA;
import package2.*;
class PackageTest2
{
    public static void main(String args[ ])
    {
        ClassA objectA = new ClassA( );
        ClassB objectB = new ClassB( );
        objectA.displayA( );
        objectB.displayB( );
    }
}
```


Importing Multiple Packages

```
package pack1:  
public class Teacher  
{.....}  
public class Student  
{.....}  
package pack2:  
public class Courses  
{.....}  
public class Student  
{.....}
```

Import packages

```
import pack1.*:  
import pack2.*:  
Student studetn1;           // create a student object
```

Be Explicit

```
import pack1.*;  
import pack2.*;  
pack1.Student student1:           // OK  
pack2.Student student2:           // OK  
Teacher teacher1:                  // No problem  
Courses course1:                   // No problem
```

Subclassing a imported class

```
// PackageTest3.java
import package2.ClassB;
class ClassC extends ClassB
{
    int n = 20;
    void displayC( )
    {
        System.out.println("Class C");
        System.out.println("m = " + m);
        System.out.println("n = " + n);
    }
}
class PackageTest3
{
    public static void main(String args[ ])
    {
        ClassC objectC = new ClassC( );
        objectC.displayB( );
        objectC.displayC( );
    }
}
```

Access Protection

<div>Access modifier →</div> <div>Access location ↓</div>	<i>public</i>	<i>protected</i>	<i>friendly (default)</i>	<i>private protected</i>	<i>private</i>
Same class	Yes	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	Yes	No
Other classes in same package	Yes	Yes	Yes	No	No
Subclass in other packages	Yes	Yes	No	Yes	No
Non-subclasses in other packages	Yes	No	No	No	No

Adding a class to a package

- Consider the following package

```
package p1;  
public ClassA  
{  
    // body of A  
}
```

- Suppose we want to add another class B to this package.

Do the following!!!

1. Define the class and make it public.

2. Place the package statement

```
package p1;
```

before the class definition as follows:

```
package p1;
public class B
{
    // body of B
}
```

3. Store this as **B.java** file under the directory **p1**.

4. Compile **B.java** file. This will create a **B.class** file and place it in the directory **p1**.

Hiding classes

- When a package is imported using *,
 - All public classes are imported
- Sometimes, we may prefer to “not to import” certain classes.
- We may like to hide these classes from accessing from outside of the package
- Such classes should be declared not public.

Do the following

```
package pl:
public class X           // public class, available outside
{
    // body of X
}
class Y                  // not public, hidden
{
    // body of Y
}
```

Thank you

Object Oriented Programming using JAVA

Dr. Purushothama B R
Computer Science and Engineering
National Institute of Technology Goa

Multithreaded Programming

- Modern operating systems can execute several programs simultaneously.
- This ability is known as **multitasking**
- In system's terminology it is called **multithreading**.

Multithreading

- Multithreading is a conceptual programming paradigm
 - Where a program (process) is divided into two or more subprograms (processes)
 - Which can be implemented at the same time in parallel
- For Example,
 - One subprogram can display animation on the screen
 - Another may build the next animation to be displayed.

Single processor computers

- In most computers, we have only single processor
- Therefore, in reality, the processor is doing only one thing at a time.
- However, processor switches between the processes so fast
 - That it appears to human beings that all of them are being done simultaneously.

Java Programs

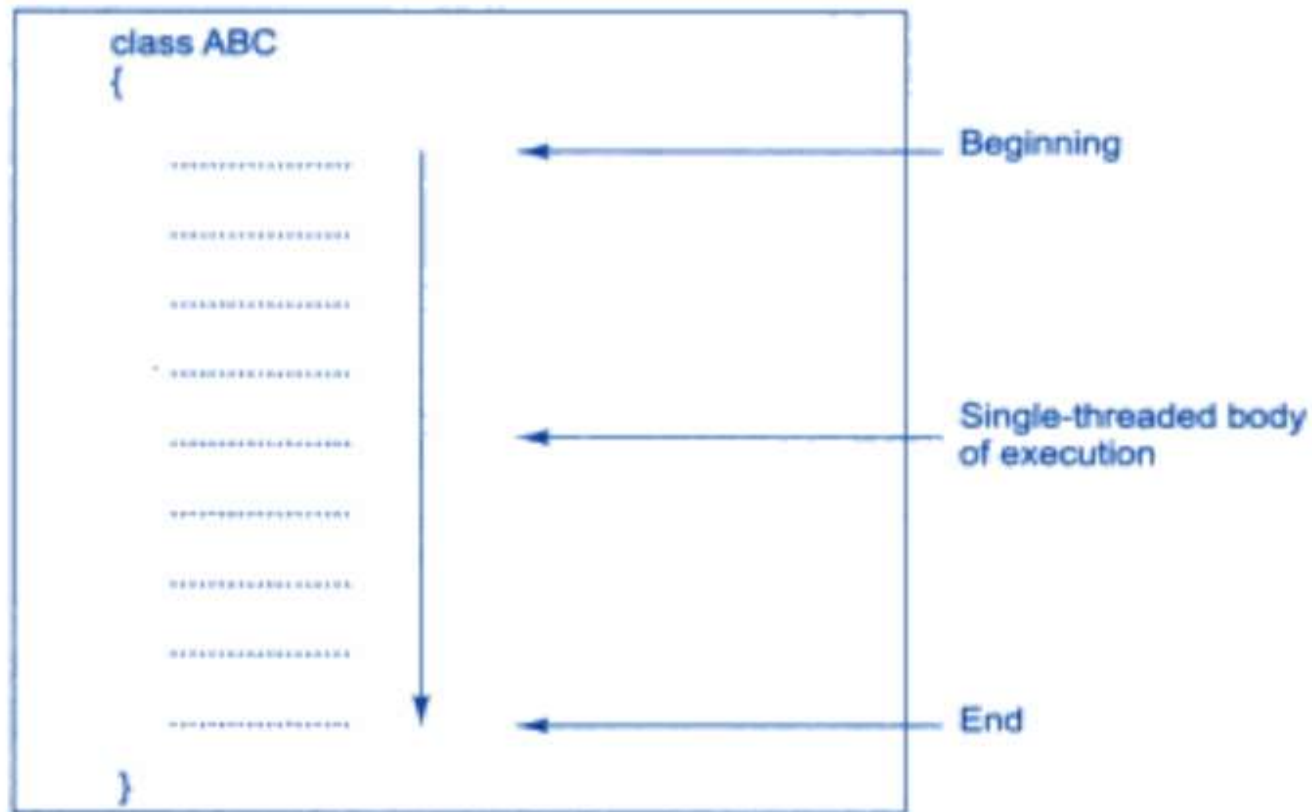
- The Java programs that we have seen so far contain only a single sequential flow of control
- When we execute a program
 - The program begins
 - Runs through a sequence of executions
 - Finally ends.
- At any given point of time, there is only one statement under execution

Thread

- Thread is similar to a program that has a single flow of control
 - It has a beginning
 - A body
 - An end
 - And executes statements sequentially
- All main programs in our discussion till today can be called as single-threaded programs.

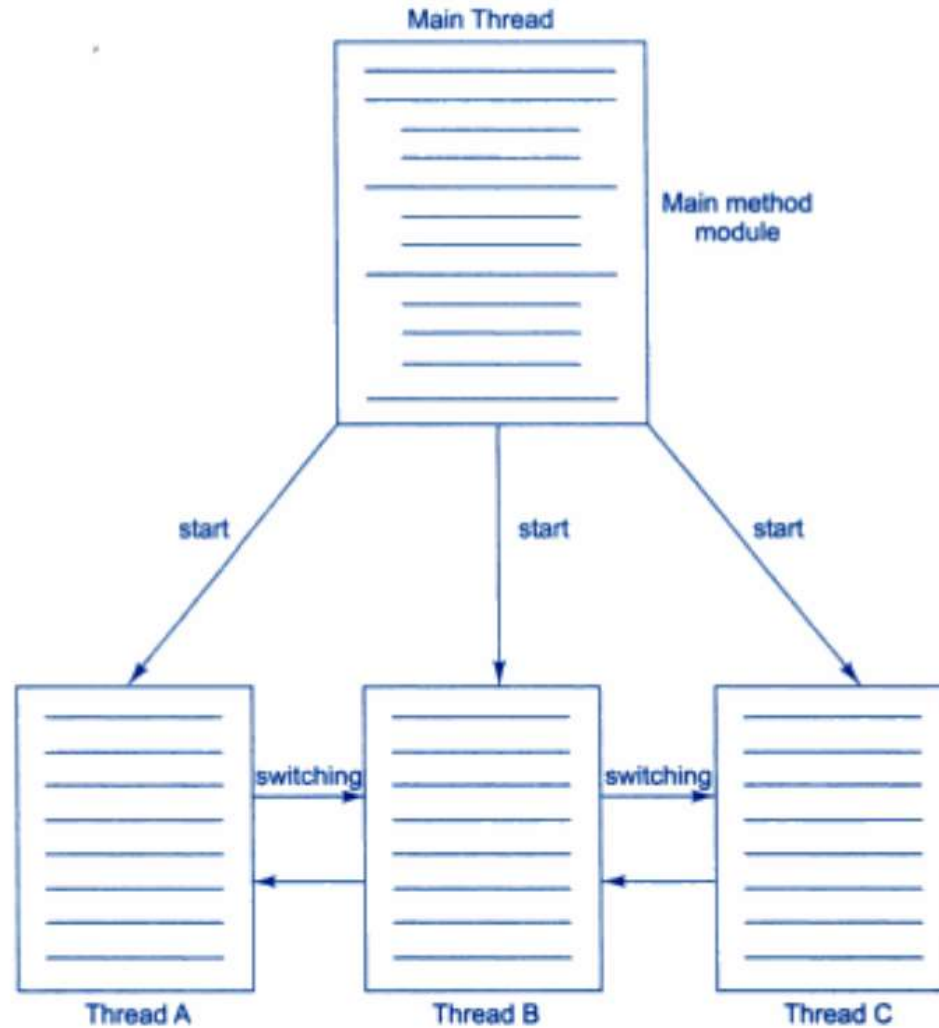
Thread

- Every program will have atleast one thread



Single threaded program

A Multithreaded Program



Creating threads

- Creating threads in Java is simple
- They are implemented in the form of objects that contain a method called `run()`
- The `run()` method is the heart and soul of any thread
- It makes up the entire body of a thread
- And it is the only method in which the threads behavior can be implemented.

run() method

- run() method should be invoked by an object of the concerned thread
- This can be achieved by
 - Creating the thread and
 - Initiating it with the help of another thread method called start()

Creating threads

```
public void run( )  
{  
    .....  
    ..... (statements for implementing thread)  
    .....  
}
```

Two ways to create a thread

- By creating a thread class
 - Define a class that extends Thread class and override its run() method with the code required by the thread
- By converting a class to a thread
 - Define a class that implements a Runnable interface.
 - The Runnable interface has only one method run()

Extending the Thread class

- Our class can be made runnable as thread by extending the class `java.lang.Thread`
- This gives us the access to all the methods in Thread class easily
- Steps
 1. Declare the class as extending the **Thread** class.
 2. Implement the **run()** method that is responsible for executing the sequence of code that the thread will execute.
 3. Create a thread object and call the **start()** method to initiate the thread execution.

Declaring the Class

```
class MyThread extends Thread
{
    .....
    .....
    .....
}
```


Implementing the run() method

```
public void run( )  
{  
    .....  
    ..... // Thread code here  
    .....  
}
```

Starting new thread

```
MyThread aThread = new MyThread( );
```

```
    aThread.start( );
```

Example Program

```
class A extends Thread
{
    public void run( )
    {
        for (int i=1; i<=5; i++)
        {
            System.out.println("\tFrom ThreadA : i = " + i);
        }
        System.out.println("Exit form A ");
    }
}
```

```
class B extends Thread
{
    public void run( )
    {
        for(int j=1; j<=5; j++)
        {
            System.out.println("\tFrom Thread B :j = " + j);
        }
        System.out.println("Exit from B ");
    }
}
```

```
class B extends Thread
{
    public void run( )
    {
        for(int k=1; k<=5; k++)
        {
            System.out.println("\tFrom Thread C : k = " + k);
        }
        System.out.println("Exit from C ");
    }
}
```

```
class ThreadTest
{
    public static void main(String args[ ])
    {
        new A( ).start( );
        new B( ).start( );
        new C( ).start( );
    }
}
```

Typical Output

First run

From	Thread	A	:	i	=	1
From	Thread	A	:	i	=	2
From	Thread	B	:	j	=	1
From	Thread	B	:	j	=	2
From	Thread	C	:	k	=	1
From	Thread	C	:	k	=	2
From	Thread	A	:	i	=	3
From	Thread	A	:	i	=	4
From	Thread	B	:	j	=	3
From	Thread	B	:	j	=	4
From	Thread	C	:	k	=	3
From	Thread	C	:	k	=	4

```

From Thread A : i = 5
Exit from A
From Thread B : j = 5
Exit from B
From Thread C : k = 5
Exit from C

```


Stopping a thread

```
aThread.stop( );
```


Blocking a thread

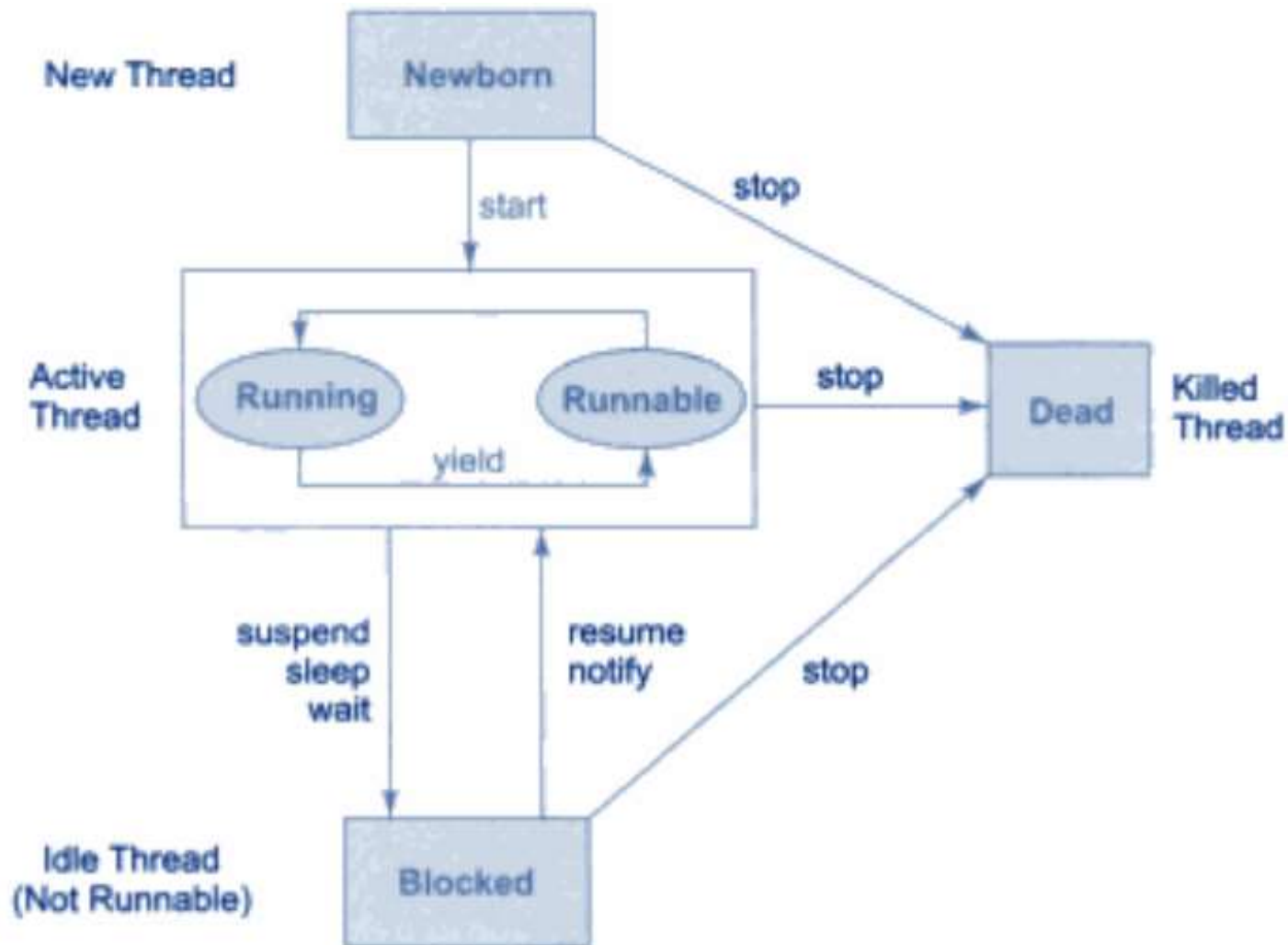
<code>sleep()</code>	<code>// blocked for a specified time</code>
<code>suspend()</code>	<code>// blocked until further orders</code>
<code>wait()</code>	<code>// blocked until certain condition occurs</code>

Lifecycle of a thread

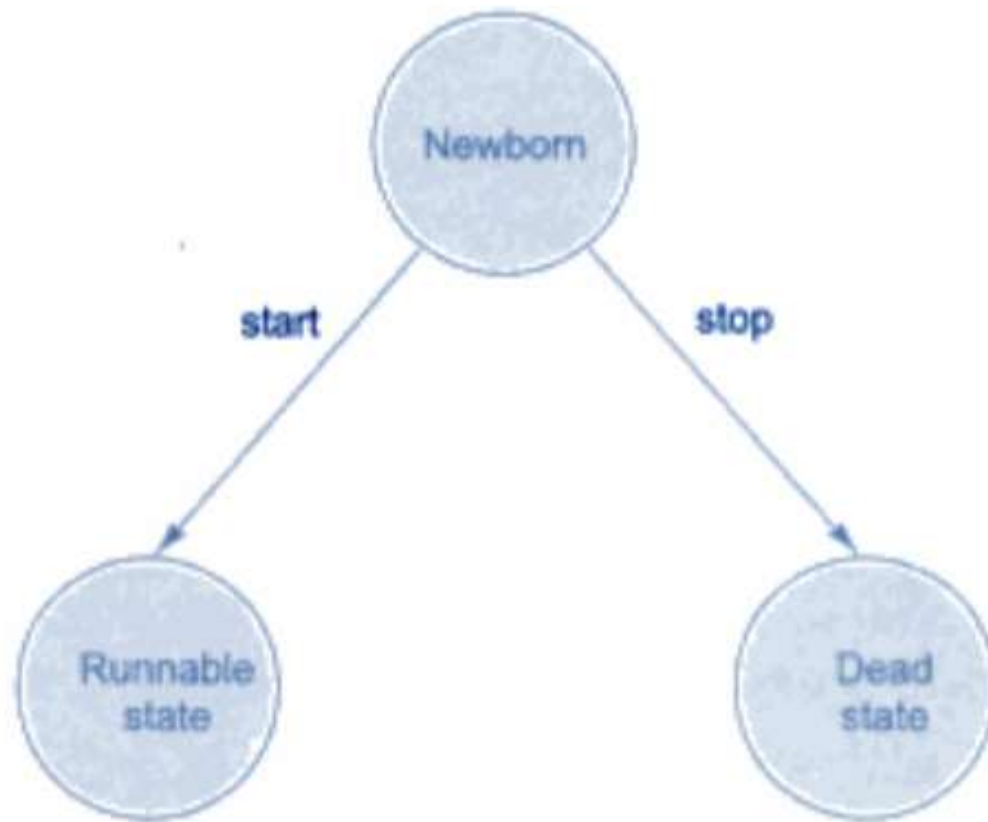
- During a life time of a thread, there are many states it can enter

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

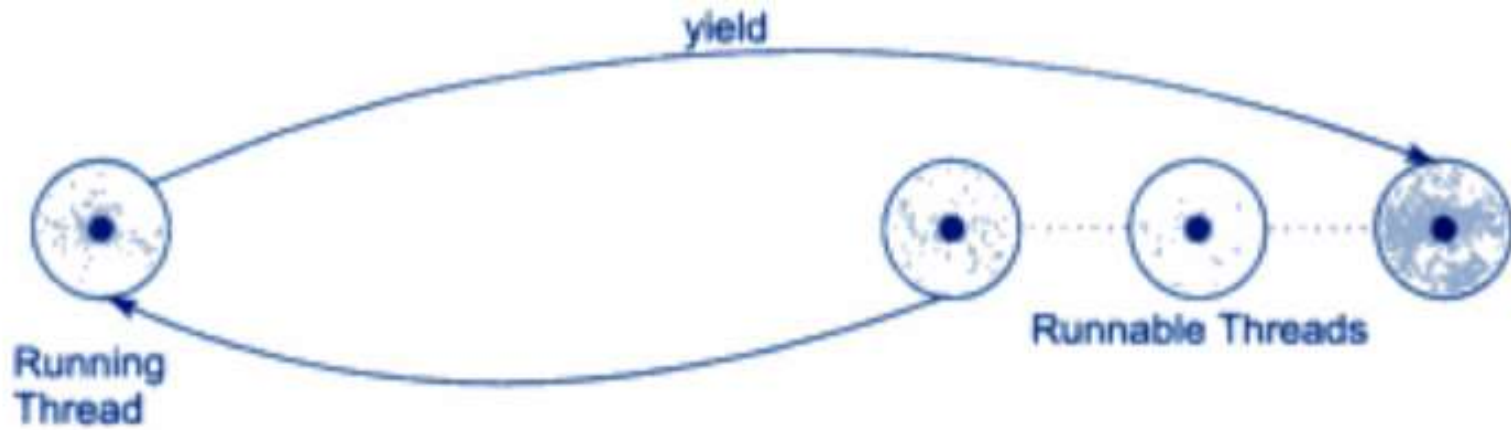
State transition diagram of a thread



New born state

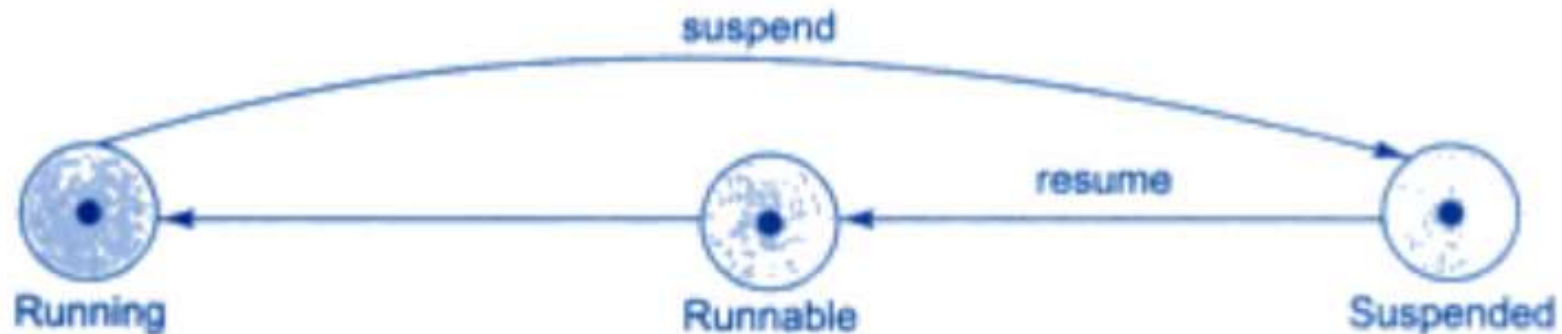


Runnable State

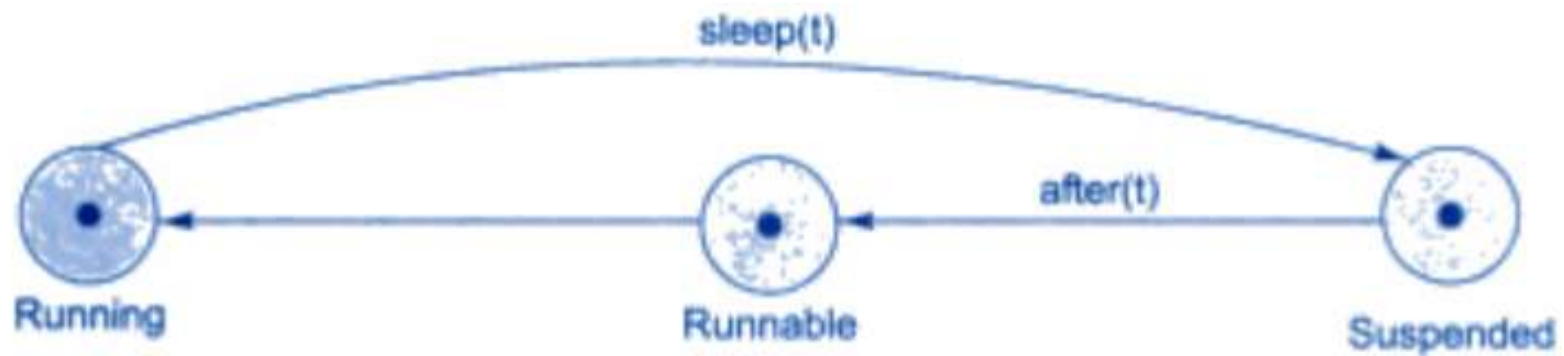


Running state

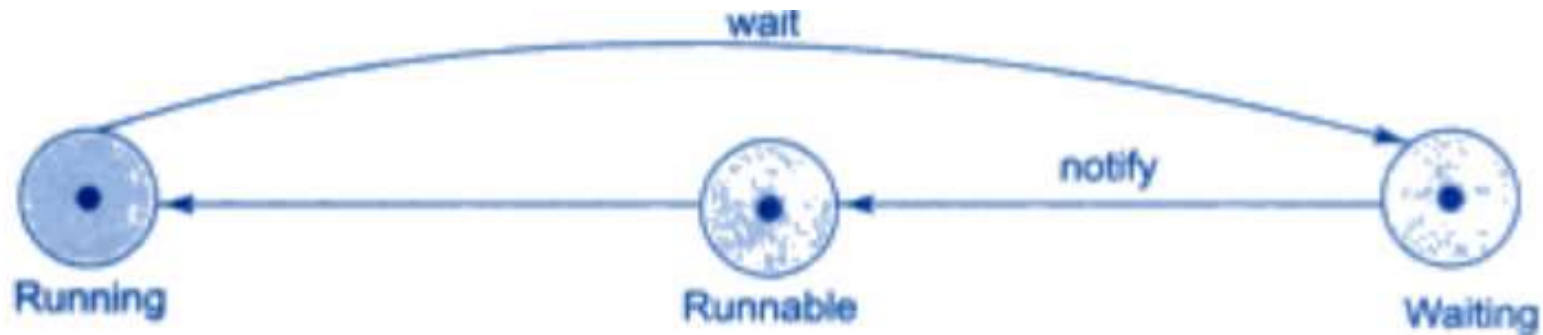
- The running thread may relinquish its control in one of the following situation



Relinquishing control using suspend()



Relinquishing control using wait() method



Blocked state and Dead state

- A thread is said to be blocked
 - When it is prevented from entering into the runnable state and the running state
- Dead state
 - A running thread ends its life when it has completed executing its `run()` method
 - It is a natural death

Thank you