JAVA

Contents

Contents

Day 1

Chapter 1

The Evolution of Java

Objectives

At the end of this session, you will be able to:

- Describe the evolution of Java, and the forces that shaped it
- Describe the basics of a Java Program
- Describe data types, variables and control structures
- Define strong typing
- Define function overloading
- Describe the new operator
- Describe Garbage Collection
- Define references

Evolution of Java

- Smalltalk was the first object-oriented language
- **Smalltalk** is a pure object-oriented programming language
- **Smalltalk** had a major influence on the design of Java

Evolution of Java

- Java was the brainchild of:
  - James Gosling
  - Patrick Naughton
  - Chris Warth
  - Ed Frank &
  - Frank Sheridan

- Made its appearance in the fall of 1992, and was initially called **Oak**

- Was renamed **Java** in **1995**

Evolution of Java

- Was originally meant to be a platform-neutral language for embedded software in devices
- The goal was to move away from platform and OS-specific compilers
- The language could be used to produce platform-neutral code

Evolution of Java

- The catalytic role of the World Wide Web in the evolution of Java
- But for the WWW, the scope of Java would have been restricted to programming devices
- The emergence of the WWW propelled the rise of Java because the WWW demanded portability

Evolution of Java

- Internet and the World Wide Web presented a heterogeneous platform of diverse CPU architectures and OSs
- Need for a portable language gathered increasing pace
- The scope of the language therefore widened from embedded controllers to the Internet.

Java and the Internet

- Needs of the Internet influencing Java
- Java, in turn influencing the Internet by rendering it dynamic
- Java made objects move around on the Internet
- Internet typically dealt with passive and active content

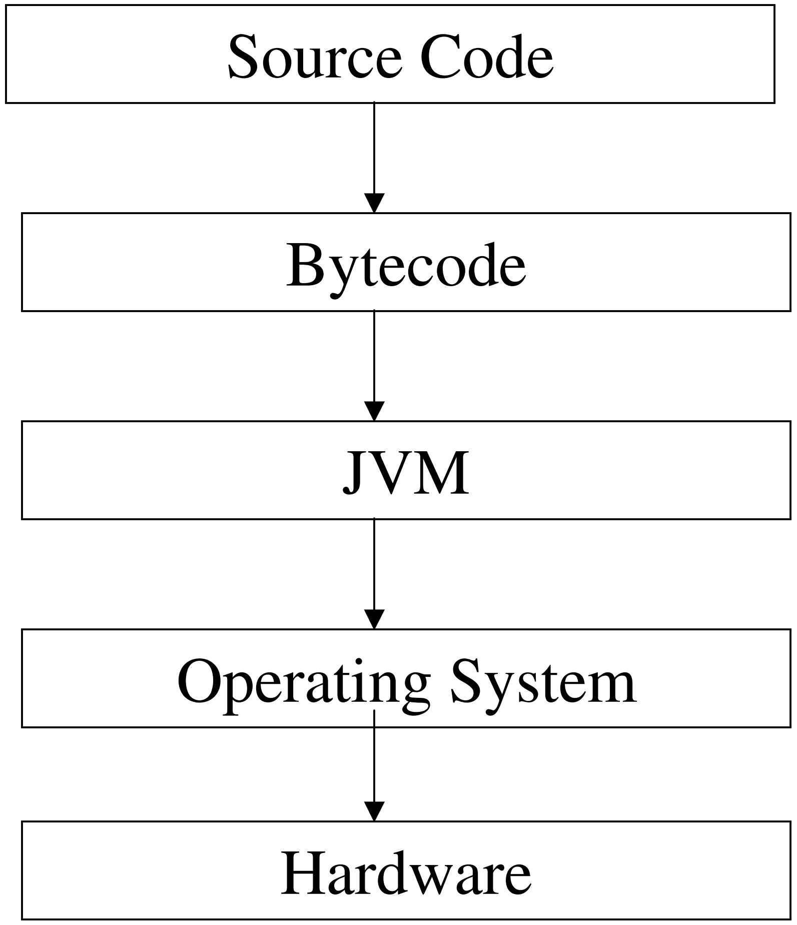
Java: Enabling a Dynamic Internet

- Java made the Internet the host of dynamic self-executing programs
- Such a program is an active agent on the client computer, yet it is initiated by the server
- The scope of a computing platform therefore widened from a client-server network to the Internet and the WWW
- Thus reinforced Sun Microsystems' claim of "**The Network is the Computer**"

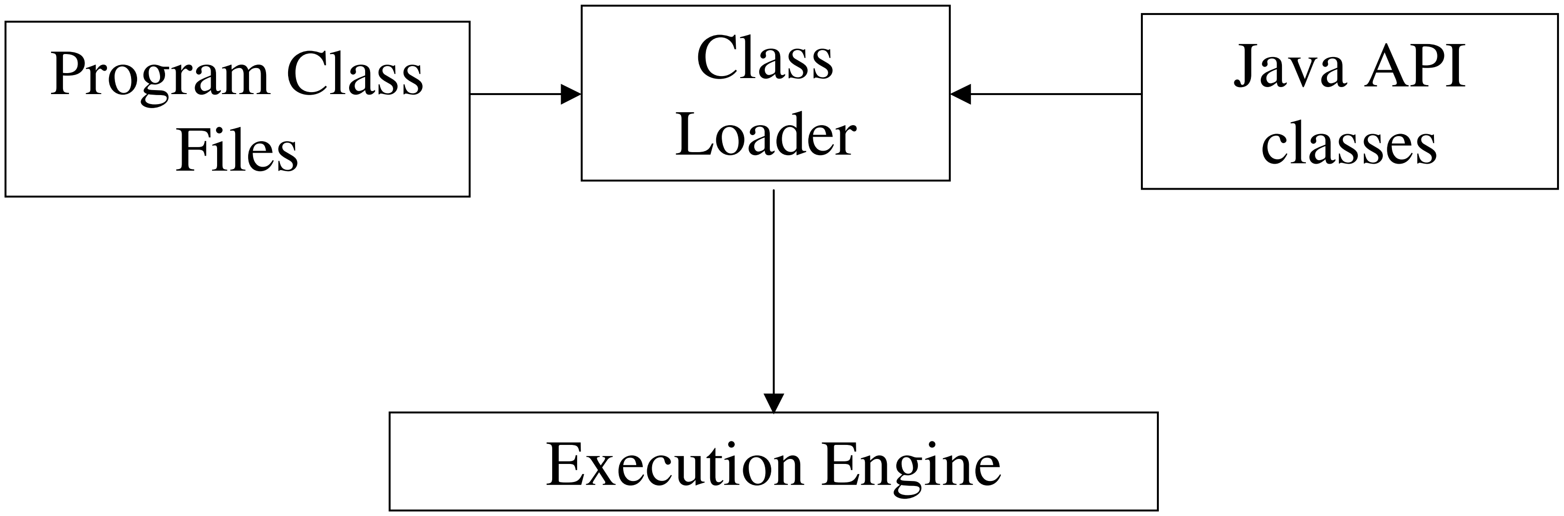
The Java Architecture

- Bytecode is the key to both security & portability
- Write source code.
- Compile to bytecode contained in .class files
- **Bytecode is a highly optimized set of instructions**
- **Bytecode executed by the Java Virtual Machine (JVM)**
- **The JVM is an interpreter for bytecode.**

The Java Architecture



The Java Architecture – The JVM



The Java Architecture – The JVM

- Most modern languages are designed to be compiled
- Compilation is a one-time exercise and executes faster
- Execution of compiled code over the Internet an impossibility
- Executable code always generated to a CPU-OS combination
- Interpreting a Java program into bytecode facilitates its execution in a wide variety of environments

The Java Architecture – The JVM

- Only the Java Virtual Machine (JVM) needs to be implemented for each platform
- Once the Java runtime package exists for a given system, any Java program can run on it.
- **The JVM will differ from platform to platform, and is, platform-specific**
- **All versions of JVM interpret the same Java bytecode.**

The Java Architecture – The JVM

- Interpreted code runs much slower compared to executable code
- The use of bytecode enables the Java runtime system to execute programs much faster
- Java facilitates on-the-fly compilation of bytecode into native code

The Java Architecture – The Adaptive Optimizer

- Another type of execution engine is an adaptive optimizer
- The virtual machine starts by interpreting bytecodes
- Monitors the activity of the running program
- Identifies the most heavily used areas of code
- As the program runs, the virtual machine compiles bytecodes to native code
- Optimizes just these heavily used areas
- The rest of the code, which is not heavily used, remain as bytecodes which the virtual machine continues to interpret.

The Java Architecture - The Class Loader

- The class loader is that part of the VM that is important from:
  - A security standpoint
  - Network mobility

- The class loader loads a compiled Java source file (**.class** files represented as bytecode) into the Java Virtual Machine (JVM)

- The bootstrap class loader loads classes, including the classes of the Java API, in some default way from the local disk

The Java Architecture - The Java .class file

- The Java class file helps make Java suitable for networks mainly in the areas of:
  - platform independence
  - network mobility

- The class file is compiled to a target JVM, but independent of underlying host platforms

- The Java class file is a binary file that can run on any hardware platform and OS that hosts the Java VM

The Java API

- The Java API files provide a Java program with a
  - standard
  - platform-independent interface to the underlying host

- To the Java program, the API looks the same and behaves predictably irrespective of the underlying platform

- The internal design of the Java API is also geared toward platform independence

The Java Programming Language

- The Java language allows you to write programs that take advantage of many software technologies:
  - object-orientation
  - multithreading
  - structured error-handling
  - garbage collection
  - dynamic linking
  - dynamic extension

The Java Buzzwords

- Simple
  - Small language [large libraries]
  - Small interpreter (40 k), but large runtime libraries (175 k)
- Object-Oriented
  - Supports encapsulation, inheritance, abstraction, and polymorphism.
- Distributed
  - Libraries for network programming
  - Remote Method Invocation
- Architecture neutral
  - Java Bytecodes are interpreted by the JVM.

The Java Buzzwords

- Secure
  - Difficult to break Java security mechanisms
  - Java Bytecode verification
  - Signed Applets.
- Portable
  - Primitive data type sizes and their arithmetic behavior specified by the language
  - Libraries define portable interfaces
- Multithreaded
  - Threads are easy to create and use
- Dynamic
  - Finding Runtime Type Information is easy

Functions

- Functions are the building blocks of a Java program
- **main( )** is the first function to be executed
- format

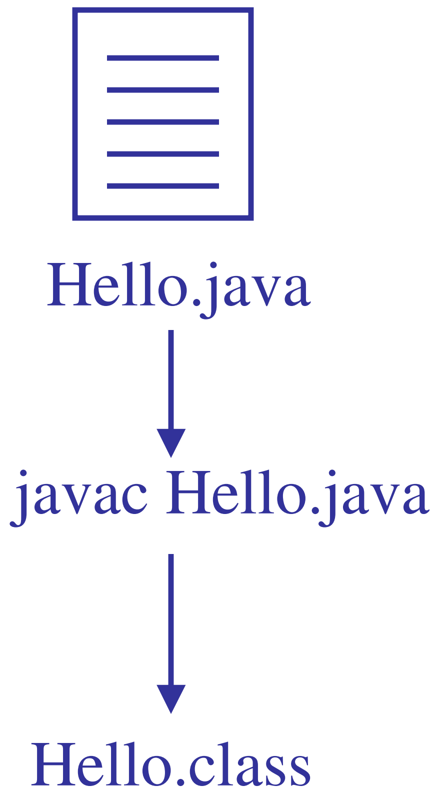
  return_type function_name(argument list)
- Prototyping required but variable name not necessary

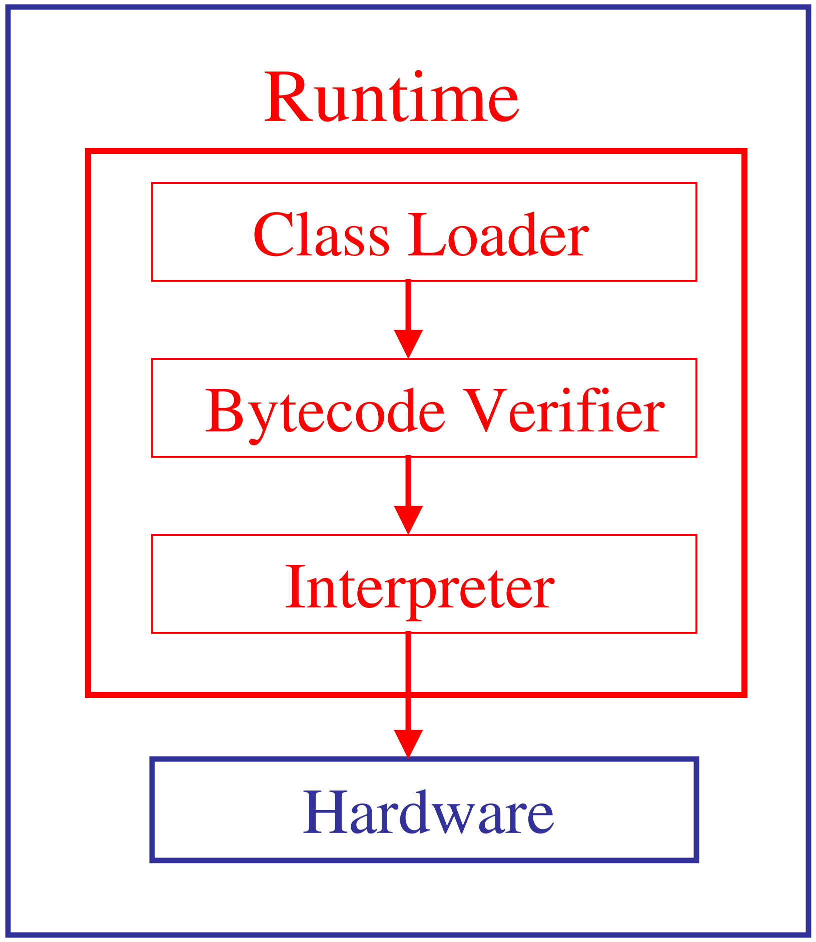
A First Java Program

```
public class Hello      {
    public static void main(String [] args){
        System.out.println(“Hello World”);
    }
}
```

- Compilation - javac Hello.java
- Execution - java Hello

A First Java Program





Language Basics

- Keywords
- Variables
- Conditional Statements
- Loops
- Data Types
- Operators
- Coding Conventions

Data Types

- Integer
  - int 32 bits
  - long 64 bits
  - short 16 bits
  - byte 8 bits
- Character
  - char 16 bits, UNICODE Character
  - String (is an object in Java)
- Floating Point
  - float 32 bits
  - double 64 bits
- Boolean
  - true false

Range of Values for Integers

Name / Type	Length	Range
byte	8 bits	-2^7 to 2^7-1
short	16 bits	-2^{15} to $2^{15} -1$
int	32 bits	-2^{31} to $2^{31} -1$
long	64 bits	-2^{63} to $2^{63} -1$

Floating Point - float and double

- Floating point literal includes either a decimal point or one of the following:
  - E or e (add exponential value)
  - F or f (float)
  - D or d (double)
    - 3.14 A simple floating-point value (a double)
    - 6.02E23 A large floating-point value
    - 2.718F A simple float size value
    - 123.4E+306D A large double value with redundant

Declaring Arrays

- Group data objects of the same type. Declare arrays of primitive or class types.
  - char s[]; or char [] s;
  - Point p[]; or Point [] p;
- Create space for reference. An array is an object, not memory reserved for primitive types.
- Use the *new* keyword to create an array object.
  - s = new char[20];
  - p = new Point[100];

```
p[0] = new Point();
p[1] = new Point();
```

Array Initialization & Multidimensional Arrays

- Create an array with initial values

```
String names[] = new String[3];
names [0] = “Jack”;
names [1] = “Jill”;
names [2] = “Tom”;
MyClass array[] = {new MyClass(), new MyClass()};
```

- Arrays of arrays :
  - int twoDim [] [] = new int [4] [];
  - twoDim [0] = new int [5];
  - twoDim [1] = new int [5];
  - int twoDim [] [] = new int [] [5];// illegeal

Array Bounds, Array Resizing, & Array Copying

- All array subscripts begin with 0:
- int list [] = new int [10]
- for (int i = 0; i< list.length; i++) **System.out.println(list[i]);**

- Can not resize an array. Can use the same reference variable to refer to an entirely new array
  - int elements [] = new int [5];
  - elements = new int [10];

- The System.arraycopy() method:
  - int elements [] = {1, 2, 3, 4, 5, 6};
  - int hold [] = {10,9,8,7,6,5,4,3,2,1};
  - System.arraycopy(elements,0,hold,0,elements.length);

String

- Is not a primitive data type; it is a class
- Has its literal enclosed in double quotes (" ")
  - “A stitch in time saves nine"
- Can be used as follows:
  - String greeting = "Good Morning !! \n";
  - String errorMessage = "Record Not Found !";

Strong Typing

- Java is a strongly typed language
- The Java compiler enforces type checking of each assignment made in a program at compile time
- Code is type checked
- If the type checking fails, then a compile- time error is issued
- Strong typing facilitates two functions of the same name to coexist
- Distinguishes between calls to these functions based on their signatures

Function Signature & Overloading

- A function signature constitutes the function name, the number, order and the data type of the arguments
- Such functions are said to be **overloaded**.
- Calls to an overloaded function are resolved using function signatures
- Overloaded functions are differentiated based on their signature

The final Keyword

- Mechanisms of telling the compiler that a piece of data is “constant”
- A constant is useful for two reasons:
  - It can be a compile-time constant that will never change
  - It can be a value initialized at runtime that you don’t want changed.
- A variable can be declared as **final**
- Doing so prevents its contents from being modified

Final Variables

- Implies that you must initialize a **final** variable when it is declared
  - **final float PI = 3.142857;**

- Subsequent parts of your program can now use **PI** as a constant

- **Thus a final variable is essentially a named constant.**

Object References

- Obtaining objects of a class is a two-step process.
  - First you must declare a variable of the class type
  - This variable does not define an object.
  - Instead, it is simply a variable that can refer to an object.
  - Or, simply an **object reference**.

  - Second, you must acquire an actual, physical copy of the object
  - Assign the address or reference of that object to that variable
  - You can do this using the **new** operator.

Object References

- The **new** operator
  - dynamically allocates memory for an object
  - creates the object on the heap or free store
  - returns a reference to it
  - This reference is, the address of the object allocated by **new.** The reference is then stored in the variable

- **Thus, in Java**
  - **all objects must be dynamically allocated**
  - stored on the **heap**

Summary

At the end of this session, you learnt to:

- Describe the evolution of Java, and the forces that shaped it
- Describe the basics of a Java Program
- Describe data types, variables and control structures
- Define strong typing
- Define function overloading
- Describe the new operator
- Describe Garbage Collection
- Define references

CHAPTER 2

Classes and Objects

Objectives

- At the end of this session, you will be able to:
- Define a class
- Describe the access modifiers, namely public and private
- Create objects based on classes
- Define constructors
- Describe the '**this**' reference
- Define **static** class members

Defining a Sample point Class

- class Point {
-   int x;   int y;
-   void setX(int x)
-   { x = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
-       void setY (int y)
-   { y = (y < 24 ? 24 : (y < 0 ? 0 : y)); }
-       int getX( )
-   { return x; }
-       int getY( )
-   { return y;}
- }

Access Specifiers

- Java provides access specifiers to control access to class members
- Access specifiers help implement:
  - Encapsulation by hiding implementation-level details in a class
  - Abstraction by exposing only the interface of the class to the external world
- The **private** access specifier is generally used to encapsulate or hide the member data in the class
- The **public** access specifier is used to expose the member functions as interfaces to the outside world

Class Declaration for Point

- class Point{
- private int x;
- private int y;

- public void setX(int x)
- {
- x= (x > 79 ? 79 : (x < 0 ? 0 :x));
- }

- public void setY (int y)
- {
- y= (y < 24 ? 24 : (y < 0 ? 0 : y));
- }

- public int getX( )
- {
- return x;
- }

Class Declaration for Point

- public int getY( )
- {
- return y;
- }

- public static void main(String args[ ])
- {
- int a, b;
- Point p1 = new Point( );
- p1.setX(22);
- p1.setY(44);
- a = p1.getX( );
- System.out.println("The value of a is"+a);
- b = p1.getY( );
- System.out.println("The value of b is"+b);
- }
- }

Initializing Member Data in an Object of a Class

- Every time an object is created, memory is allocated for it
- This does not automatically ensure proper initialization of member data within the object
- While designing a class, the class designer can define within the class, a special method called ‘*constructor*’
- This *constructor* is automatically invoked whenever an object of the class is created

Constructors

- A constructor function has the same name as the class name
- A class can contain more than one constructor
- Facilitates multiple ways of initializing an object
- Constructors can therefore be overloaded.

Class Declaration for Point

- class Point
- {
- private int x;
- private int y;
- public Point( )
- {
- x = y = 0;
- }

- public Point(int x, int y)
- {
- x = (x > 79 ? 79 : (x < 0 ? 0 :x));
- y = (y < 24 ? 24 : (y < 0 ? 0 : y));
- }

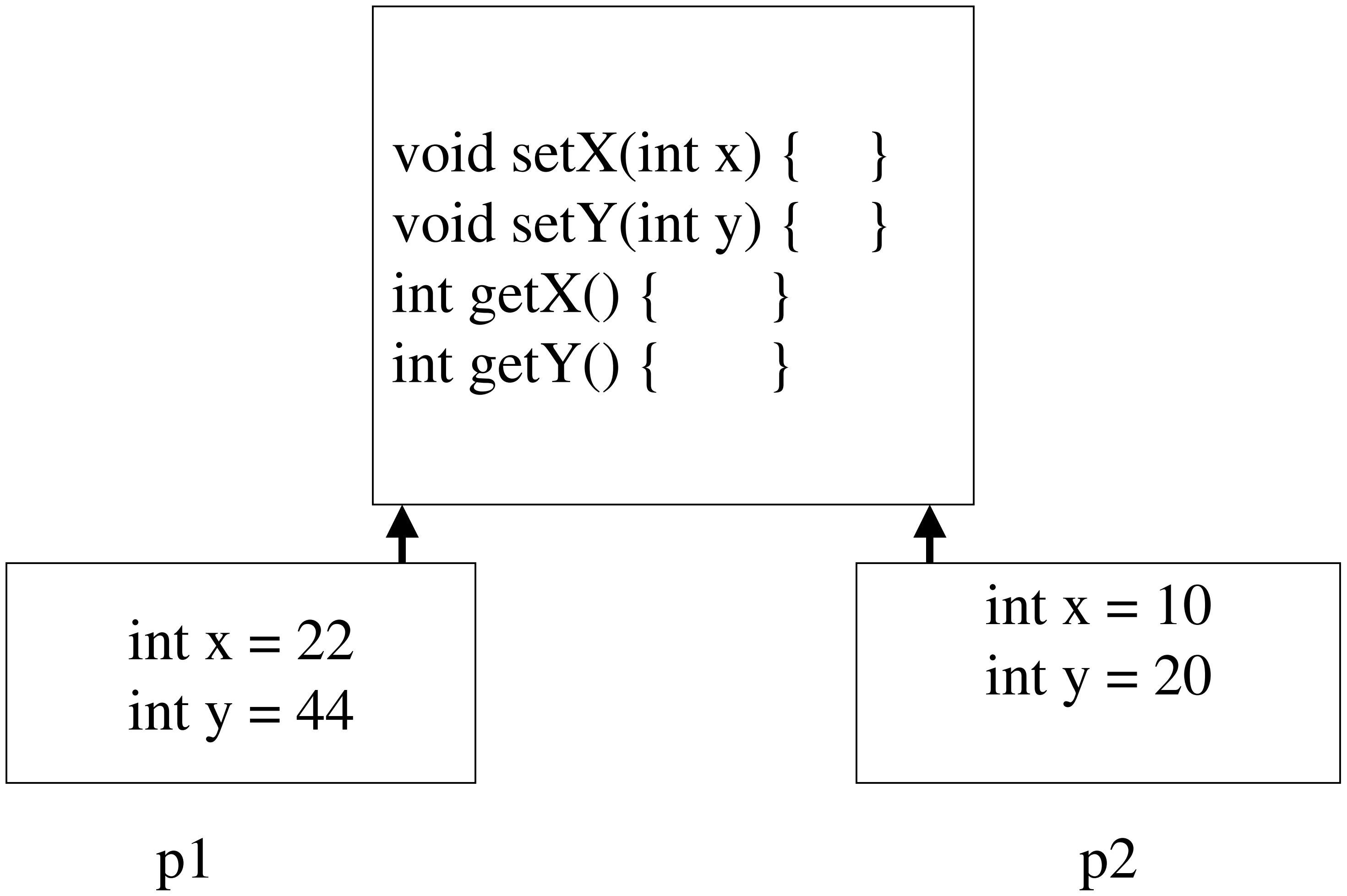
Class Declaration for Point

- public int getX()
- {
- return x;
- }
- public int getY()
- {
- return y;
- }
- }
- class PointDemo{
- public static void main(String args[ ])
- {
- int a, b,c,d;
- Point p1 = new Point( );
- Point p2 = new Point(22, 44);
- a = p1.getX( );
- b = p1.getY( );
- c = p1.getX( );
- d = p1.getY( );

Class Declaration for Point

- System.out.println("The value of xcoord of p1 is :" + a);
- System.out.println("The value of ycoord of p1 is :" + b);
- System.out.println("The value of xcoord of p2 is :" + c);
- System.out.println("The value of ycoord of p2 is :" + d);
- }
- }

The this Reference



The this Reference

- Each class member function contains an implicit reference of its class type, named **this.**

- The **this** reference, created automatically by the compiler

- Contains the address of the object through which the function is invoked

Static Class Members – Static Data Members

- Class members can be made **static**
- Static members are shared by all objects of a class

```
class StaticDemo
{
private static int a = 0;
private int b;
public void set ( int i, int j)
{
a = i; b = j;
}

public void show( )
{
System.out.println("this is static a: " + a );
System.out.println( "this is non-static b: " + b );
}
public static void main(String args[ ])
{
StaticDemo x = new StaticDemo( );
StaticDemo y = new StaticDemo( );
x.set(1, 1);
x.show( );
y.set(2, 2);
y.show( );
x.show( );
}
}
```

Java’s Cleanup Mechanism – The Garbage Collector

- Objects on the heap must be deallocated or destroyed, and their memory released for later reallocation
- Java handles object deallocation automatically through **garbage collection**
- Objects without references will be reclaimed.

Day 2

Inheritance

Objectives

At the end of this session, you will be able to:

- Describe Java's inheritance model and its language syntax
- State how a reference variable of a superclass can reference objects of its subclasses
- Describe the usage of the keyword **super**
- Define a multilevel hierarchy
- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Describe **abstract** classes
- Regulate extension of class hierarchies
- Describe the **Object** class

Inheritance

- Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications
- Using inheritance, you can create a general class at the top
- This class may then be inherited by other, more specific classes
- Each of these classes will add only those attributes and behaviours that are unique to it

Generalization/Specialization

- In keeping with Java terminology, a class that is inherited is referred to as a **superclass**.
- The class that does the inheriting is referred to as the **subclass**.
- **Each instance of a subclass includes all the members of the superclass.**
- **The subclass inherits all the properties of its superclass.**

Java’s Inheritance Model

- Java uses the single inheritance model
- **In single inheritance, a subclass can inherit from one and one only one superclass**
- **Code Syntax for Inheritance: class derived-class-name extends base-class-name**

Inheritance – An Example

- class A{
- int m, n;
- void display1( ){
- System.out.println("m and n are:"+m+" "+n);
- }}
- class B extends A{
- int c;
- void display2( ){
- System.out.println("c :" + c);
- }
- void sum()
- {
- System.out.println("m+n+c = " + (m+n+c));
- }
- }

Inheritance – An Example

- class InheritanceDemo{
- public static void main(String args[ ]){
- A s1 = new A();
- B s2 = new B();
- s1.m = 10; s1.n = 20;
- System.out.println("State of object A:");
- s1.display1();
- s2.m = 7; s2.n = 8; s2.c = 9;
- System.out.println("State of object B:");
- s2.display1();
- s2.display2();
- System.out.println("sum of m, n and c in object B is:");
- s2.sum();
- }}

Accessing Superclass Members from a Subclass Object

- A subclass includes all of the members of its superclass
- But, it cannot directly access those members of the superclass that have been declared as **private**.
- class A
- {
- int money;
- private int pocketMoney;
- void fill(int money, int pocketMoney)
- {
- this.money = money;
- this.pocketMoney = pocketMoney;
- }
- }

Accessing Superclass Members from a Subclass Object

```
class B extends A{
int total;
void sum( ){
total = money + pocketMoney;
}}
class AccessDemo{
public static void main(String args[ ]){
B subob = new B();
subob.fill(10,12);
subob.sum();
System.out.println("Total: " + subob.total);
}
}
```

A Possible Solution To The Program

```
class A
{
int money;
private int pocketMoney;

void fill(int money, int pocketMoney)
{
this.money = money;
this.pocketMoney = pocketMoney;
}
public int getPocketMoney(){
return pocketMoney;
}
}
```

A Possible Solution To The Program

- class B extends A{
- int total;
- void sum( ){
- total = money + getPocketMoney();
- }}

- class AccessDemo{
- public static void main(String args[ ]){
- B subob = new B();
- subob.fill(10,12);
- subob.sum();
- System.out.println("Total: " + subob.total);
- }}

A Practical Example

- You will now evolve a class description for a closed rectangular solid. It has attributes width, height, and depth
- Using these attributes, one can compute its volume
- The aforesaid points lead you to the definition of a RectangularSolid class

```
class RectangularSolid
{
double width;
double height;
double depth;
```

A Practical Example

```
RectangularSolid(double w, double h, double d)
{
width  = w;
height = h;
depth  = d;
}
RectangularSolid(double length)
{
width = height = depth = length;
}

RectangularSolid()
{
width = height = depth = 10;
}

double volume()
{
return width *height * depth;
}
}
```

A Practical Example

```
 class RectangularSolidWeight extends RectangularSolid
{
double weight;

public RectangularSolidWeight(double w, double h, double d, double wt)
{
width  = w;
height = h;
depth  = d;
weight = wt;
}
}

class RectangularSolidImpl
{
```

```
public static void main(String args[])
{
RectangularSolidWeight r1 = new RectangularSolidWeight(10,20,15,35.0);
RectangularSolidWeight r2 = new RectangularSolidWeight(1,2,3,5.0);
double volume;
volume = r1.volume();
System.out.println("Volume of rectangular solid r1 is " + volume);
System.out.println();//print a blank line
volume = r2.volume();
System.out.println("Volume of rectangular solid r2 is " + volume);
}
}
```

A Superclass Reference Variable Can Reference a Subclass Object

- A reference variable of type superclass can be assigned a reference to any subclass object derived from that superclass

```
class SuperRefDemo
 {
  public static void main(String args[])
  {
  RectangularSolidWeight rsw = new RectangularSolidWeight(4,5,7,8.0);
  RectangularSolid rs = new RectangularSolid();
  double volume;
  volume = rsw.volume();
  System.out.println("volume of rsw is :"+volume);
  System.out.println("Weight of rsw is :" +rsw.weight);
```

A Superclass Reference Variable Can Reference a Subclass Object

```
 rs = rsw;
 volume = rs.volume();
 System.out.println("Volume of  RectangularSolidWeight is :" + volume);
 System.out.println("Weight of  RectangularSolid is :" + rs.weight);
 rs = rsw;
 volume = rs.volume();
}
}
```

A Superclass Reference Variable Can Reference a Subclass Object

- **Method calls in Java are resolved dynamically at runtime**
- **In Java all variables know their dynamic type**
- **Messages (method calls) are always bound to methods on the basis of the dynamic type of the receiver**
- **This method resolution is done dynamically at runtime**

Using super

- The class **RectangularSolidWeight** derived from the class **RectangularSolid** was not implemented efficiently

- The constructor for the subclass **RectangularSolidWeight** explicitly initializes the **width, height** and **depth** attributes of the superclass **RectangularSolid**

Using super

- It is an important principle of class design that every class should keep the details of its implementation to itself

- One aspect is to keep its data **private** to itself.

- This should hold true even in the case of a superclass-subclass relationship in a class hierarchy

- The subclass should not have access to the implementation details of its superclass

Using super

- Therefore, data members should be private in a superclass
- Otherwise, encapsulation can be broken at will by just creating a dummy subclass and accessing all the superclass data members
- The aforesaid conclusion apparently seems to give rise to another problem
- Consider a situation where you straight away and independently create an object of the subclass without first creating an object of the superclass

- The creation and initialization of the superclass object is a prerequisite to the creation of the subclass object.

- The logical sequence of steps therefore should be:
  - When a subclass object is directly created, it
  - first results in the creation of the underlying superclass object
  - the initialization of its attributes that is ensured by the invocation of a relevant superclass constructor.
  - The initialized superclass attributes are then inherited by the subclass object
  - finally followed by the creation of the subclass object
  - initialization of its own specific attributes through a relevant constructor invocation in the subclass

Using super

- The composition of the subclass constructor now merits your attention
- It is vital that a subclass object inherit initialized attributes from the superclass object
- Therefore, the first statement of the subclass constructor must:
  - result in an invocation of the constructor of its immediate superclass and
  - thereby initializing the attributes of the superclass object prior to being inherited by the subclass

Using super

- But the subclass constructor cannot access the private attributes of the superclass

- Moreover, constructors of the superclass are never inherited by the subclass

- This is the only exception to the rule that a subclass inherits all the properties of its superclass

- So, that leaves the subclass with neither access to the private members of the superclass, nor access to the superclass constructors.

Using super

- The key to this problem is the **super** keyword. **super** has two uses.
  - The first is the call to the superclass' constructor from the subclass constructor
  - The second usage is to access a member of the superclass that has been overridden by a subclass

- A subclass constructor can call a constructor defined in its immediate superclass by employing the following form of **super:**
  - super(parameter-list);

Using super to Call Superclass Constructors

- super( ) must always be the first statement executed inside a subclass’ constructor.
- It clearly tells you the order of invocation of constructors in a class hierarchy.
- **Constructors are invoked in the order of their derivation**
- A more efficient and robust definition of the superclass **RectangularSolid** class and the corresponding changes in the subclass **RectangularSolidWeight** follows:

Using super to Call Superclass Constructors

```
class RectangularSolid
{
  private double width;
  private double height;
  private double depth;

RectangularSolid(double w, double h, double d)
{
  width  = w;
  height = h;
  depth  = d;
}
RectangularSolid(double length)
{
  width = height = depth = length;
}

RectangularSolid()
{
 width = height = depth = 10;
}
```

Using super to Call Superclass Constructors

```
double volume()
{
 return width *height * depth;
}
}

class RectangularSolidWeight extends RectangularSolid
{
double weight;

public RectangularSolidWeight(double w, double h, double d, double wt)
{
 super(w,h,d);
 weight = wt;
}
```

Using super to Call Superclass Constructors

```
 public RectangularSolidWeight(double len, double  wt)
{
 super(len);
 weight = wt;
}

public RectangularSolidWeight( )
{
 super();
 weight = 0;
}
}
class SuperDemo
{
  public static void main(String args[])
   {
   RectangularSolidWeight rsw1 = new  RectangularSolidWeight(10,20,15,34.3);
   RectangularSolidWeight rsw2 = new  RectangularSolidWeight();
   RectangularSolidWeight rswcube = new RectangularSolidWeight(3,2);
   double volume;
   volume = rsw1.volume();
```

Using super to Call Superclass Constructors

```
    System.out.println("Volume of rsw1 is :" +    volume);
    System.out.println("Weight of rsw1 is :" +  rsw1.weight);
    System.out.println();
    volume = rsw2.volume();
    System.out.println("Volume of rsw2 is :" +  volume);
    System.out.println("Weight of rsw2 is :" + rsw2.weight);
    System.out.println();
    volume = rswcube.volume();
    System.out.println("Volume of rsw2 is :" +  volume);
    System.out.println("Weight of rswcube is :" +  rswcube.weight);
    System.out.println();
    }
  }
```

Using super to Call Superclass Constructors

- When a subclass calls **super( )**, it is calling the constructor of its immediate superclass
- This is true even in a multileveled hierarchy
- **super( )** must always be the first statement inside a subclass constructor.

Order of Invocation of Constructors in a Class Hierarchy

- ***Constructors in a class hierarchy are invoked in the order of their derivation.***
- class X {
- X( )
-   { System.out.println("Inside X's Constructor"); } }
-
- class Y extends X {
- Y( )
-   { System.out.println("Inside Y's Constructor"); } }
- class Z extends Y {
- Z()
-   { System.out.println("Inside Z's Constructor"); } }
- class OrderOfConstructorCallDemo
- {
-   public static void main(String args[])
-    {
-     Z z = new Z(); } }

Defining a Multilevel Hierarchy

- Java allows you to define multiple layers in an inheritance hierarchy
- You can define a superclass and a subclass, with the subclass in turn becoming a superclass for another subclass.
- You will now extend the **RectangularSolidWeight** class further to derive a new class called **RectangularSolidShipment** class.
- **RectangularSolidShipment** inherits all of the traits of **RectangularSolid** and **RectangularSolidWeigh**t classes, and adds its own specific attribute called **cost**

Defining a Multilevel Hierarchy

```
class RectangularSolid
{
private double width;
private double height;
private double depth;

RectangularSolid(double w, double h, double d)
{
width  = w;
height = h;
depth  = d;
}

RectangularSolid(double length)
{
width = height = depth = length;
}
```

Defining a Multilevel Hierarchy

```
RectangularSolid()
{
width = height = depth = 10;
}

double volume()
{
return width *height * depth;
}
}
class RectangularSolidWeight extends RectangularSolid
{
double weight;

public RectangularSolidWeight(double w, double  h, double d, double wt)
{
super(w,h,d);
weight = wt;
}
```

Defining a Multilevel Hierarchy

```
public RectangularSolidWeight(double len, double  wt)
{
super(len);
weight = wt;
}
public RectangularSolidWeight( )
{
super();
weight = 10;
}
}

class RectangularSolidShipment extends RectangularSolidWeight
{
double cost;

RectangularSolidShipment(double w, double h, double d, double  wt, double c)
{
super(w,h,d,wt);
cost = c; }
```

Defining a Multilevel Hierarchy

```
RectangularSolidShipment()
{
super();
cost = 10;
}

RectangularSolidShipment(double len, double wt, double c)
{
super(len,wt);
cost = c;
}
}

class MultilevelDemo
{
```

Defining a Multilevel Hierarchy

```
public static void main(String args[])
{
RectangularSolidShipment s1 = new RectangularSolidShipment(10,20,15,10,3.41);
RectangularSolidShipment s2 = new RectangularSolidShipment(2,3,4,0.76,1.28);
double volume;
volume = s1.volume();
System.out.println("Volume of Shipment s1 is     :" + volume);
System.out.println("Weight of Shipment s1 is     :" + s1.weight);
System.out.println("Shipping Cost of shipment  s1 is " + s1.cost);
System.out.println();
volume = s2.volume();
System.out.println("Volume of Shipment s2 is     :" + volume);
System.out.println("Weight of Shipment s2 is     :" + s2.weight);
System.out.println("Shipping Cost of shipment     S2 is " + s2.cost);
System.out.println();
}
}
```

Method Overriding

- When a method in a subclass has the same prototype as a method in the superclass, then the method in the subclass is said to override the method in the superclass

- When an overridden method is called from an object of the subclass, it will always refer to the version of that method defined by the subclass

- The version of the method defined by the superclass is hidden or overridden

```
 class A
{
int a,b;
A(int m, int n)
{
a = m;
b = n;
}
```

Method Overriding

```
void display()
{
System.out.println("a and b are :" + a +" " +     b);
}
}

class B extends A
{
int c;
B(int m, int n, int o)
{
super(m,n);
c = o;
}

void display()
{
System.out.println("c :" + c);
}
}
```

Method Overriding

```
class OverrideDemo
{
public static void main(String args[])
{
B subOb = new B(4,5,6);
subOb.display();
}
}
```

Using super to Call an Overridden Method

```
class A
{
int a,b;

A(int m, int n)
{
a = m;
b = n;
}

void display()
{
System.out.println("a and b are :" + a +" " +     b);
}
}

class B extends A
{
int c;
```

Using super to Call an Overridden Method

- B(int m, int n, int o)
- {
- super(m,n);
- c = o;
- }
- void display()
- {
- super.display();
- System.out.println("c :" + c);
- }
- }

- class OverrideDemo
- {
- public static void main(String args[])
- {
- B subOb = new B(4,5,6);
- subOb.display();
- }
- }

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding forms the basis of one of Java's most powerful concepts: **dynamic method dispatch**

- Dynamic method dispatch occurs when the Java language resolves a call to an overridden method at runtime, and, in turn, implements runtime polymorphism

- **Java makes runtime polymorphism possible in a class hierarchy with the help of two of its features:**
  - **superclass reference variables**
  - **overridden methods**

Dynamic Method Dispatch or Runtime Polymorphism

- A superclass reference variable can hold a reference to a subclass object
- Java uses this fact to resolve calls to overridden methods at runtime
- **When an overridden method is called through a superclass reference, Java determines which version of the method to call based upon the type of the object being referred to at the time the call occurs**

Why Overridden Methods? A Design Perspective

- Overridden methods in a class hierarchy is one of the ways that Java implements the "**single interface, multiple implementations**" aspect of polymorphism

- Part of the key to successfully applying polymorphism is understanding the fact that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization

Why Overridden Methods? A Design Perspective

- The superclass provides all elements that a subclass can use directly
- It also declares those methods that the subclass must implement on its own
- This allows the subclass the flexibility to define its own method implementations, yet still enforce a consistent interface
- In other words, the subclass will override the method in the superclass

Overridden Methods and Runtime Polymorphism - An Example

- The following program:
  - Creates a superclass called **Figure** that stores the dimensions of various two dimensional objects
  - **Figure** defines a method called **area( )** that computes the area of an object
  - The program derives two subclasses from **Figure**
  - The two subclasses are **Rectangle** and **Triangle**
  - Each of these subclasses overrides **area( )** so that it returns the area of a **Rectangle** and a **Triangle** respectively

Overridden Methods and Runtime Polymorphism - An Example

```
class Figure
{
double dimension1;
double dimension2;

Figure(double x, double y)
{
dimension1 = x;
dimension2 = y;
}

double area()
{
System.out.println("Area of Figure is undefined");
return 0;
}
}

class Rectangle extends Figure
{
Rectangle(double x, double y)
{
super(x,y);
}
```

Overridden Methods and Runtime Polymorphism - An Example

- double area()
- {
- System.out.print("Area of rectangle is :");
- return dimension1 * dimension2;
- }
- }

- class Triangle extends Figure
- {
- Triangle(double x, double y)
- {
- super(x,y);
- }

- double area()
- {
- System.out.print("Area for rectangle is :");
- return dimension1 * dimension2 / 2;
- }
- }

Overridden Methods and Runtime Polymorphism - An Example

- class FindArea
- {
- public static void main(String args[])
- {
- Figure f   = new Figure(10,10);
- Rectangle r = new Rectangle(9,5);
- Triangle t = new Triangle(10,8);
- Figure figref;
- figref = r;
- System.out.println("Area of rectangle is :" + figref.area());
- figref = t;
- System.out.println("Area of traingle is :" + figref.area());
- figref = f;
- System.out.println(figref.area());
- }
- }

Pure Abstractions - Abstract Classes

- Often, you would want to define a superclass that declares the structure of a given abstraction without providing the implementation of every method

- The objective is to:
  - Create a superclass that only defines a generalized form that will be shared by all of its subclasses
  - leaving it to each subclass to provide for its own specific implementations
  - Such a class determines the nature of the methods that the subclasses ***must implement***
  - Such a superclass is unable to create a meaningful implementation for a method or methods

Pure Abstractions - Abstract Classes

- The class **Figure** in the previous example is such a superclass.
  - **Figure is a pure geometrical abstraction**
  - You have only kinds of figures like **Rectangle**, **Triangle** etc. which actually are subclasses of class **Figure**
  - The class **Figure** has no implementation for the **area( )** method, as there is no way to determine the area of a **Figure**
  - The **Figure** class is therefore a partially defined class with no implementation for the **area( )** method
  - The definition of **area()** is simply a placeholder

Pure Abstractions - Abstract Classes

- The key, therefore, is to have a mechanism to ensure that a subclass does, indeed, override, all necessary methods.

- **This mechanism comes in the shape of the abstract method.**

- Abstract methods in a superclass **must** be overridden by its subclasses as the subclasses cannot use the abstract methods that they inherit

- These methods are sometimes referred to as **subclassers' responsibility** as they have no implementation specified in the superclass

Pure Abstractions - Abstract Classes

- To use an abstract method, use this general form: **abstract type name(parameter-list);**
- **Abstract methods do not have a body**
- **Abstract methods are therefore characterized by the lack of the opening and closing braces that is customary for any other normal method**
- **This is a crucial benchmark for identifying an abstract class**

Pure Abstractions - Abstract Classes

- Any class that contains one or more **abstract** methods must also be declared **abstract**
  - A class is declared as an **abstract** class by preceding the keyword **abstract** before the **class** keyword at the beginning of the class declaration
  - There can be no objects of an abstract class
  - An abstract class, cannot, therefore, be instantiated with the **new** keyword
  - **Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract**
  - **It is perfectly acceptable for an abstract class to implement a concrete method**

Improved Version of the Figure Class Hierarchy

- There is no meaningful concept of **area( )** for an undefined two-dimensional geometrical abstraction such as a **Figure**
- The following version of the program declares **area( )** as abstract inside class **Figure**.
- This implies that class **Figure** be declared **abstract**, and all subclasses derived from class **Figure must override area( )**.

Improved Version of the Figure Class Hierarchy

```
abstract class Figure
{
double dimension1;
double dimension2;

Figure(double x, double y)
{
dimension1 = x;
dimension2 = y;
}

abstract double area( );
}

class Rectangle extends Figure
{
Rectangle(double x, double y)
{
super(x,y);
}
```

Improved Version of the Figure Class Hierarchy

```
double area()
{
System.out.print("Area of rectangle is :");
return dimension1 * dimension2;
}
}

class Triangle extends Figure
{
Triangle(double x, double y)
{
super(x,y);
}

double area()
{
System.out.print("Area for rectangle is :");
return dimension1 * dimension2 / 2;
}
}
```

Improved Version of the Figure Class Hierarchy

```
class FindArea
{
public static void main(String args[])
{
Figure figref;
Rectangle r = new Rectangle(9,5);
Triangle t  = new Triangle(10,8);
figref = r;
System.out.println("Area of rectangle is :" + figref.area());
figref = t;
System.out.println("Area of traingle is :" + figref.area());
}
}
```

The Role of the Keyword final in Inheritance

- The **final** keyword has two important uses in the context of a class hierarchy. These uses are highlighted as follows:

- **Using final to Prevent Overriding**
  - While method overriding is one of the most powerful feature of object oriented design, there may be times when you will want to prevent certain critical methods in a superclass from being overridden by its subclasses.
  - Rather, you would want the subclasses to use the methods as they are defined in the superclass.
  - This can be achieved by declaring such critical methods as **final**.

The Role of the Keyword final in Inheritance

- **Using final to Prevent Inheritance**
  - Sometimes you will want to prevent a class from being inherited.
  - This can be achieved by preceding the class declaration with **final**.
  - Declaring a class as **final** implicitly declares all of its methods as **final** too.
  - It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide concrete and complete implementations.

The Cosmic Class – The Object Class

- Java defines a special class called **Object.**

- All other classes are subclasses of **Object**.

- **Object** is a superclass of all other classes; i.e., Java's own classes, as well as user-defined classes.

- This means that a reference variable of type **Object** can refer to an object of any other class.

The Cosmic Class – The Object Class

- Object defines the following methods, which means that they are available in every object.

Method	Explanation
Object clone()	Create a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is reclaimed from the heap by the garbage collector
final Class getClass()	Obtains the class of an object at runtime
int hashCode	Returns the hash code associated with the invoking object
final void notify()	Resumes execution of a thread waiting on the invoking object
final void notifyAll()	Resumes execution of all waiting threads on the invoking object.
String toString()	Returns a string that describes the object.
final void wait final void wait(long milliseconds) final void wait(long milliseconds, long nanoseconds)	Waits on another thread of execution.

Summary

- In this session, you learnt to:
- Describe Java's inheritance model and the language syntax
- State how a reference variable of a superclass can reference objects of its subclasses
- Describe the usage of the keyword **super**
- Define a multilevel hierarchy
- Describe method overriding
- Describe dynamic method dispatch, or runtime polymorphism
- Describe **abstract** classes
- Regulate extension of class hierarchies
- Describe the **Object** class

Day 3

CHAPTER 3

Packages and Interfaces

Objectives

At the end of this session you will be able to:

- Organize classes into Packages
- Create interfaces
- Describe the design goals of the Java language in providing an interface

Organizing classes into Packages

- Packages are containers for classes and interfaces
- Classes and interfaces are grouped together in containers called **packages**
- To avoid namespace collision, we put the classes into separate containers called **packages**
- Whenever you need to access a class, you access it through its package by prefixing the class with the package name

Need for Packages

- **Packages are containers used to store the classes and interfaces into manageable chunks.**

- Packages also help control the accessibility of your classes

- This is also called visibility control

- Example:
- *package* MyPackage;
- class MyClass {   ....}
- class YourClass{   .....}

Storing the Packages

- Packages are stored as directories
- All the classes you create in the package **MyPack** should be saved in the **directory MyPack**
- First create a directory by the name **MyPack** (packagename)
- **Remember, the case should match exactly**

Understanding CLASSPATH

- What is **CLASSPATH**?
- **CLASSPATH is an environment variable that tells the Java runtime system where the classes are present**

- When a packages is not created, all classes are stored in the default package

- **The default package is stored in the current directory**.

- **The current directory is the default directory for CLASSPATH**.

Understanding CLASSPATH

- When you create your own package for example **MyPack**, all the **.class** files including **MyClass** are saved in the directory **MyPack**.

- In order for a program to find **MyPack**, one of two things must be true:
  - Either the program is executed from a directory immediately above **MyPack**, or
  - **CLASSPATH** must be set to include the path to **MyPack**

A Package Example

```
package empPack;

class EmpClass
{
String empName;
double salary;

EmpClass(String name, double sal)
{
empName = name;
salary = sal;
}

void display()
{
System.out.println(empName + " : $"+salary);
}
}
```

A Package Example

```
class EmpSal
{
public static void main(String args[])
{
EmpClass emp[] = new EmpClass[4];
emp[0] = new EmpClass("A.D Tedd",450.20);
emp[1] = new EmpClass("D.M Scott",725.93);
emp[2] = new EmpClass("B.W TayLor",630.80);
emp[3] = new EmpClass("T.N Blair",545.60);
for (int i=0; i<4; i++)
emp[i].display();
}
}
```

Access Protection

- Packages facilitate access-control
- **Once a class is packaged, its accessibility is controlled by its package**
- That is, whether other classes can access the class in the package depends on the **access specifiers** used in its class declaration
- There are four visibility control mechanisms packages offer:
  - private
  - no-specifier (default access)
  - protected
  - public

Packages & Access Control

Specifier	**Accessibility**
private	Accessible in the same class only
protected	Subclasses and non-subclasses in the same package, and subclasses in other packages
No-specifier (default access)	Subclasses and non-subclasses in the same package
public	Subclasses and non-subclasses in the same package, as well as subclasses and non-subclasses in other packages. In other words, total visibility

Importing Packages

- Java has used the package mechanism extensively to organize classes with similar functionality in one package

- If you want to use these classes in your applications, you can do so by including the following statement at the beginning of your program:
  - *import packagename.classname;*

- *If the packages are nested you should specify the hierarchy.*
  - *import package1.package2.classname;*

Importing Packages

- **The class you want to use must be qualified by its package name**.
- If you want to use several classes from a package, it would be cumbersome to type so many classes qualified by their packages.
- It can be made easy by giving a star(*) at the end of the import statement. For example:
  - **import package1.*;**

Interfaces

What is an Interface?

- **Definition:** An interface is a named collection of method declarations (without implementations)
  - An interface can also include constant declarations
  - Interfaces are an integral part of Java
  - **An interface is a complex data type similar to class in Java**.
  - An interface is syntactically similar to an abstract class
  - An interface is a collection of abstract methods and final variables

What is an Interface?

- A class **implements** an interface using the **implements** clause.
  - An interface defines a protocol of behavior
  - An interface lays the specification of what a class is supposed to do
  - An interface tells the implementing class what behaviours to implement
  - How the behaviour is implemented is the responsibility of each implementing class
  - Any class that implements an interface adheres to the protocol defined by the interface, and in the process, implements the specification laid down by the interface

Interfaces – An Elegant Alternative to Multiple Inheritance

- Why are interfaces required when you have to provide method implementation in the class? Why not write the methods directly in the class?

- The answer is – If you have to use certain methods of class **A** in class **B,** you will extend class **B** from class **A,** provided, class A and class B are strongly related

- If you want to use some methods of class **A** in class B without forcing an inheritance relationship, you determine whether there exists a collaboration between them, and define an association between them.

Interfaces – An Elegant Alternative to Multiple Inheritance

- Defining an interface has the advantage that an interface definition stands apart from any class or class hierarchy
- Moreover, it consists of a specification of behaviour/s in the form of abstract methods
- **This makes it possible for any number of independent classes to implement the interface**
- **Thus, an interface is a means of specifying a consistent specification, the implementation of which can be different across many independent and unrelated classes to suit the respective needs of such classes**

Interfaces – An Elegant Alternative to Multiple Inheritance

- Java does not support multiple inheritance
- This is a constraint in class design, as a class cannot achieve the functionality of two or more classes at a time
- Interfaces help us make up for this loss. A class can implement more than one interface at a time
- Thus, interfaces enable you to create richer classes and at the same time the classes need not be related

Interfaces – An Elegant Alternative to Multiple Inheritance

- You have defined a superclass **A** with attributes and behaviours
- You are defining another class **B** wherein there is a need for some of the functionality of class **A**
- Moreover, class **B** bears a strong semantic relation to class **A** which makes out a case for class **B** to be derived from class **A**
- In other words, there is a *natural inheritance relationship* between class **A** and class **B**

Interfaces – An Elegant Alternative to Multiple Inheritance

- The utility of abstract methods defined in an abstract superclass is restricted to subclass derived from that abstract superclass, which in turn will override these abstract methods

- **On the other hand, abstract methods declared in an interface can be defined by independent classes not necessarily related to one another through a class hierarchy**

- **The "single-interface, multiple implementations" aspect of runtime polymorphism that you observed earlier in a class hierarchy is given a much wider import and context through interfaces**

Defining an Interface

- An interface is syntactically similar to a class. It’s general form is:

```
public interface FirstInterface
{
 int addMethod(int x, int y);
float divMethod(int m, int n);
void display();
int VAR1 = 10;
float VAR2 = 20.65;
 }
```

Implementing Interfaces

- A class implements an interface
- A class can implement more than one interface by giving a comma-separated list of interfaces

```
class MyClass implements FirstInterface
{
public int  addMethod(int a, int b)
{
return(a+b);
}

public float divMethod(int i, int j)
{
return(i/j);
}

public void display()
{
System.out.println(“Varaiable 1 :” +VAR1);
System.out.println(“Varaiable 2 :” +VAR2);
}
}
```

Applying Interfaces

- Software development is a process where constant changes are likely to happen
- There can be changes in requirement, changes in design, changes in implementation
- **Interfaces support change**
- **Programming through interfaces helps create software solutions that are reusable, extensible, and maintainable**

Applying Interfaces

```
interface IntDemo
{
void display();
}

class classOne implements IntDemo
{

void add(int x, int y)
{
System.out.println("The sum is :" +(x+y));
}

public void display()
{
System.out.println("Welcome to Interfaces");
}
}
```

Applying Interfaces

```
class classTwo implements IntDemo
{
void multiply(int i,int j, int k)
{
System.out.println("The result:" +(i*j*k) );
}

public void display()
{
System.out.println("Welcome to Java ");
}
}

class DemoClass
{
public static void main(String args[])
{
classOne c1= new classOne();
c1.add(10,20);
c1.display();
classTwo c2 = new classTwo();
c2.multiply(5,10,15);
c2.display();
}
}
```

Interface References

- When you create objects, you refer them through the class references. For example :
  - ClassOne c1= new classOne(); /* Here, c1 refers to the object of the class classOne. */

- **You can also make the interface variable refer to the objects of the class that implements the interface**

- The exact method will be invoked at run time

- It helps us achieve run-time polymorphism

```
interface IntDemo
{
void display();
}

class classOne implements IntDemo
{
void add(int x, int y)
{
System.out.println("The sum is :" +(x+y));
}

public void display()
{
System.out.println("Class one display method ");
}
}
```

```
class classTwo implements IntDemo
{
void multiply(int i,int j, int k)
{
System.out.println("The result:" +(i*j*k) );
}

public void display()
{
System.out.println("Class two display method" );
}
}

class DemoClass
{
public static void main(String args[])
{
IntDemo c1= new classOne();
c1.display();
c1 = new classTwo();
c1.display();
}
}
```

Extending Interfaces

- Just as classes can be inherited, interfaces can also be inherited
- One interface can extend one or more interfaces using the keyword **extends**
- When you implement an interface that extends another interface, you should provide implementation for all the methods declared within the interface hierarchy

Summary

In this session, you learnt to:

- Organize classes into Packages
- Create interfaces
- Describe the design goals of the Java language in providing an interface

Day 4

CHAPTER 4

Exception Handling

Objectives

At the end of this session, you will learn to:

- Describe the exception handling mechanism of Java
- Describe the use of the keywords that comprise Java's exception handing mechanism

What is an Exception?

- In procedural programming, it is the responsibility of the programmer to ensure that the programs are error-free
- Errors have to be checked and handled manually by using some error codes
- But this kind of programming was very cumbersome and led to spaghetti code
- Java provides an excellent mechanism for handling runtime errors

What is an Exception?

- **An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions**

- **The ability of a program to intercept run-time errors, take corrective measures and continue execution is referred to as exception handling**

What is an Exception?

- Consider an example in which we try to open a file and read from it:
  - ask the user to enter the name of a file
  - open the file
  - read the contents
  - close the file
- There are various situations when an exception could occur:
  - Attempting to access a file that does not exist
  - Inserting an element into an array at a position that is not in its bounds
  - Performing some mathematical operation that is not permitted
  - Declaring an array using negative values
  - Using an un-initialized instance variable

Exception Types

- Exceptions are implemented in Java through a number of classes
- **Throwable** is at the top of the exception class hierarchy
- It has two subclasses – the **Exception** class and the **Error** class.
- The **Exception** class has a subclass called **RunTimeException**.
- Exceptions that belong to **RunTimeException** handle various situations such as division by zero, file not found etc.

Uncaught Exceptions

```
class  Demo {
public static void main(String args[])
 {
int x = 0;
int y = 50/x;
System.out.println("y  = " +y);
}
 }
```

- Although this program will compile, but when you execute it, the Java run-time-system will generate an exception and displays the following output on the console

- java.lang.ArithmeticException: / by zero
- at Demo.main(Demo.java:4)

Exception Types

- There are several built-in exception classes that are used to handle the very fundamental errors that may occur in your programs.

- You can create your own exceptions also by extending the **Exception** class.

- These are called user-defined exceptions, and will be used in situations that are unique to your applications.

Handling Runtime Exceptions

- Whenever an exception occurs in a program, an object representing that exception is created and thrown in the method in which the exception occurred

- Either you can handle the exception, or ignore it

- In the latter case, the exception is handled by the Java run-time-system and the program terminates

- However, handling the exceptions will allow you to fix it, and prevent the program from terminating abnormally

Handling Runtime Exceptions

- Java's exception handling is managed using the following keywords: **try, catch, throw, throws and finally.**

```
▪ try
▪  {
▪   // code comes here
▪   }
▪ catch(TypeofException  obj)
▪  {
▪              //handle the exception
▪    }
▪       finally
▪   {
▪            //code to be executed before the program ends
▪   }
```

Handling Runtime Exceptions

- Any part of the code that can generate an error should be put in the **try** block
- Any error should be handled in the **catch** block defined by the **catch** clause
- This block is also called the **catch block**, or the **exception handler**
- The corrective action to handle the exception should be put in the **catch** block

Handling Runtime Exceptions

```
class ExceptDemo
{
public static void main(String args[])
{
int x, a;

try
{
x = 0;
a = 22 / x;
System.out.println("This will be bypassed.");
}
catch (ArithmeticException e)
{
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

Multiple Catch Statements

- A single block of code can raise more than one exception
- You can specify two or more **catch** clauses, each catching a different type of execution
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

Multiple Catch Statements

```
class MultiCatch
{
public static void main(String args[])
{
try
{
int l = args.length;
System.out.println("l = " + l );
int b = 42 / l;
int arr[] = { 1 };
arr[22] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}
```

Multiple Catch Statements Involving Exception Superclasses and Subclasses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their exception superclasses
- This is because a catch statement that uses a superclass will catch exceptions of that type as well as exceptions of its subclasses
- Thus, a subclass exception would never be reached if it came after its superclass that manifests as an **unreachable code error**

Nested try **Statements**

- The **try** statement can be nested
- Each time a **try** statement is encountered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a **catch** handler for a particular exception, the outer block's catch handler will handle the exception
- This continues until one of the **catch** statement succeeds, or until all of the nested **try** statements are exhausted

Using throw

- System-generated exceptions are thrown automatically
- At times you may want to throw the exceptions explicitly which can be done using the **throw** keyword
- The exception-handler is also in the same block
- The general form of throw is:
  - throw **ThrowableInstance**
- Here, **ThrowableInstance** must be an object of type **Throwable**, or a subclass of **Throwable**

Using throw

```
class ThrowEg
{
static void demofunc()
{
try
{
throw new ArithmeticException( );
}
catch(ArithmeticException e)
{
System.out.println("Caught inside demoproc.");
throw e;
}
}

public static void main(String args[])
{
```

```
try
{
demofunc();
}
catch(ArithmeticException e)
{
System.out.println("Recaught: " + e);
}
}
}
```

Using throws

- Sometimes, a method is capable of causing an exception that it does not handle
- Then, it must specify this behaviour so that callers of the method can guard themselves against that exception
- While declaring such methods, you have to specify what type of exception it may throw by using the **throws** keyword
- A **throws** clause specifies a comma-separated list of exception types that a method might throw:
  - type method-name(parameter list) throws exception-list

Using throws

- class ThrowsDemo
- {
- static void throwOne()
- {
- System.out.println("Inside throwOne.");
- throw new FileNotFoundException();
- }

- public static void main(String args[])
- {
- throwOne();
- }
- }

Using throws

```
import java.io.*;
class ThrowsDemo
{
static void throwOne() throws FileNotFoundException
{
System.out.println("Inside throwOne.");
throw new FileNotFoundException();
}
public static void main(String args[]) {
try
{
throwOne();
}
catch (FileNotFoundException e)
{
System.out.println("Caught " + e);
}
}
}
```

Using finally

- When an exception occurs, the execution of the program takes a non-linear path, and could bypass certain statements
- A program establishes a connection with a database, and an exception occurs
- The program terminates, but the connection is still open
- To close the connection, **finally** block should be used
- The finally block is guaranteed to execute in all circumstances

Using finally

```
import java.io.*;
class FinallyDemo
{
static void funcA() throws FileNotFoundException
{
try
{
System.out.println("inside funcA( )");
throw new FileNotFoundException( );
}
finally
{
System.out.println("inside finally of funA( )");
}
}
```

Using finally

```
static void funcB()
{
try
{
System.out.println("inside funcB( )");

}
finally
{
System.out.println("inside finally of funB( )");
}
}
}
```

Using finally

```
 static void funcC()
{
try
{
System.out.println("inside funcC( )");
}
finally
{
System.out.println("inside finally of funcC( )");
}
}

public static void main(String args[])
{
try
{
funcA();
}
```

Using finally

```
catch (Exception e)
{
System.out.println("Exception caught");
}
funcB( );
funcC( );
}
}
```

Checked and Unchecked Exceptions

- The subclasses of Exception class are further divided into two categories:
  - **checked exceptions**
  - **unchecked exceptions**

- Checked exceptions are exceptions that should be specified in the **throws** list of the methods that may throw these exceptions, and does not handle them itself

Checked and Unchecked Exceptions

- Most exceptions derived from **RuntimeException** are automatically available since they are part of java.lang
- These exceptions need not be included in a method's **throws** list. Such exceptions are called unchecked exceptions
- The compiler does not check to see if a method handles or throws these exceptions
- Hence these exceptions are called unchecked exceptions

Summary

In this session, you learnt to:

- Describe the exception handling mechanism of Java
- Describe the use of the keywords that comprise Java’s exception handing mechanism

Day 5

CHAPTER 5

Multithreading

Objectives

At the end of this session, you will learn to:

- Define multithreading
- Differentiate between multitasking and multithreading
- Describe Java’s multithreading mechanism in the form of the Thread class and the Runnable interface
- Describe some key methods of the Thread class
- Describe Thread priorities
- Describe race conditions that are likely to occur between threads that are not synchronized
- Define thread synchronization through the use of the synchronized keyword
- Facilitate inter-thread communication with the help of wait( ), notify(), and notifyAll( ) methods

Need for Multithreading

- Have you faced the following situations:
  - Your browser cannot skip to the next web page because it is downloading a file?
  - You cannot enter text into your current document until your word process completes the task of saving the document to disk

What is Multitasking?

- Multitasking is synonymous with process-based multitasking, whereas multithreading is synonymous with thread-based multitasking
- All modern operating systems support multitasking
- A process is an executing instance of a program
- Process-based multitasking is the feature by which the operating system runs two or more programs concurrently

What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler
- A single program can perform two tasks using two threads
- Only one thread will be executing at any given point of time given a single-processor architecture

Multitasking Vs. Multithreading

- when compared to multitasking processes
  - Each process requires its own separate address space
  - Context switching from one process to another is a CPU-intensive task needing more time
  - Inter-process communication between processes is again expensive as the communication mechanism has to span separate address spaces

- These are the reasons why processes are referred to as heavyweight tasks

Multitasking Vs. Multithreading

- Multitasking threads cost less in terms of processor overhead because of the following reasons
  - Multiple threads in a program share the same address space, and cooperatively share the same heavyweight process
  - Context switching from one thread to another is less CPU-intensive
  - Inter-thread communication, on the other hand, is less expensive as threads in a program communicate within the same address space

- Threads are therefore called lightweight processes

What is Multithreading

- A multithreaded application performs two or more activities concurrently
- Accomplished by having each activity performed by a separate thread
- Threads are the lightest tasks within a program, and they share memory space and resources with each other

Single-Threaded Systems

- Single-threaded systems use an approach called an event loop with polling.

- In this model:
  - A single thread of control runs in an infinite loop
  - Polling a single event queue to decide which instruction to execute next
  - Until this instruction returns, nothing else can happen in the system
  - This results in wastage of precious CPU cycles

Java's Multithreading Model

- Java has completely done away with the event loop/ polling mechanism

- *In Java*
  - *All the libraries and classes are designed with multithreading in mind*
  - *This enables the entire system to be asynchronous*

- In Java:
  - the **java.lang.Thread** class is used to create thread-based code
  - imported into all Java applications by default

Thread Priorities

- A thread priority:
  - decides how that thread should be treated with respect to other threads
  - is set when created
  - is used to decide when to switch from one running thread to another

- This is called a **context switch**.

- Higher priority threads are guaranteed to be scheduled before lower priority threads

Deciding On a Context Switch

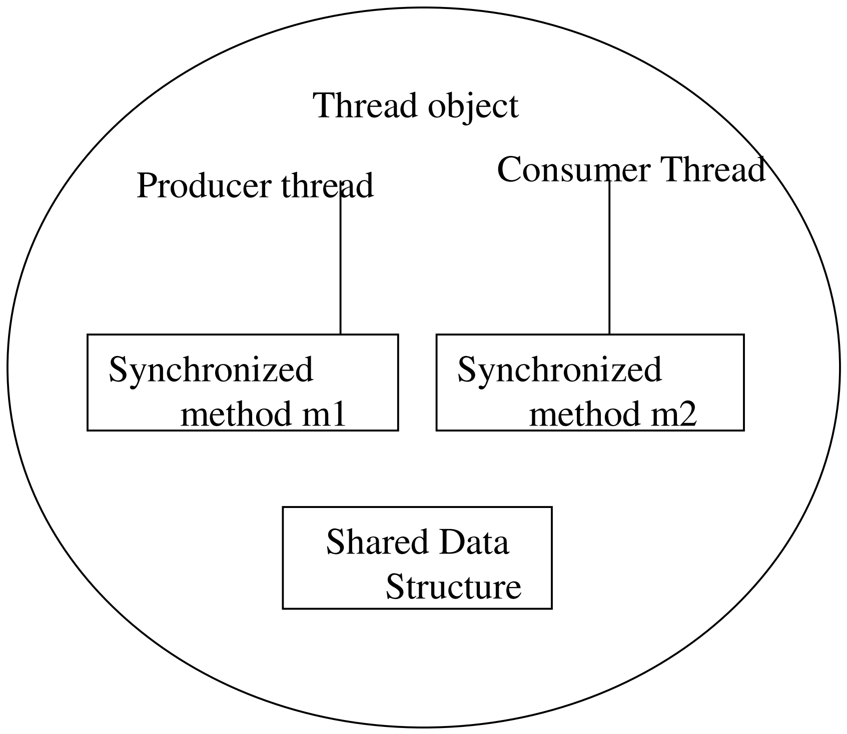
- A thread can voluntarily relinquish control:
  - by explicitly yielding, sleeping, or blocking on pending Input/Output
- All threads are examined and the highest-priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread:
  - a lower-priority thread that does not yield the processor is superseded, or preempted by a higher-priority thread
  - Whenever a higher priority thread wants to run, it does
  - This is called preemptive multitasking
- **When two threads with the same priority are competing for CPU time, threads are time-sliced in round-robin fashion in case of Windows like OSs**

Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- For example, imagine a Java application where:
  - one thread (the producer) writes data to a data structure,
  - while a second thread (the consumer) reads data from the data structure

Synchronization

- this example use concurrent threads that share a common resource: a data structure.
-



Synchronization

  - The current thread operating on the shared data structure, must be granted mutually exclusive access to the data
  - the current thread gets an exclusive lock on the shared data structure, or a **mutex**

- A **mutex** is a concurrency control mechanism used to ensure the integrity of a shared data structure

Synchronization

- Mutex is not assured, if, the methods of the object, accessed by competing threads are ordinary methods
- It might lead to a race condition when the competing threads will race each other to complete their operation
- A **race condition** can be prevented by defining the methods accessed by the competing threads as **synchronized**

Synchronization

- Synchronized methods are an elegant variation on an time-tested model of interprocess-synchronization: the **monitor**.
- The monitor is a thread control mechanism
- When a thread enters a monitor **(synchronized method)**, all other threads must wait until that thread exits the monitor
- The monitor acts as a concurrency control mechanism

Thread Messaging

- In Java, you need not depend on the OS to establish communication between threads
- All objects have predefined methods, which can be called to provide inter-thread communication.

The Thread Class

- Java’s multithreading feature is built into the **Thread** class

- The **Thread** class has two primary thread control methods:
  - **public void start() -** The **start()** method starts a thread execution
  - **public void run()** - the **run()** method actually performs the work of the thread, and is the entry point for the thread
  - The thread *dies* when the **run( )** method terminates
  - You never call **run( )** explicitly
  - The **start( )** method called on a thread automatically initiates a call to the thread’s **run( )** method

The main Thread

- When a Java program starts executing
  - the main thread begins running
  - the main thread is immediately created when **main()** commences execution
  - Information about the main or any thread can be accessed
  - by obtaining a reference to the thread using a public, static method in the **Thread** class called **currentThread( )**

Obtaining Thread-Specific Information

```
class ThreadInfo
{
public static void main(String args[])
{
Thread t = Thread.currentThread( );
System.out.println("Current Thread :" + t);
t.setName("Demo Thread");
System.out.println("New name of the thread :"  +  t);
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println("Main Thread Interrupted");
}
}
}
```

Creating Threads

- A thread can be created by instantiating an object of type **Thread**.

- Java defines two ways in which this can be accomplished:
  - implementing the **Runnable** interface
  - extending the **Thread** class

- **Creating Threads – Implementing Runnable**
  - Create a class that implements the **Runnable** interface
  - **Runnable** abstracts a unit of executable code for the thread
  - A thread can be constructed on any object that implements the **Runnable** interface
  - To implement **Runnable**, a class need implement only a single method called **run( )**

Creating Threads – Implementing Runnable

- After you define a class that implements **Runnable**, you will instantiate an object of type thread from within an object of that class. This thread will end when **run( )** returns, or terminates.

- This is mandatory because a thread object confers multithreaded functionality to the object from which it is created.

- Therefore, at the moment of thread creation, the thread object must know the reference of the object to which it has to confer multithreaded functionality. This point is borne out by one of the constructors of the **Thread** class.

Creating Threads – Implementing Runnable

- The **Thread** class defines several constructors one of which is:
  - **Thread(Runnable threadOb, String threadName)**
- In this constructor, **threadOb:**
  - is an instance of a class implementing the **Runnable** interface
  - ensures that the thread is associated with the **run( )** method of the object implementing **Runnable**
- This defines where execution of the thread will begin
- The name of the new thread is specified by **threadName**.

Creating Threads – Implementing Runnable

```
class DemoThread implements Runnable
{
Thread t;
DemoThread()
{
t = new Thread(this, "Demo Thread");
System.out.println("Child Thread: " + t);
t.start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
```

Creating Threads – Implementing Runnable

```
catch (InterruptedException e)
{
System.out.println("Child Thread Interrupted");
}
System.out.println("Exiting Child Thread");
}
}
class ThreadImpl
{
public static void main(String args[])
{
new DemoThread();

try
{
for(int i=5; i>0; i--)
{
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
}
```

Creating Threads – Implementing Runnable

```
catch (InterruptedException e)
{
System.out.println("Main Thread Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Extending Thread

- An alternative way to create threads:
  - Is to create a new class that extends **Thread**
  - Create an instance of that class
  - The extending class must override the **run( )** method
  - run( ) is the entry point for the new thread
  - The **start( )** method must be invoked on the thread
  - **start( )** initiates a call to the thread's **run( )** method

- The earlier program is rewritten by extending the **Thread** class.

Extending Thread

```
class DemoThread extends Thread
{
DemoThread()
{
super("Demo Thread");
System.out.println("Child Thread:" + this);
start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
}
```

Extending Thread

```
catch (InterruptedException e)
{
System.out.println("Child Thread Interrupted");
}
System.out.println ("Exiting Child Thread");
}
}

class ThreadImpl
{
public static void main(String args[])
{
new DemoThread();
try
{
for(int i=5; i>0; i--)
{
```

Extending Thread

```
System.out.println("Child Thread: " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main Thread     Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Implementing Runnable or Extending Thread

- A modeling heuristic pertaining to hierarchies says that classes should be extended only when they are enhanced or modified in some way

- So, if the sole aim is to define an entry point for the thread:
  - (by overriding the **run( )** method)
  - and not override any of the **Thread** class' other methods
- it is recommended to implement the **Runnable** interface

Creating Multiple Threads

- You can launch as many threads as your program needs

- The following example is a program spawning multiple threads:

```
class DemoThread implements Runnable
{
String name;
Thread t;

DemoThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New Thread: " + t);
t.start();
}
```

Creating Multiple Threads

```
public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(1000); }
}
catch (InterruptedException e)
{
System.out.println(name + "Interrupted");
}
System.out.println(name +"Exiting");
}
}

class MultiThreadImpl
{
public static void main(String args[])
{
```

Creating Multiple Threads

```
new DemoThread("One");
new DemoThread("Two");
new DemoThread("Three");
try
{
Thread.sleep(10000);
}
catch (InterruptedException e)
{
System.out.println("Main Thread Interrupted");
}
System.out.println("Main Thread Exiting");
}
}
```

Control Thread Execution - The isAlive( ) & join( ) methods

- Two ways exist by which you can determine whether a thread has finished:

- The **isAlive( )** method will return true if the thread upon which it is called is still running; else it will return false.

- The **join( )** method waits until the thread on which it is called terminates.

Control Thread Execution - The IsAlive() & Join( ) methods

```
class DemoThread implements Runnable
{
String name;
Thread t;
DemoThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New Thread: " + t);
t.start();
}

public void run()
{
try
{
for(int i=5; i>0; i--)
{
System.out.println("Child Thread: " + i);
Thread.sleep(1000); }
}
```

Control Thread Execution - The IsAlive() & Join( ) methods

```
catch (InterruptedException e)
{
System.out.println(name + "Interrupted");
}
System.out.println(name +"Exiting");
}
}
class MultiThreadImpl
{
public static void main(String args[])
{
DemoThread t1 = new DemoThread("One");
DemoThread t2 = new DemoThread("Two");
DemoThread t3 = new DemoThread("Three");
System.out.println("Thread One is alive: " +   t1.t.isAlive());
System.out.println("Thread Two is alive: " +  t2.t.isAlive());
System.out.println("Thread Three is alive: "  + t1.t.isAlive());
try
{
System.out.println("Waiting for child  threads to finish");
t1.t.join();
t2.t.join();
t3.t.join();
}
```

Control Thread Execution - The IsAlive() & Join( ) methods

```
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
System.out.println("Thread One is alive: " +  t1.t.isAlive());
System.out.println("Thread One is alive: " +  t1.t.isAlive());
System.out.println("Thread One is alive: " +   t1.t.isAlive());
System.out.println("Main thread exiting");
}
}
```

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run
- Higher-priority threads get more CPU time than lower-priority threads
- A higher priority thread can also preempt a lower priority thread
- Actually, threads of equal priority should evenly split the CPU time

Thread Priorities

- For safety:
  - Threads that share the same priority should yield the CPU once in a while
  - This ensures that all threads have a chance to run in a non-preemptive operating system
  - In a non-preemptive OS, most threads get a chance to run
  - Threads invariably encounter some blocking situation, such as sleeping for a specified time, or waiting for I/O
  - When a thread is blocked, the blocked thread is suspended, and other threads can run.

Thread Priorities

- CPU-intensive threads must be made to yield control occasionally, so that other threads can run
- Every thread has a priority.
- When a thread is created it inherits the priority of the thread that created it
- The methods for accessing and setting priority are as follows:
  - public final int getPriority( );
  - public final void setPriority (int level);

Thread Priorities

- When multiple threads are ready to be executed, the runtime system chooses the highest priority runnable thread to run
- Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread start executing
- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion

Thread Priorities

```
class Counter implements Runnable
{
int count = 0;
Thread t;
private volatile boolean running = true;
public Counter(int p_level)
{
t = new Thread(this);
t.setPriority(p_level);
}

public void run()
{
while (running)
{
count++;
}
}
```

Thread Priorities

```
public void stop()
{
running = false;
}

public void start()
{
t.start();
}
}

class HighLowPriorityTest
{
public static void main(String args[])
{
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Counter high = new Counter(Thread.NORM_PRIORITY+2);
Counter low = new  Counter(Thread.NORM_PRIORITY-2);
low.start();
high.start();
```

Thread Priorities

```
try
{
Thread.sleep(10000);
}
catch(InterruptedException e)
{
System.out.println("Main Thread  Interrupted");
}
low.stop();
high.stop();
try
{
low.t.join();
high.t.join();
}
catch(InterruptedException e)
{
System.out.println("interrupted Exception caught");
}
System.out.println("Low Priority Thread’s Iterations: " + low.count);
System.out.println("High Priority Thread’s Iterations: " + high.count);
}
}
```

Synchronization

- Threads often need to share data
- Need for a mechanism to ensure that the shared data will be used by only one thread at a time
- This mechanism is called synchronization.
- Key to synchronization is the concept of the **monitor** (also called a semaphore).

Using Synchronized Methods

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them

- To enter an object's monitor, just call a **synchronized** method on the object

- While a thread is inside a **synchronized** method, all other threads that try to call that synchronized method (or any other **synchronized** method on the same object) will have to wait

Using Synchronized Methods

```
class Callme
{
void call( String msg)
{
System.out.print( "[" + msg );
try
{
Thread.sleep(1000);
}
catch (InterruptedException e)
{
System.out.println( "Interrupted");
}
System.out.println( "]" );
}
}
```

Using Synchronized Methods

```
class Caller implements Runnable
{
String msg;
Callme target;
Thread t;
public Caller (Callme targ, String s)
{
target = targ;
msg = s;
t = new Thread( this);
t.start( );
}

public void run( )
{
target.call( msg);
}
}
class Synch
{
```

Using Synchronized Methods

```
public static void main (String args[ ] )
{
Callme target = new Callme( );
Caller obj1 = new Caller( target, "Hello");
Caller obj2 = new Caller( target, "Synchronized");
Caller obj3= new Caller( target, "World");
try
{
obj1.t.join( );
obj2.t.join( );
obj3.t.join( );
}
catch (InterruptedException e)
{
  System.out.println( "Interrupted");
}
}
}
```

Using Synchronized Methods

- To fix this problem:
  - You must **serialize** access to **call( )**
  - You must restrict its access to only one thread at a time.
  - You need to precede **call( )**'s definition with the keyword **synchronized**

```
class Callme {
synchronized void call( String msg){........ } }
class Callme {
   void call( String msg)
    { System.out.print( "[" + msg )
      try {
            Thread.sleep(1000);
           }catch (InterruptedException e){
        System.out.println( "Interrupted");  }
     System.out.println( "]" ) } }
```

- class Caller implements Runnable {
-    String msg;

The Synchronized Statement

- Callme target;
- Thread t;
- public Caller (Callme targ, String s)
- { target = targ;
- msg = s;
- t = new Thread(this);
- t.start( ); }
- public void run( )
- { synchronized(target) { //synchronized block
- target.call(msg); }}
- class Synch { public static void main (String args[ ])
- {Callme target = new Callme( );
- Caller obj1 = new Caller(target, “Hello”);
- Caller obj2 = new Caller(target, “Synchronized”);
- Caller obj1 = new Caller(target, “World”);
- try {ob1.t.join( );
- ob2.t.join( );
- ob3.t.join( ); }
- catch (InterruptedException e) { System.out.println(“Interrupted”); }}}

Inter-Thread Communication

- Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request

- The words ***wait*** and ***notify*** encapsulate the two central concepts to thread communication
    - A thread waits for some condition or event to occur
    - You notify a waiting thread that a condition or event has occurred

- To avoid polling, Java's elegant inter-thread communication mechanism uses:
  - **wait( )**
  - **notify( ), and notifyAll( )**

Inter-thread Communication

- wait( ), notify( ) and notifyAll( ) are:
  - Declared as **final** in **Object**
  - Hence, these methods are available to all classes
  - These methods can only be called from a **synchronized** context
- **wait( )** tells the calling thread to give up the monitor, and go to sleep until some other thread enters the same monitor, and calls **notify( )**
- **notify( )** wakes up the other thread that called **wait( )** on the same object

Inter-thread Communication

- The following sample program incorrectly implements a simple form of the producer/consumer problem.

- It consists of four classes namely:
  - **Q**, the queue that you are trying to synchronize
  - **Producer**, the threaded object that is producing queue entries
  - **Consumer**, the threaded object that is consuming queue entries
  - **PC**, the class that creates the single Queue, Producer, and Consumer

Inter-thread Communication

```
class Q
{
int n;
synchronized int get( )
{
System.out.println( "Got: " + n);
return n;
}
synchronized void put( int n)
{
this.n = n;
System.out.println( "Put: " + n);
}
}
class Producer implements Runnable
{
Q q;
Producer ( Q q)
{
this.q = q;
new Thread( this, "Producer").start( );
}
```

Inter-thread Communication

```
public void run( )
{
int i = 0;
while (true)
{
q.put (i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer ( Q q)
{
this.q = q;
new Thread (this, "Consumer").start( );
}
public void run( )
{
while (true)
{
q.get( );
}
```

Inter-thread Communication

```
}
}
class PC
{
public static void main (String args [ ] )
{
Q q = new Q( );
new Producer (q);
new Consumer (q);
System.out.println("Press Control-C to stop");
}
}
```

Inter-thread Communication Using wait( ) & notify( )

```
class Q
{
int n;
boolean valueset = false;
synchronized int get( )
{
if (!valueset)
try
{
wait( );
}
catch (InterruptedException e)
{
System.out.println("InterruptedException caught");
}
System.out.println( "Got: " + n);
valueset = false;
notify( );
return n;
}
```

Inter-thread Communication

```
synchronized void put( int n)
{
if (valueset)
try
{
wait( );
}
catch (InterruptedException e)
{
System.out.println("InterruptedException caught");
}
this.n = n;
valueset = true;
System.out.println( "Put: " + n);
notify( );
}
}


class Producer implements Runnable
{
Q q;
Producer ( Q q)
```

Inter-thread Communication

```
{
this.q = q;
new Thread( this, "Producer").start( );
}
public void run( )
{
int i = 0;
while (true)
{
q.put (i++);
}
}
}
class Consumer implements Runnable
{
Q q;
Consumer ( Q q)
{
this.q = q;
```

Inter-thread Communication

```
new Thread (this, "Consumer").start( );
}
public void run( )
{
while (true)
{
q.get( );
}
}
}
class PC
{
public static void main (String args [ ] )
{
Q q = new Q( );
new Producer (q);
new Consumer (q);
System.out.println("Press Control-C to stop");
}
}
```

Summary

- In this session, you learnt to:
- Define multithreading
- Differentiate between multitasking and multithreading
- Describe Java's multithreading mechanism in the form of the Thread class and the Runnable interface
- Define Thread priorities
- Describe some key methods of the Thread class
- Describe race conditions that are likely to occur between threads that are not synchronized
- Define thread synchronization through the use of the synchronized keyword
- Facilitate inter-thread communication with the help of wait( ), notify(), and notifyAll( ) methods

Day 6

CHAPTER 6

Utility Classes and Interfaces

Objectives

At the end of this session, you will learn to:

- Describe the need for wrapper classes
- Define wrapper classes
- Describe the context of using the Cloneable interface
- Facilitate the processing of arrays and collections through the Enumeration interface
- Define dynamic arrays in the form of Vector objects
- Describe the context in which to use the instanceof operator
- Describe finalization

java.util

- Some of the most important utility classes are provided in **java.util** package
- They are: **Vector**, **Dictionary**, **Hashtable**, **StringTokenizer** and **Enumeration**
- The **java.util** package hosts frequently used
  - data structures like Stack, LinkedList, HashMap etc
  - functionalities like sort, binary search etc.

Wrapper Classes

- Heterogeneous values can be stored in structures like **vectors**, or **hashtables**
- However, only objects can be stored in vectors and hashtables
- **Primitive data types cannot be stored directly in vectors and hashtables,** and hence have to be converted to objects
- We have to wrap the primitive data types in a corresponding object, and give them an object representation

Wrapper Classes

- **Definition: The process of converting the primitive data types into objects is called wrapping.**

- To declare an integer ‘i’ holding the value 10, you write   int i = 10;

- The object representation of integer ‘i’ holding the value 10 will be:
  - **Integer** iref = new Integer(i);

- Here, class **Integer** is the wrapper class wrapping a primitive data type **i**.

The Character Class

- **Character** class is a wrapper class for character data types. The constructor for **Character** is:
  - Character(char *c*)
  - Here, *c* specifies the character to be wrapped by the **Character** object.

- After a **Character** object is created, you can retrieve the primitive character value from it using:
  - char charValue( )

The Boolean **Class**

- The Boolean class is a wrapper around **boolean** values.

- It has the following constructors:
  - **Boolean(boolean *bValue*)**
    - Here, *bValue* can be either **true** or **false**.
  - **Boolean(String *str*)**
    - The object created by this constructor will have the value **true** or **false** depending upon the string value in *str* – “**true**” or “**false**”
    - The value of *str* can be in upper case or lower case

The Double **Class**

- Class **Double** is a wrapper for floating-point values of type **double**
- **Double** objects can be constructed with a **double** value, or a string containing a floating-point value.
- The constructors for double are shown here:
  - Double(double num)
  - Double(String str) throws NumberFormatException
- Some methods of the **Double** class:
  - static Double valueOf(String str) throws NumberFormatException
  - Double doubleValue( ) returns the value of the invoking object as a **double** value

The Cloneable Interface

- When you make a copy of an object reference:
  - The original and copy are references to the same object
  - This means a change to either variable also affect the other

- Assume a class Day { ……… }
- Day bday = new Day(1975, 04, 22);
- Day d = bday;
- d.advance(100);

The Cloneable Interface

- Day bday = new Day(1975, 04, 22);
- Day d = (Day)bday.clone( );
- d.advance(100);

- The **clone( )** method:
  - is a protected member of **Object**,
  - can only be invoked on an object that implements Cloneable

- Object cloning performs a bit-by-bit copy

The Enumeration **Interface**

- The **Enumeration** interface is implemented by the classes like **Vector, Dictionary**, and **Hashtable** that deal with collections
- Whenever you have to manipulate such collections, you should use the methods present in the **Enumeration** interface
- For example, you may:
  - Want to find if there are any more objects present in the collection
  - You may also want to retrieve the values from the collection
- To achieve this, you should implement the **Enumeration** interface

The Enumeration **Interface**

- The **Enumeration** interface allows you to traverse through each element in the collection

- The two methods in the **Enumeration** interface that help you to enumerate the elements in a collection:
  - **boolean hasMoreElements()** – returns **true** if some more objects are present in the collection, else returns **false**.
  - **Object nextElement()** – returns the next object in the collection.

The Vector Class

- A vector is:
  - an array that can store objects
  - dynamic
  - heterogeneous

- You can add and remove objects from a vector at run time
- import java.util.*;
- class VecExample {
- public static void main(String args[]) {
- Vector v = new Vector(5, 2);
- System.out.println("Initial Size: " + v.size());
- System.out.println("Initial Capacity: " + v.capacity());
- v.addElement(new Integer(10));

The Vector Class

- v.addElement(new Integer(20));
- v.addElement(new Integer(30));
- v.addElement(new Integer(40));
- v.addElement(new Integer(50));
- v.addElement(new Double(60));
- System.out.println("Capacity after 6 Additions :"+v.capacity());
- System.out.println("FirstElement :" + (Integer)v.firstElement());
- System.out.println("LastElement :" + (Double)v.lastElement());
- if(v.contains(new Integer(10)))
- System.out.println("Vector contains 10");
- Enumeration enum = v.elements();
- System.out.println("\n Elements in Vector v are: ");
- while(enum.hasMoreElements())
- System.out.println(enum.nextElement()+ " "); }}

The instanceof Operator

- Knowing the type of an object during runtime is useful in situations involving casting

- In Java:
  - invalid casts cause runtime errors
  - many invalid casts can be caught at compile time
  - casts involving class hierarchies can produce invalid casts that can be detected only at runtime

The instanceof **Operator**

- The **instanceof** operator has this general form:
  - **object** instanceof **type**
- class A { int i, j;}
- class B extends A {int a, b;}
- class C extends A {int m, n;}
- class InstanceofImpl {public static void main(String args[ ])
- { A a = new A( );
-   B b = new B( );
- C c = new C( );
- A ob = b;
- if (ob instanceof B) System.out.println(“ob now refers to B”);
- ob = c;
- if (ob instanceof C) System.out.println(“ob now refers to C”); }}

The finalize( ) Method

- Often, an object needs to perform some action when it is destroyed
- The action could pertain to:
  - releasing a file handle
  - reinitializing a variable, such as a counter
- Java's answer is a mechanism called **finalization**
- By using **finalization**, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector

The finalize( ) **Method**

- To add a finalizer to a class, you simply define the **finalize( )** method
- The Java runtime calls that method whenever it is about to recycle an object of that class
- Inside the **finalize( )** method, you will specify those actions that must be performed before an object is destroyed

Example Demonstrating Garbage Collection and Finalization

```
class CounterTest { public static int count;
   public CounterTest ( ) { count++; }
   public static void main(String args[ ])   {
     CounterTest ob1 = new CounterTest ( ) ;
     System.out.println("Number of objects :" + CounterTest.count) ;
     CounterTest ob2 = new CounterTest ( );
     System.out.println("Number of objects :" + CounterTest.count) ;
     Runtime r = Runtime.getRuntime( );
     ob1 = null;
     ob2 = null;
     r.gc( );  }
protected void finalize( )
  {
   System.out.println("Program about to terminate");
   CounterTest.count --;
   System.out.println("Number of objects :" + CounterTest.count) ;
  }}
```

Summary

In this session, you learnt to:

- Describe the need for wrapper classes
- Define wrapper classes
- Describe the context of using the Cloneable interface
- Facilitate the processing of arrays and collections through the Enumeration interface
- Define dynamic arrays in the form of Vector objects
- Describe the context in which to use the instanceof operator
- Describe finalization

CHAPTER 7

Applets

Objectives

At the end of this session, you will learn to:

- Define an applet
- Write Java code to create an applet
- Describe the lifecycle of an applet
- Deploy an applet either through a browser or the appletviewer
- Pass parameters to an applet from a HTML page
- Describe the security restrictions on applets
- Define a trusted applet

Introduction to Applets

- Applets are:
  - small Java programs that are transmitted over the network
  - executed in a Java-enabled web browser
- Applets are also called downloadable Java programs.
- The early role of the Internet was confined to static content
- But, applets enable us to create dynamic web pages

Introduction to Applets

- Applets can have graphic interfaces in order to interact with the user

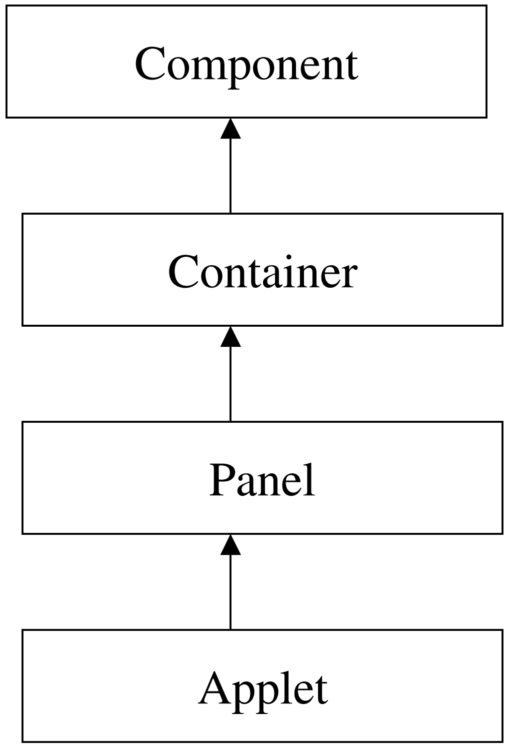
- Hence, applets helped Java become a popular language of choice for the internet

- import java.applet.*;
- import java.awt.*;
- public class MyFirstApplet extends Applet {
- public void paint(Graphics g) {
- g.drawString(“A First Applet”,50,50);} }}

Running an Applet in a Browser

- In order to run applets in a Web browser:
  - You need to embed an applet in a HTML file
  - A Web browser understands an HTML file
  - The HTML file contains the APPLET tag.
  - The HTML file looks like this. <applet code = “MyFirstClass” width=300 height=400> </applet>
  - Save this file with a **.html** extension.
- The **code** option:
  - shows the class file (compiled code) of the applet
  - tells the browser which applet to run

Applet Hierarchy



Lifecycle of an Applet

- The following are stages of an applet's lifecycle:
  - It can initialize itself
  - It can start running
  - It can stop running
  - It can perform a final cleanup, just before being unloaded

- The Applet class consists of the following lifecycle methods:
  - **init()**
  - **start()**
  - **stop()**
  - **destroy()**

Lifecycle of an Applet

- **Initialization**:
  - Initialization takes place when the applet is instantiated and loaded for the first time in the browser
  - All startup or one-time activities like declaring variables, setting background and foreground colors, setting fonts etc can be done as part of initialization

- To achieve this you need to override the **init()** method of the Applet class
  - **public void init() { // code comes here}**

Lifecycle of an Applet

- **Starting:**
  - After the applet is initialized, it starts and start( ) executes
  - start( ) called to restart an applet after it has been stopped
  - start( ) is called each time an applet's HTML page is displayed

- Whenever the applet is invoked, it starts again, whereas the initialization takes place only once

- To start the applet, you need to override the start() method of the Applet class
  - **public void start() { // code comes here }**

Lifecycle of an Applet

- **Stopping**: stop( ) method is called
  - When the browser leaves the HTML page containing the applet

- Stop functionality is achieved by overriding the **stop( )** method of the Applet class.
  - **public void stop() { // code comes here }**

Lifecycle of an Applet

- **Destroying:**
  - It happens when the applet is permanently removed from the web browser
  - It specifies the cleanup tasks that should be done prior to applet removal
- The cleanup activities could be:
  - closing the files that are in use
  - closing the connection to the database
- To achieve this you need to override the **destroy( )** method of the Applet class
  - **public void destroy( ) {   // code comes here }**

Applet Architecture

- Applets :
  - window–based programs
  - communicate with the user through a Graphical User Interface(GUI)
  - respond to events like mouse-click, button-press etc
  - run at the client side
  - can communicate to the server

Applet Architecture

- import java.applet.*; import java.awt.*;
- /* <applet code = “SimpleExample” height = 400 width =300 > </applet> */
- public class SimpleExample extends Applet {
- String str;
- public void init() {
- setBackground(Color.red);
- setForeground(Color.yellow);
- Font f = new Font(“Courier”, Font.PLAIN, 14);
- setFont(f);
- str = “Inside init() method”;   }
- public void start()
- {str += “Inside start() method”; }
- public void stop()
- { str += “Inside stop() method”;}
- public void paint(Graphics g)
- { str += “Inside paint() method”;
-   g.drawString(str,40,50); }   }

Applet Display Methods

- **repaint() :**
  - Will call update() as soon as possible
  - May skip some of the requests to update the screen
  - Gives best performance for a given infrastructure

- Call repaint() instead of update()

Applet Display Methods

```
import java.awt.*;
import java.applet.*;
/* <applet code="ScrollEx" height=400 width=300>  </applet>  */
public class ScrollEx extends Applet implements Runnable{
String str = "Welcome to Applets";
Thread t = null;
boolean  stopThread;
public void init() {
   setBackground(Color.yellow);
   setForeground(Color.red);
}
public void start() {
         t = new Thread(this);
         stopThread = false;
         t.start();
}
```

Applet Display Methods

- public void run() {
- char ch;
- for(; ;) {
- try {
- repaint();
- Thread.sleep(250);
- ch = str.charAt(0);
- str = str.substring(1, str.length());
- str += ch;
- if(stopThread)
- break;
- } catch(InterruptedException e) { System.out.println(“Error”); }}} public void stop() {
- stopThread = true;
- t = null; }
- public void paint(Graphics g) {
- g.drawString(str,50,60); }}

The Status Bar

- The browser and the appletviewer have an additional display area called the status bar.
- status bar is used to display messages that give feedback, suggest options, display error messages etc
- use **showStatus( )** method to display text in the status bar
- serves as a debugging aid

Example : showStatus()

- import java.awt.*; import java.applet.*;
- /* <applet code=”StatusEx” width=300 height=400 > </applet> */
- public class StatusEx extends Applet {
-    public void init() {
-    setBackground (Color.pink);
-   }
-   public void paint(Graphics g) {
-     g.drawString(“Welcome to Applets”, 50, 50);
-     showStatus(“This is the status displayed on the status bar”);
- }
- }

Passing Parameters to Applets

- To pass arguments to applets:
  - You use the applet's HTML file
  - Applets can receive arguments from the HTML file containing the <APPLET> tag through the use of applet parameters

- To set up and handle parameters in an applet, you need two things:
  - **<PARAM>** tag.
  - A code in your applets to parse those parameters.

Passing Parameters to Applets

- The **<PARAM>** tag in turn consists of two attributes:
  - **NAME** - a parameter *name*, which is simply a name you pick
  - **VALUE -** which is the actual value of that particular parameter

- The **<PARAM>** tag is enclosed within **<APPLET>** tag

- Example:

```
<APPLET   CODE="ParamApplet"   WIDTH=300        HEIGHT=400    >
  <PARAM              NAME=font          VALUE="TimesRoman">
  <PARAM                  NAME=size              VALUE="20">
</APPLET>
```

Receive Parameters in Applets

- receive parameters in the **init( )** method, using the **getParameter( )** method

- The **getParameter( )** method:
  - takes one argument, a string representing the name of the parameter you're looking for
  - returns a string containing the corresponding value of that parameter
  - Returns a null if the parameter referred to does not exist

Passing Parameters to Applets

- import java.awt.*; import java.applet.*;
- /* <APPLET CODE="ParamApplet" HEIGHT=400 WIDTH=300> <PARAM NAME=font VALUE="TimesRoman"><PARAM NAME=size VALUE="20"> </APPLET> */
- public class ParamApplet extends Applet {
- String fName, str;
- int fSize;
- public void init() { fName = getParameter(“font”);
-         if (fName == null) fName = “Courier”;
-        str = getParameter(“size”);
-   try {if (str != null) fSize = Integer.parseInt(str);
-         else
-         fSize = 0;
-      }catch (NumberFormatException ne){
-         fSize = -1;   } }
- public void paint(Graphics g)
- {g.drawString(“FontName:” +fName,30,30);
-   g.drawString(“FontSize:” +fSize,30,70); }}

Applet Security Restrictions

- Since applets come from an unknown host, and we cannot trace their origin, there is a possibility:
  - that an applet may contain some malicious code
  - or may contain some virus that could corrupt your system

- Therefore:
  - all applets are, by default, treated as un-trusted applets
  - applets are not allowed to access the user’s system
  - applets are allowed to execute in a specific environment termed as a security sandbox

Applet Security Restrictions

- **File Access Restrictions:**
- File system is one of the most vulnerable places for malicious attacks
- Some applet designers could design applets:
  - that can modify files on your system when you run an applet
  - that could plant viruses on your system, or just destroy data
- Hence, an applet is not allowed access to local file system resources

Applet Security Restrictions

- **Network Restrictions:** Applets can only make network connections back to the Web server they were loaded from
- However, security permissions can be relaxed for digitally signed applets
- **Trusted Applets:** Digitally signed applets are the ones that are verified by the certifying authorities like Verisign.
- For the applets to be labeled as trusted applets, they should be approved by certifying authorities

Applet Security Restrictions

- Applets cannot access system properties
- Applets cannot have their own class loader
- Applets cannot call native methods
- Applets cannot execute commands on the local system.

Summary

In this session, you learnt to:

- Define an applet
- Write Java code to create an applet
- Describe the lifecycle of an applet
- Deploy an applet either through a browser or the appletviewer
- Pass parameters to an applet from a HTML page
- Describe the security restrictions on applets
- Define a trusted applet

Day 7

CHAPTER 8

Event Handling

Objectives

At the end of this session, you will be able to:

- Define the need for event handling
- Describe Java's event handling mechanism – The Delegation Event Model
- Define an Event Source
- Define an Event Listener
- Describe event classes that encapsulate various kinds of events
- Describe the process of a listener registering with an event source
- Describe the need for an event listener to implement the relevant listener interface/s for handling specific event/s
- Write applets that handle events
- Define adaptor classes, inner classes, and anonymous inner classes

Introduction

- In console-based programs, the program:
  - may prompt the user for some input
  - processes the user input and displays the result

- In console-based programming, the user acts according to the program, and therefore, the program drives the user

- In a GUI-based program, the user initiates the interaction with the program through GUI events such as a button click

- In a GUI environment, the user drives the program

Introduction

- Whenever a user interacts with these GUI controls:
  - some event is generated
  - some action implicitly takes place that validates or adds functionality to the event

- This type of programming is called **event-driven** programming, where the program is driven by the events.

- Events are best used in GUI-based programs

What is an Event?

- When an event occurs, the GUI run-time system:
  - intercepts the event
  - notifies the program that some event has occurred
- Thus, an **event** is a signal from the GUI run-time system to the program that a user interaction has taken place
- This specific signal has to be interpreted by the program, and it must take appropriate action on the occurrence of the specific event

What is an Event?

- The GUI object on which an event is generated is called the source of the event
- In Java, all events are implemented as classes.
- When an event occurs, an object:
  - of the respective event class is created
  - encapsulates a state change in the source that generated the event
- Thus, an event can be captured as an object, that describes a state change on the source

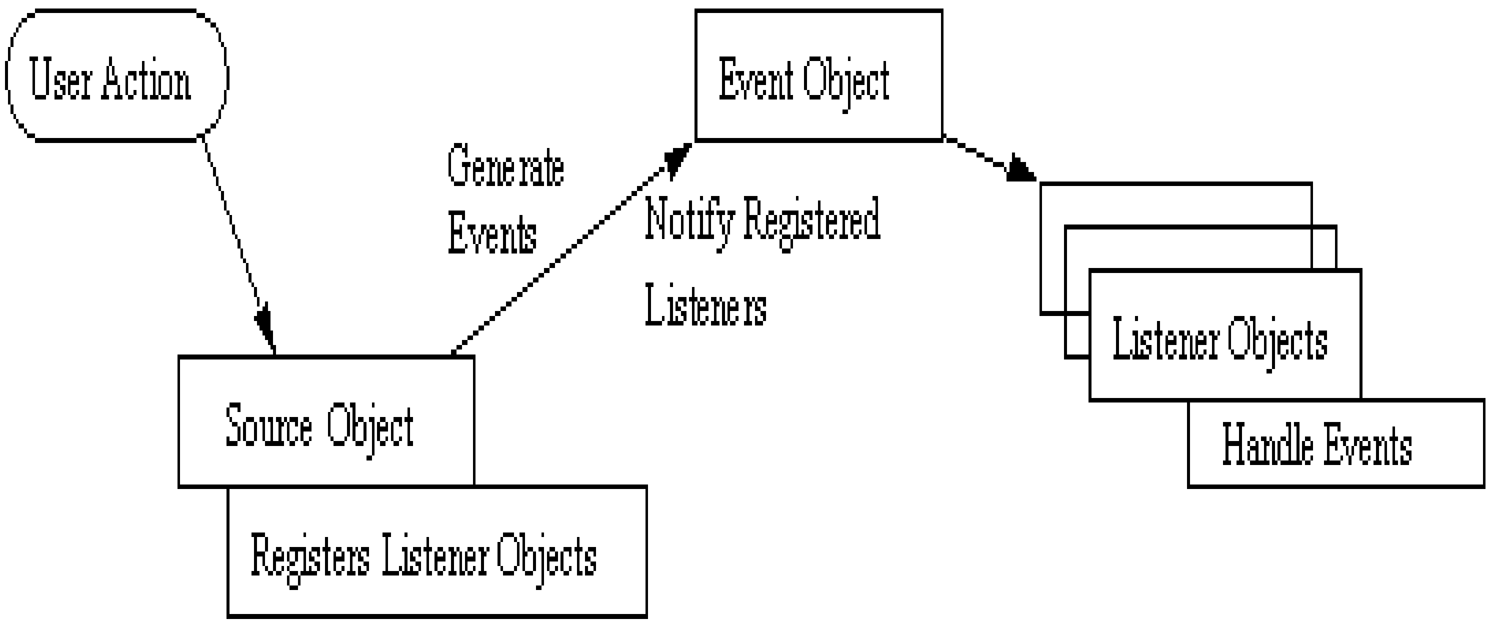
The Delegation Event Model

- **The Delegation Event Model** is a specification to generate and process events

- **The Delegation Event Model** specifies that
  - a source generates an event
  - notification of the event is sent to one or more registered listeners

- Listeners are objects:
  - that register themselves with the source
  - that implement the relevant interface that contain method/s capable of handling the corresponding event

Delegation Event Model



Delegation Event Model – Event Source

- **Event Sources:** An **event-source** is an object that generates an **event**
- An event occurs when the internal state of the object is changed in some way
- A source may generate more than one event
- For example, a button, a list, a checkbox or a scrollbar are examples of **event-sources**.

Delegation Event Model – Event Source

- **Event Sources:** Listeners register themselves with a source to facilitate notifications to these listeners about a specific type of event on the source

- Each type of event has its own registration method whose general form is:
  - public void add*Type*Listener(*Type*Listener *lis)*
  - where –'*Type*' is the type of the event(Action event, Window event, or Mouse event).
  - The registering method takes a parameter *lis*, which is a reference to the event listener.

Delegation Event Model – Event Listener

- A listener:
  - is an object that is notified when an event occurs
  - must have been registered with one or more sources to receive notifications about specific types of events
  - must implement methods defined in the relevant listener interface to process specific types of events

- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event** package

Delegation Event Model

- Some of the event sources that generate events are:
  - **Button** - generates action events when the button is pressed
  - **TextField** - generates text event when text is entered
  - **Scrollbar** - generates adjustment events when scrollbar is manipulated
  - **Checkbox** - generates item events when the check box is selected or deselected
  - **Window** - generates window events when window is opened, closed, quit, activated, deactivated, iconified, deiconified, or closed

Event Classes

- Whenever an event is generated, an object representing that event is created
- Using this event object, you can handle events
- Events are represented as classes
- Event classes are an effective means of encapsulating the events
- The event classes form the basis for Java's event-handling mechanism

AWTEvent Subclasses

Examples for events:

- **ActionEvent** – generated when:
  - a button is clicked
- **MouseEvent** – generated when the mouse is
  - pressed, released, clicked, or when the mouse enters or exits a component
  - moved, or dragged
- **WindowEvent** – generated when the window is activated, deactivated, minimized, maximized, opened, closed or quit.

ActionEvent **Class**

- An **ActionEvent** is generated when a button is clicked

- **ActionEvent** object has following methods:
  - **String getActionCommand()**
    - returns label on the button
  - **Object getSource( )**
    - Returns a reference to the source

MouseEvent Class

- **MouseEvent** class is a subclass of the **InputEvent** class.
- A mouse event is generated when the mouse is pressed, released, clicked, entered, exited etc

Event Listener Interfaces

- The delegation event model has two parts: sources and listeners
- Listeners are created by implementing one, or more of the interfaces defined by the **java.awt.event** package
- When an event occurs, the event source
  - invokes the appropriate method defined by the listener object's implementing interface
  - provides an event object as its argument

ActionListener Interface

ActionListener Interface

- has a single method that is invoked when an action event occurs
- The method takes the reference of **ActionEvent** as its argument
- **void actionPerformed(ActionEvent ae)**

MouseListener Interface

- The **MouseListener** interface contains:
  - five methods
  - these methods take a **MouseEvent** reference as an argument

- The methods are:
  - **void mousePressed(MouseEvent me)**
  - **void mouseReleased(MouseEvent me)**
  - **void mouseClicked(MouseEvent me)**
  - **void mouseEntered(MouseEvent me)**
  - **void mouseExited(MouseEvent me)**

WindowListener Interface

- The **WindowListener** interface has:
  - seven methods that are invoked when window events take place

- These methods take a **WindowEvent** reference as an argument.
  - **void windowOpened(WindowEvent we)**
  - **void windowClosed(WindowEvent we)**
  - **void windowClosing(WindowEvent we)**
  - **void windowIconified(WindowEvent we)**
  - **void windowDeiconified(WindowEvent we)**
  - **void windowActivated(WindowEvent we)**
  - **void windowDeactivated(WindowEvent we)**

Implementing the Delegation Event Model

- Implementing the **Delegation Event Model** in a program will mandate the following steps:
  - Identify the events that will occur in the program, and their source
  - Identify the appropriate listener interfaces and implement them
  - Register the listener with the source in order for the listener to receive event notifications
  - Write the event handling code by implementing the methods defined in the listener interfaces

Implementing the Delegation Event Model - Handling Mouse Events

- import java.applet.*;
- import java.awt.*;
- import java.awt.event.*;

- /* <applet code="MouseEx" height=400 width=300> </applet> */

- public class MouseEx extends Applet implements MouseListener {

- String str = " ";
- int xcord = 30, ycord = 30;

- public void init() {
- addMouseListener(this);

- }

Implementing the Delegation Event Model - Handling Mouse Events

- public void mouseEntered(MouseEvent me) {
-   str = “Mouse entered”;
-   repaint();
- }

- public void mouseExited(MouseEvent me) {
  str = “Mouse Exited”;
-   repaint();
- }

Implementing the Delegation Event Model - Handling Mouse Events

- public void mouseClicked(MouseEvent me) {
  str = “Mouse clicked”;
- repaint();
- }
- public void mousePressed(MouseEvent me) {
- xcord = me.getX();
- ycord = me.getY();
- str = “Mouse Pressed”;
- repaint();
- }

Implementing the Delegation Event Model - Handling Mouse Events

- public void mouseReleased(MouseEvent me) {
-   str = “Mouse Released”;
-    repaint();
- }
- public void paint(Graphics g) {
- d.drawString(str, xcord, ycord);
- }
- }

Adapter Classes

- Adapter classes are classes that:
  - implicitly implement the listener interfaces
  - provide empty implementation for all the methods in the interfaces
- You can extend the adapter class and override only the methods you want to implement
- Java has provided corresponding adapter classes for most of the event listeners
- These adapter classes are present in *java.awt.event* package

Adapter Classes

Adapter class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ConatinerListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener
KeyAdapter	KeyListener
FocusAdapter	FocusListener

Nested and Inner Classes

- A nested class is:
  - a class defined within a class
  - scope is controlled by the scope of the outer class
  - has access to all the variables and methods of its outer class
  - However, vice versa does not hold true.
- There are two types of nested classes; static and non-static.
- **The most important type of nested class is the inner class. An inner class is a non-static nested class that:**
  - has access to all of the variables and methods of its outer class, and may refer to them directly
  - is fully within the scope of its enclosing class

Using Inner Classes With Event Handlers

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/* <applet code="InnerClassDemo" height=400 width=300> </applet> */
public class InnerClassDemo extends Applet {
 public void init()
  {
    addMouseListener(new MyMouseAdapter());
  }
class MyMouseAdapter extends MouseAdapter
 {
   public void mousePressed (MouseEvent me)
   {
    showStatus ("Mouse Pressed");
    }
 } // end of inner class
} // end of enclosing class
```

Anonymous Inner Classes

- A class that does not have a name is called an anonymous class
- Therefore, an inner class without a name is called an anonymous inner class
- Anonymous inner classes facilitate you to write clean and manageable event handling code

Anonymous Inner Classes

- import java.applet.*;
- import java.awt.*;
- import java.awt.event.*;
- /* <applet code="AnonymousInnerClassDemo" width=300 height=400 > </applet> */
- public class AnonymousInnerClassDemo extends Applet {
- public void init()
-   {
-     addMouseListener(new MouseAdapter( ) {
-     public void mousePressed (MouseEvent me)
-    {
-     showStatus ("Mouse Pressed"); } });
-   } }

Summary

In this session, you learnt to:

- Define the need for event handling
- Describe Java's event handling mechanism – The Delegation Event Model
- Define an Event Source
- Define an Event Listener
- Describe event classes that encapsulate various kinds of events
- Describe the process of a listener registering with an event source
- Describe the need for an event listener to implement the relevant listener interface/s for handling specific event/s
- Write applets that handle events
- Define adaptor classes, inner classes, and anonymous inner classes

CHAPTER 9

I/O Streams

Objectives

In this session, you will learn to:

- Define input and output streams in Java
- Define **Byte** streams and **Character** streams
- Define the predefined stream objects defined in the **System** class, namely **in**, **out**, and **err**
- Highlight the preference for **Character** over **Byte** streams
- Describe the need for stream wrapping or constructor wrapping
- Define serialization
- Implement object serialization with the help of the **ObjectInputStream** and the **ObjectOutputStream**

Streams

- Java programs perform I/O through streams. A stream is:
  - an abstraction that either produces or consumes information
  - linked to a physical device by the Java I/O system
- All streams behave similarly, even if the actual physical devices to which they are linked differ
- Thus the same I/O classes can be applied to any kind of device as they abstract the difference between different I/O devices

Streams

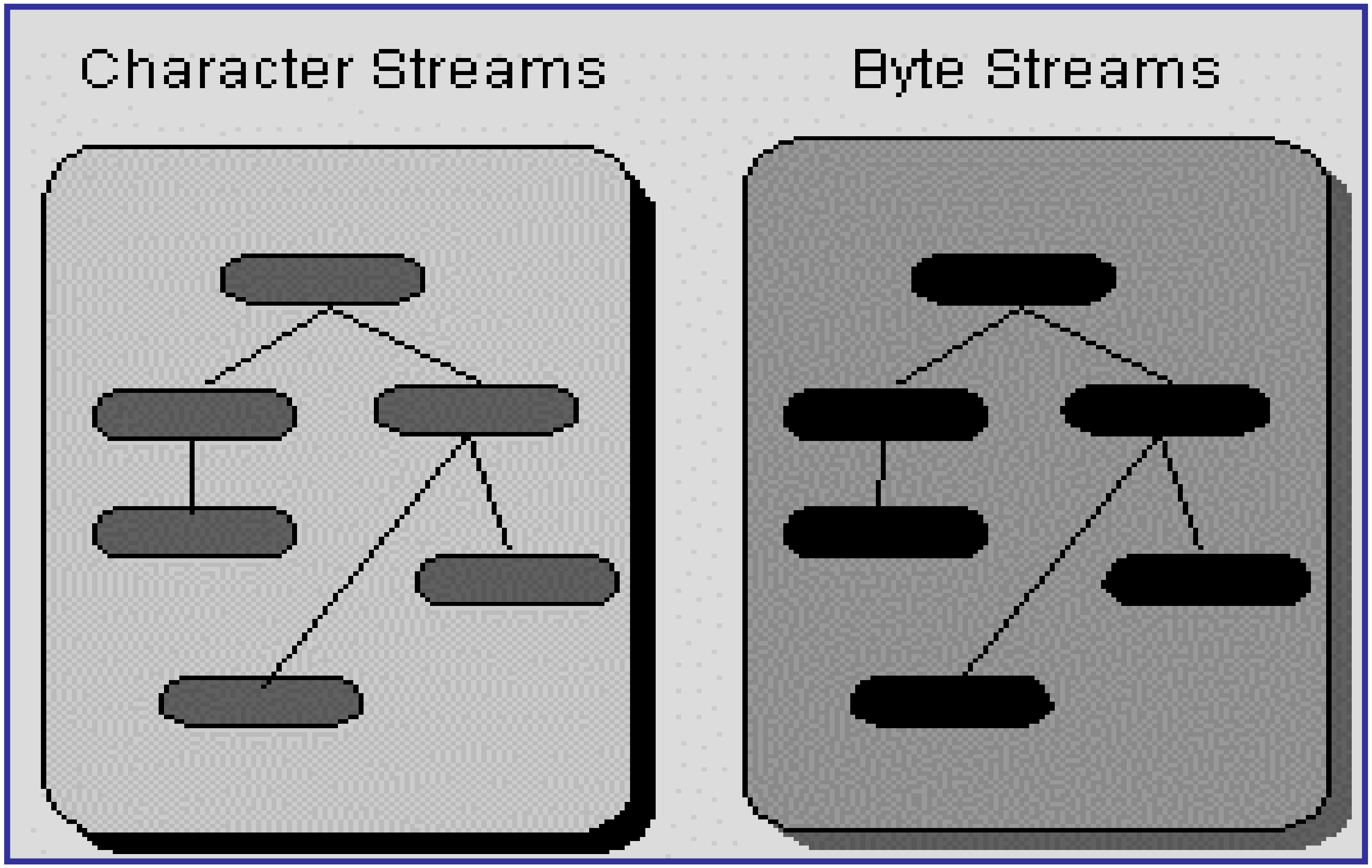
- Java's stream classes are defined in the **java.io** package
- Java 2 defines two types of streams:
  - **byte streams**
  - **character streams**
- Byte streams:
  - provide a convenient means for handling input and output of bytes
  - are used for reading or writing binary data
- Character streams:
  - provide a convenient means for handling input and output of characters
  - use **Unicode**, and, therefore, can be internationalized

The Predefined Streams

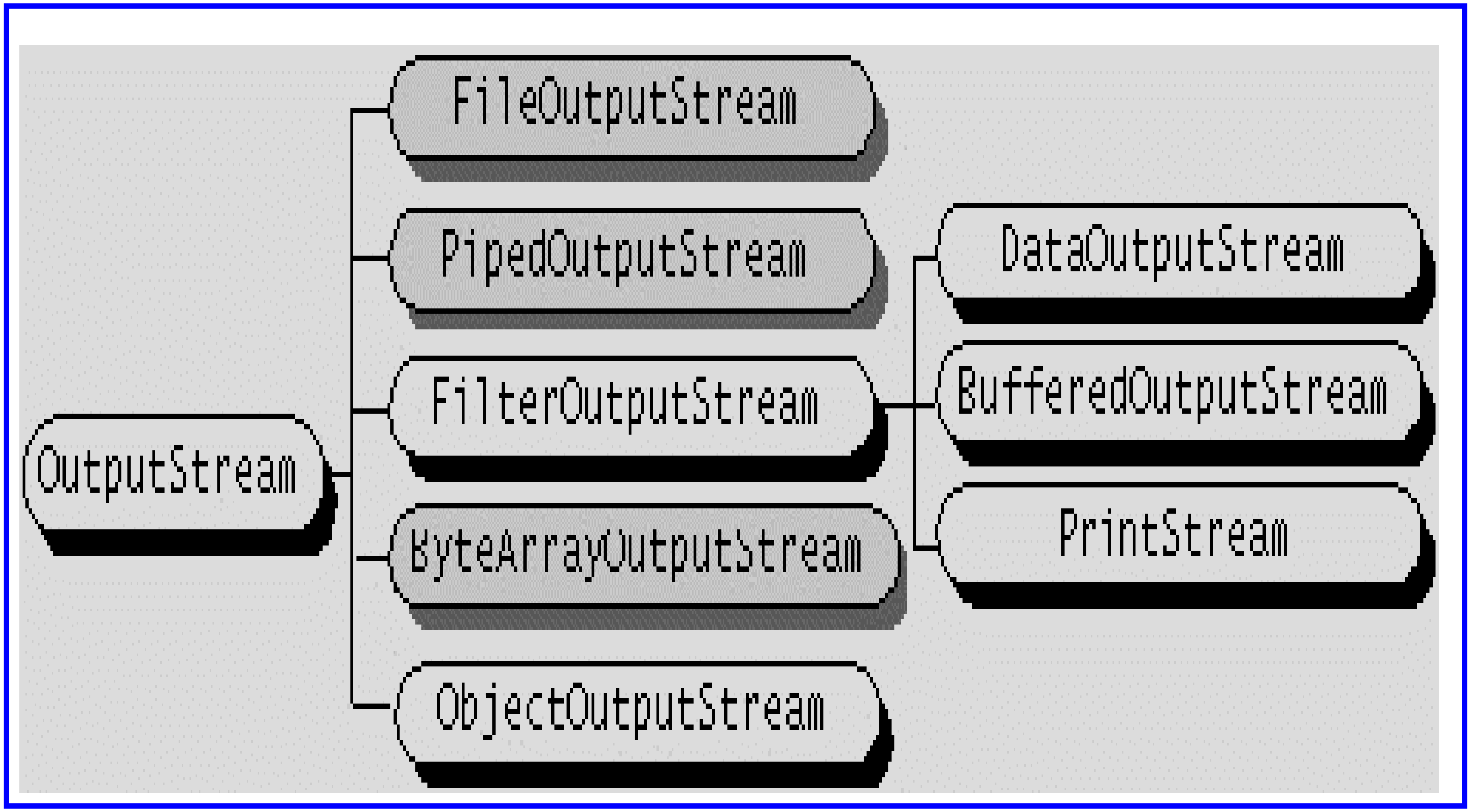
- **System** class of the java.lang package contains three predefined stream variables, **in**, **out**, and **err.**

- These variables are declared as **public** and **static** within **System**:
  - **System.out** refers to the standard output stream which is the console
  - **System.in** refers to standard input, which is the keyboard by default
  - **System.err** refers to the standard error stream, which also is the console by default

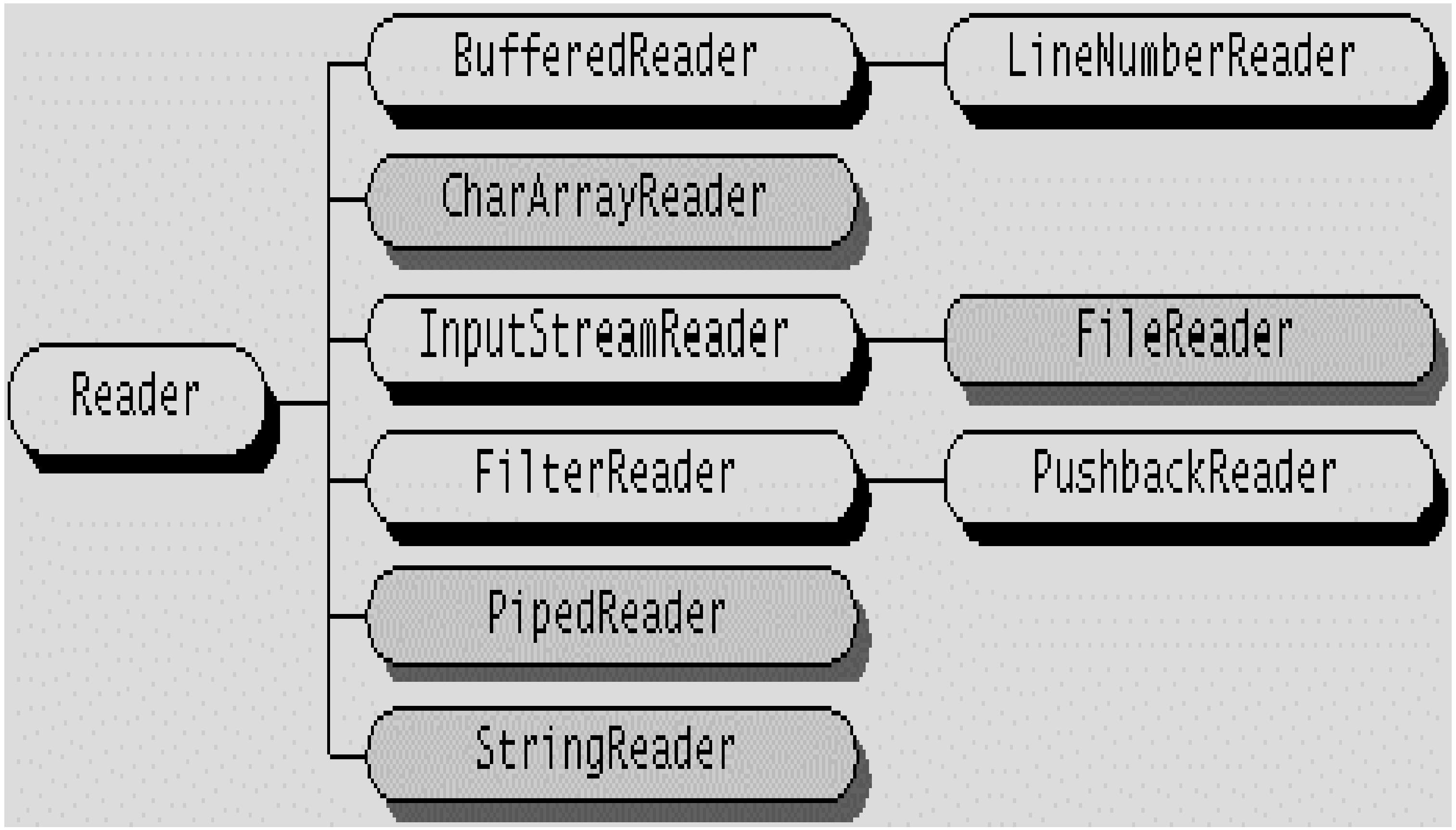
Streams



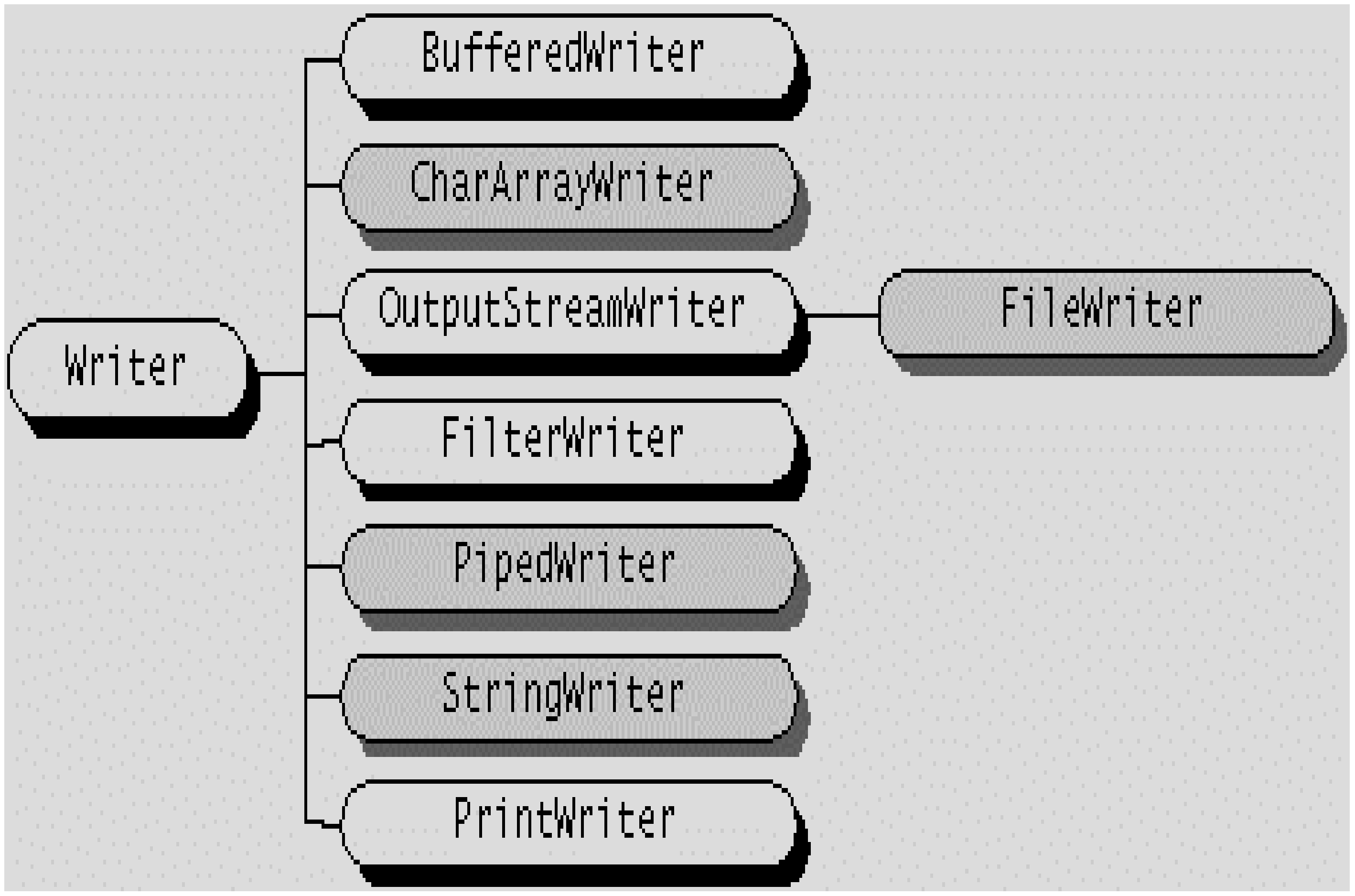
Byte Streams



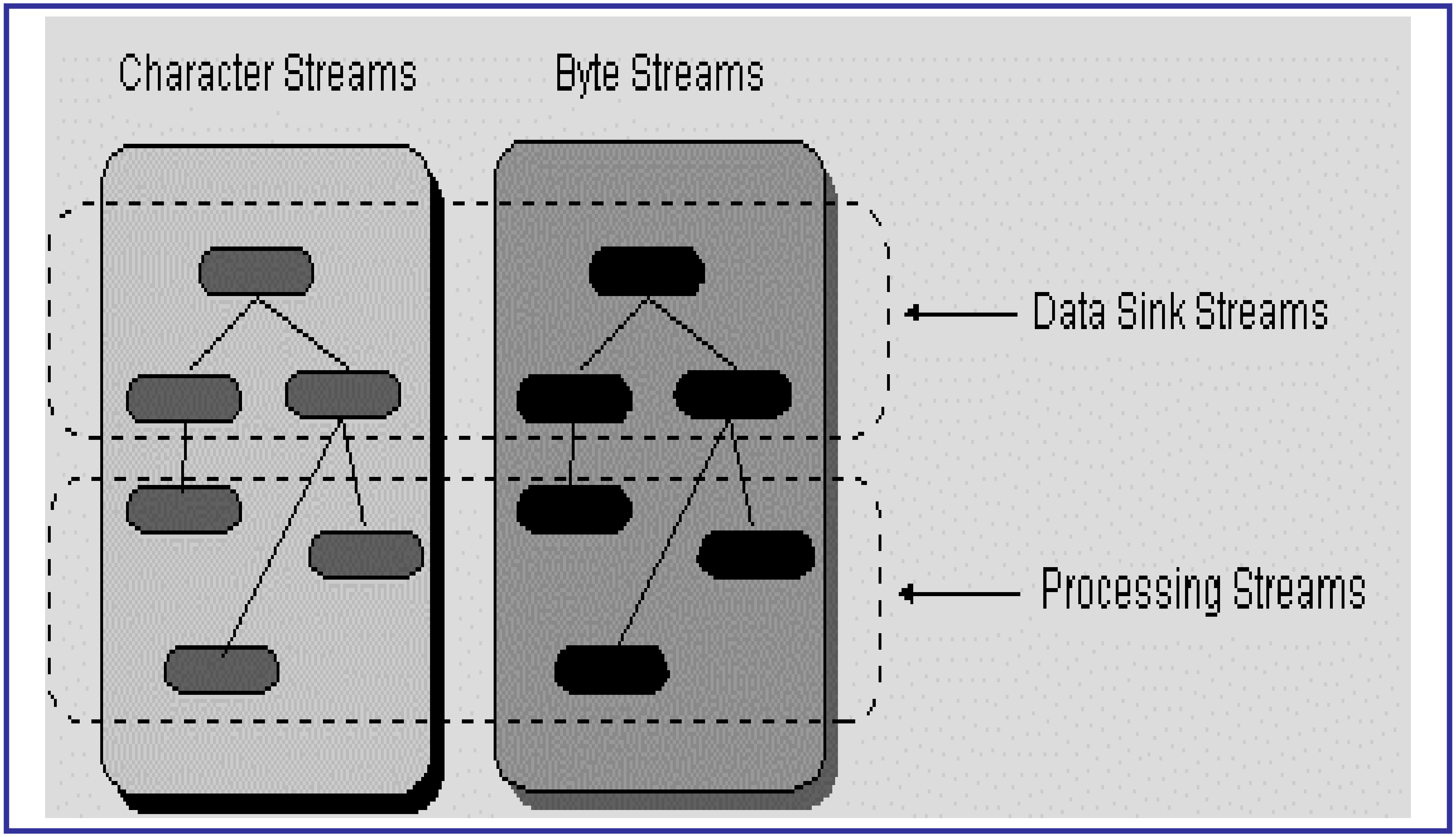
Character Streams



Character Streams



Streams


Character Streams
Byte Streams
Data Sink Streams
Processing Streams

Reading Console Input - Stream Wrapping

- The preferred method of reading console input in Java 2 is to:
  - use a character stream
  - **InputStreamReader** class acts as a bridge between byte and character streams
  - console input is accomplished by reading from **System.in**
  - To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream

Reading Characters

- //use a buffered reader to read characters from the console
- import java.io.*;
- class BRRead{
-   public static void main (String args[ ]) throws IOException
-    {
-      char c;
-      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
-      System.out.println(“Enter Characters, ‘q’ to quit”);
-      do {
-        c = (char) br.read( );
-        System.out.println(c);
-       }while (c != ‘q’);
-    }
- }

Reading Strings

```
//use a buffered reader to read strings from the console
import java.io.*;
class BRRead{
  public static void main (String args[ ]) throws IOException
   {
     String str;
     BufferedReader br = new BufferedReader(new
         InputStreamReader(System.in));
     System.out.println(“Enter Characters, ‘stop’ to quit”);
     do {
        str = br.readLine( );
        System.out.println ( str );
     }while (!str.equals( “stop”));
    }
}
```

Writing Console Output

- Console output is accomplished with **print( )** and **println( )**
- These methods are defined by the class **PrintStream**, which is the type of object referenced by **System.out**
- **System.out** is a byte stream used to write bytes
- Since **PrintStream** is an output stream derived from **OutputStream**, it implements the low-level method **write( )**.
- Thus, **write( )** can be used to write to the console.

Using FileReader and FileWriter

```
import java.io.*;
public class Copy {
      public static void main(String[] args) throws  IOException {
                File inputFile = new File(”Source.txt");
                File outputFile = new File(”Target.txt");
                FileReader in = new FileReader(inputFile);
                FileWriter out = new FileWriter(outputFile);
                int c;
                while ((c = in.read()) != -1)
                 out.write(c);
                in.close();
                out.close();
         }
}
```

Serializing Objects

- How to Write to an **ObjectOutputStream**

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

- How to Read from an **ObjectOutputStream**

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Object Serialization

```
class MyClass implements Serializable {
        String s;
        int i;
        double d;
        public MyClass(String s, int i, double d) {
                this.s = s;
                this.i = i;
                this.d = d;
        }
        public String toString() {
                return “s=“ + s + “; i=“ + i + “; d=“ + d;
        }
}
```

Object Serialization

- import java.io.*;
- public class SerializationDemo {
- public static void main(String args[]) {
- try {
-         MyClass object1 = new MyClass(“Hello”, -7, 2.7e10);
-         System.out.println(“object1; “ + object1);
-         FileOutputStream fos = new FileOutputStream(“serial”);
-         ObjectOutputStream oos = new ObjectOutputStream(fos);
-         oos.writeObject(object1);
-         oos.flush();
-         oos.close();
-         }
- catch(Exception e) {
-         System.out.println(“Exception during serialization: “ + e);
-         System.exit(0);
- }

Object Serialization

```
// Object Deserialization
 try { MyClass object2;
          FileInputStream fis = new FileInputStream(“serial”);
          ObjectInputStream ois = new ObjectInputSream(fis);
          object2 = (MyClass)ois.readObject();
          ois.close();
          System.out.println(“object2: “ + object2); }
catch(Exception e) {
          System.out.println(“Exception during deserialization: “ + e);
          System.exit(0);
}
}
}
```

Summary

- In this session, you learnt to:
- Define input and output streams in Java
- Define **Byte** streams and **Character** streams
- Define the predefined stream objects defined in the **System** class, namely **in**, **out**, and **err**
- Highlight the preference for **Character** over **Byte** streams
- Describe the need for stream wrapping or constructor wrapping
- Define serialization
- Implement object serialization with the help of the **ObjectInputStream** and the **ObjectOutputStream**

Day 8

CHAPTER 10

Swing

Objectives

At the end of this session, you will learn to:

- Describe the limitations of AWT
- Identify the need for creating platform-neutral GUI-based applications
- Describe the history, evolution and role of Swing in creating platform-neutral GUI-based applications
- Describe the Model-View-Controller framework on which Swing is based
- Describe various controls and layout managers in Swing
- Create a GUI-based application using Swing

Why Swing?

- AWT is not functional enough for full scale applications
  - the widget library is small
  - the widgets only have basic functionality
  - extensions commonly needed
- AWT Components rely on native peers:
  - **Look and Feel** tied to operating system
  - functionality determined by operating system
  - widgets won’t perform exactly the same on different platforms
  - faster, because the OS handles the work

Project JFC

- Sun began a project to beef up Java in 1996
- The result was the Java Foundation Classes released in 1998
- Swing is the new and improved GUI toolkit included in the JFC
- With Swing, Java can produce highly complex GUIs for industrial strength applications.
- Swing components are pure Java components no longer relying on their native peers and are there platform neutral

Why is Swing so Hot?

- Swing
  - uses **lightweight** components
  - uses a variant of the **Model View Controller Architecture** (**MVC**)
  - has **Pluggable Look And Feel** (**PLAF**)
  - uses the **Delegation Event Model**
  - components can have transparent portions
  - components can be any shape, and can overlap each other
  - **Look and Feel** drawn at runtime so can vary
  - functionality is the same on all platforms
  - though slower, preferred by the industry

Java Foundation Classes

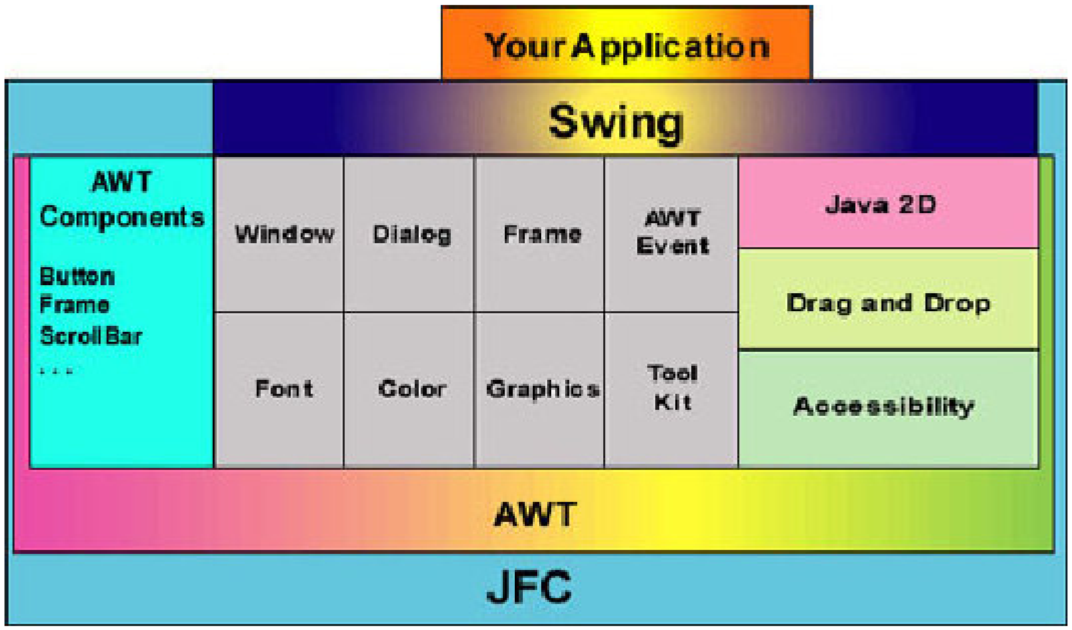
AWT

Swing

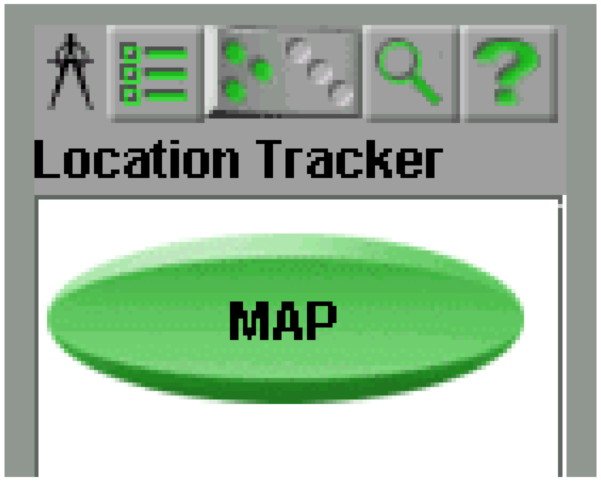
Accessibility

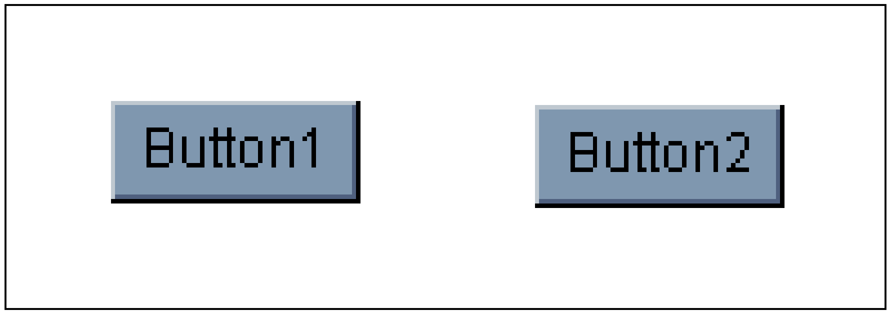
2D API

Drag and Drop



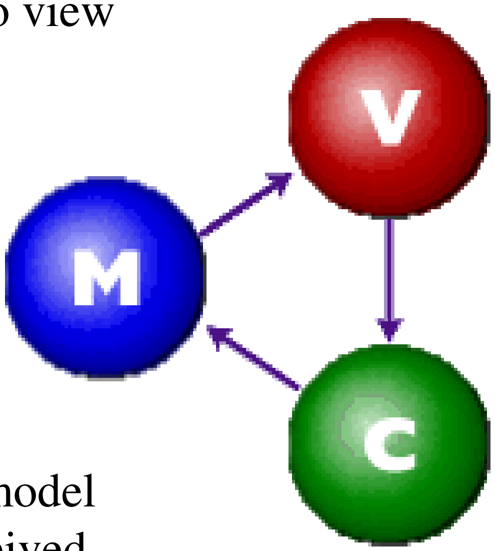
Lightweight vs Heavyweight





MVC Communication

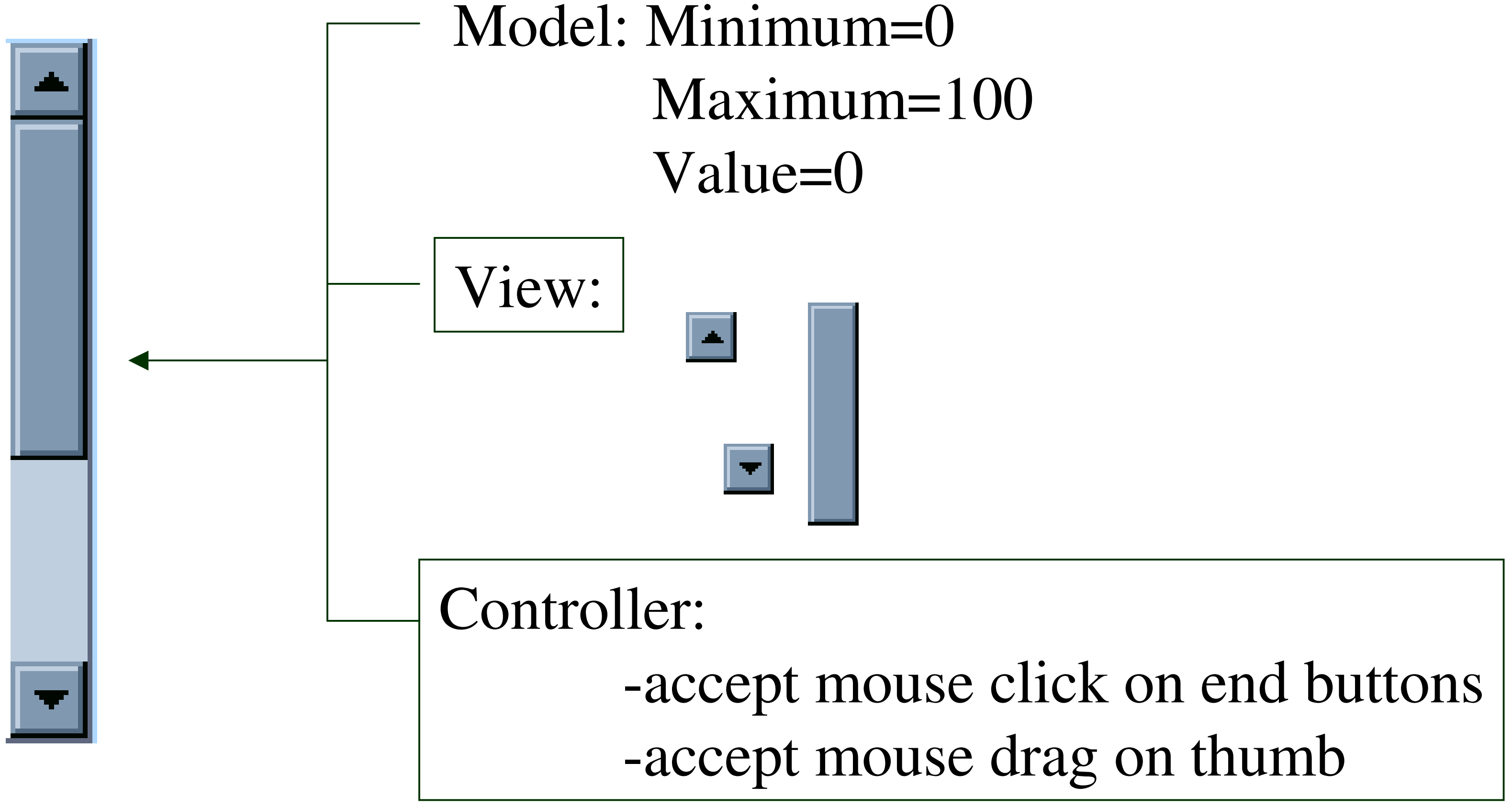
Model passes data to view for rendering



View determines which events are passed to controller

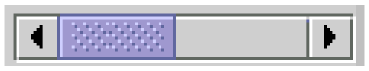
Controller updates model based on events received

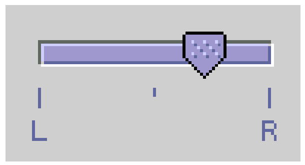
MVC Example


Model: Minimum=0
Maximum=100
Value=0
View:
Controller:
-accept mouse click on end buttons
-accept mouse drag on thumb

MVC in Java

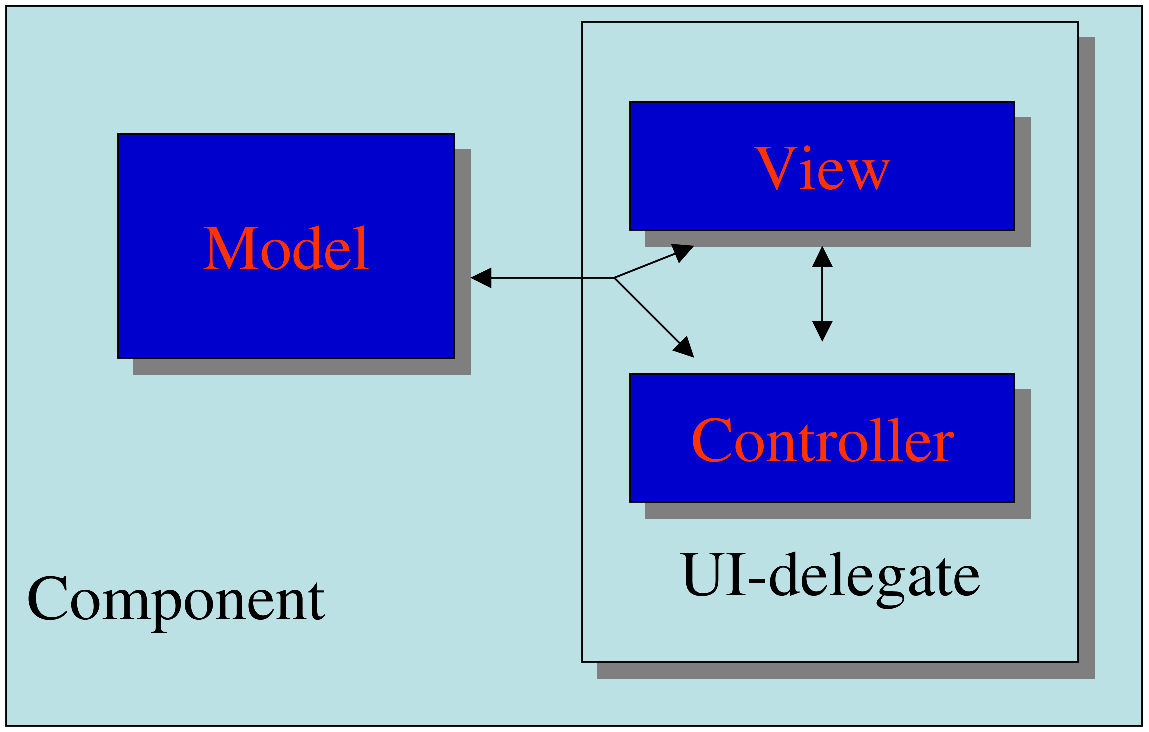
- Swing uses the **model-delegate** design, a similar architecture to **MVC**
- The View and Controller elements are combined into the **UI delegate** since Java handles most events in the AWT anyway
- Multiple views can be used with a single model
- Changes to a single Model can affect different views


JScrollBar



JSlider

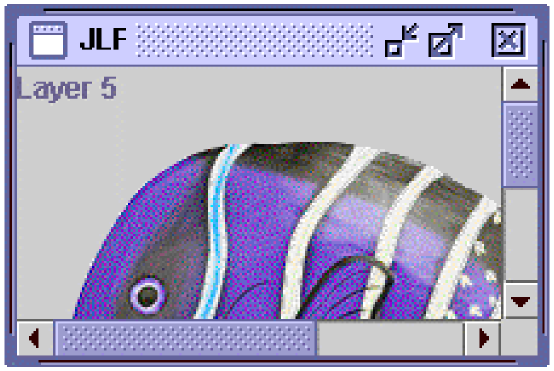
MVC in Java



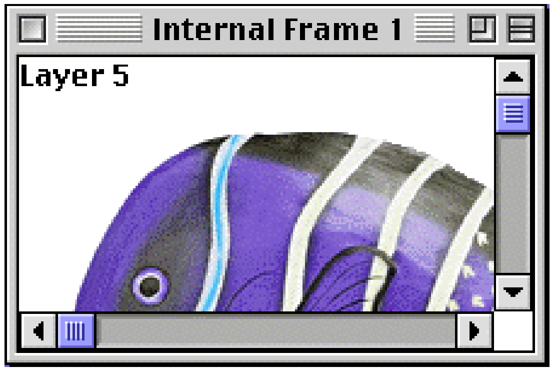
PLAF Features in Swing

- The PLAF:
  - has a default **Metal** style
  - can emulate **Motif**, and **Windows** styles
  - supports **Mac** style through download
  - can be used to design new styles
  - can be reset at runtime

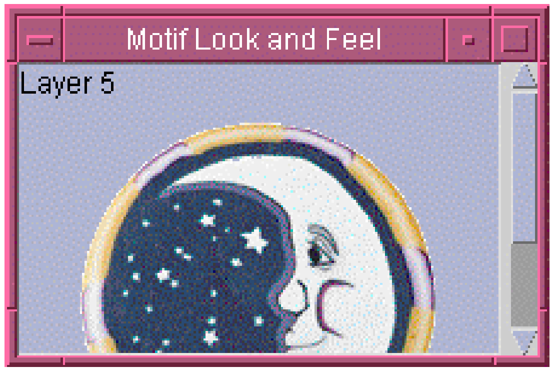
PLAF examples



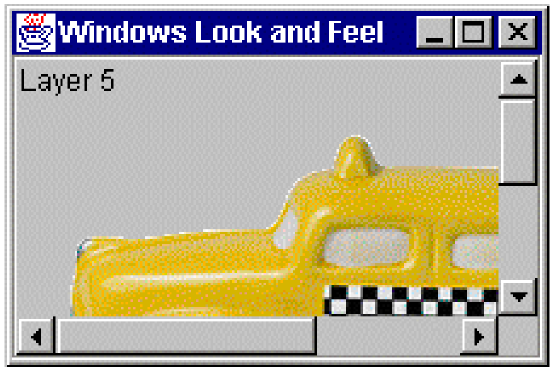
Java Look and Feel



MacOS Look and Feel

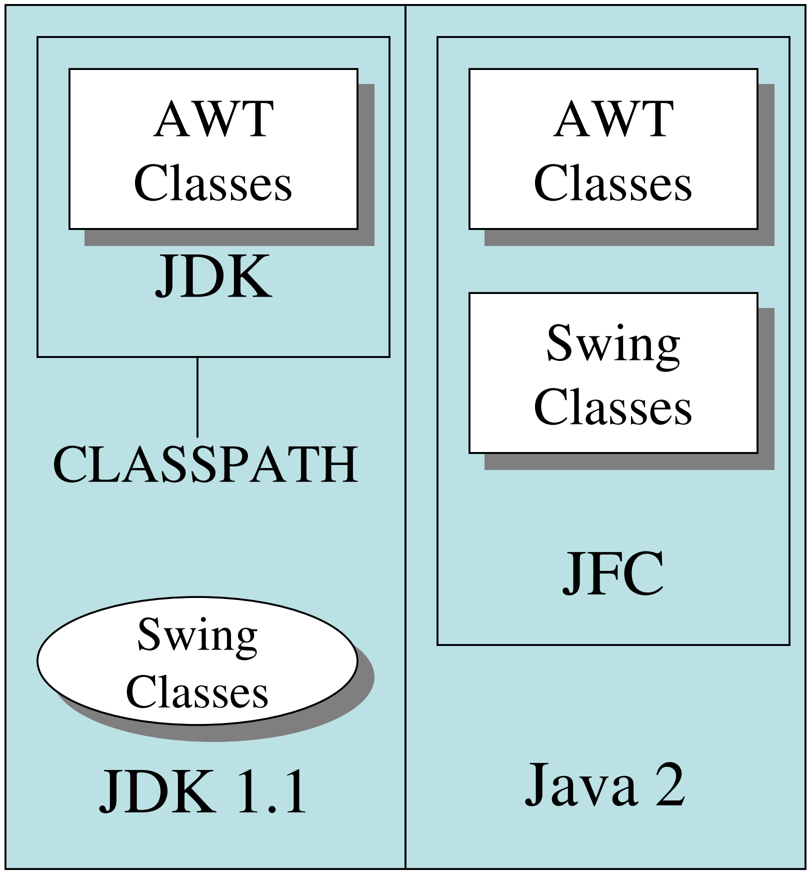


Motif Look and Feel

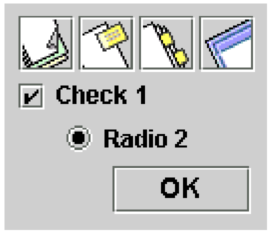


Windows Look and Feel

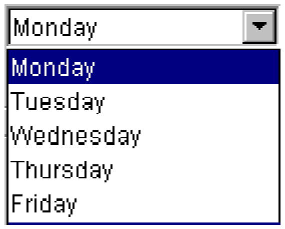
How do I Use Swing?



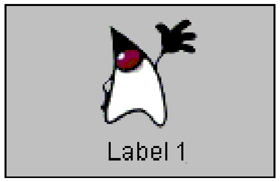
Useful Components



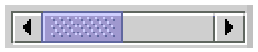
JButton



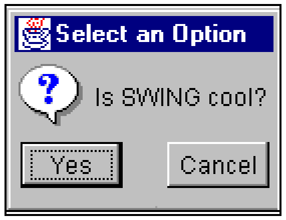
JComboBox



JLabel

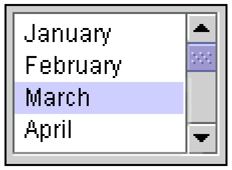


JScrollBar

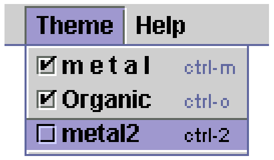


Is SWING cool?

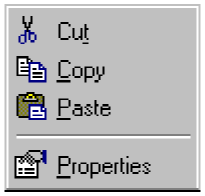
JDialog



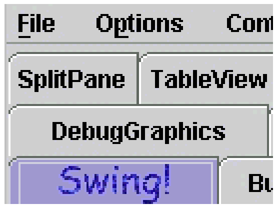
JList



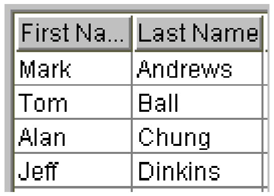
JMenu



JPopupMenu

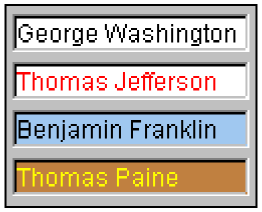


JTabbedPane

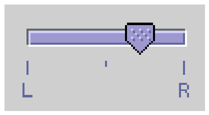


First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

JTable



JTextField



JSlider

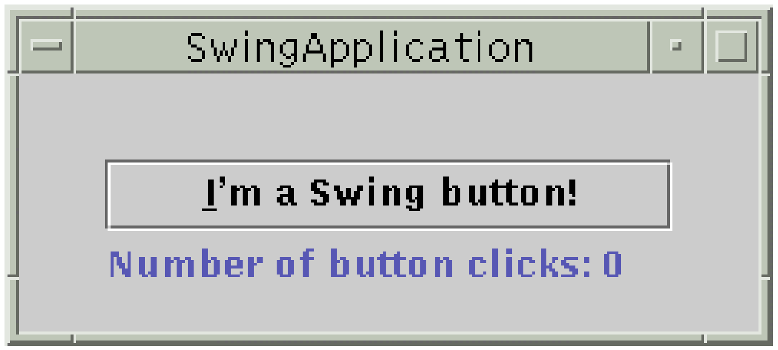
Methods Inherited from Component Class

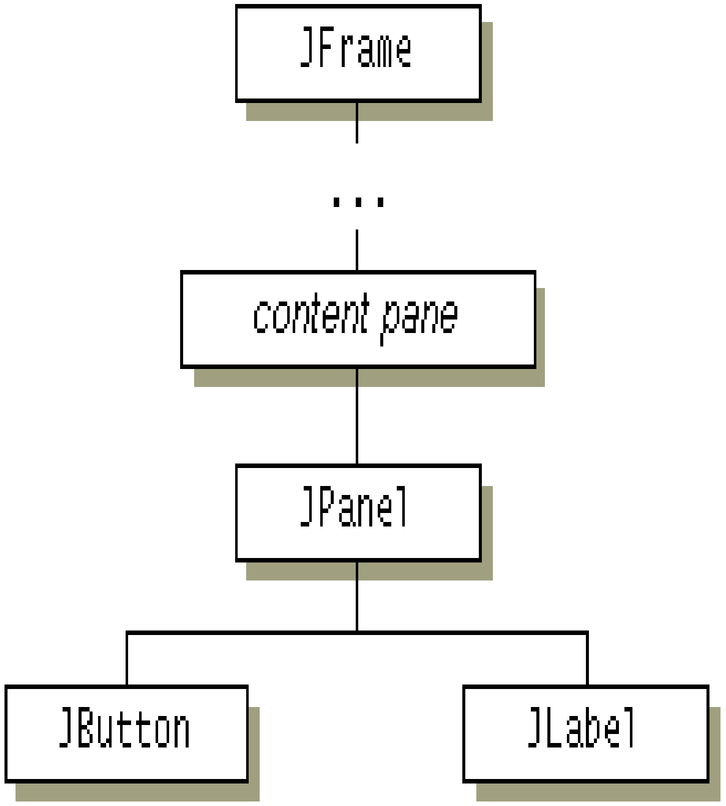
- get/setBackground()
- is/setEnabled()
- get/setFont()
- get/setForeground()
- get/setLayout()
- get/setLocationOnScreen()
- get/setName()
- get/setParent()
- is/setVisible()

New Methods

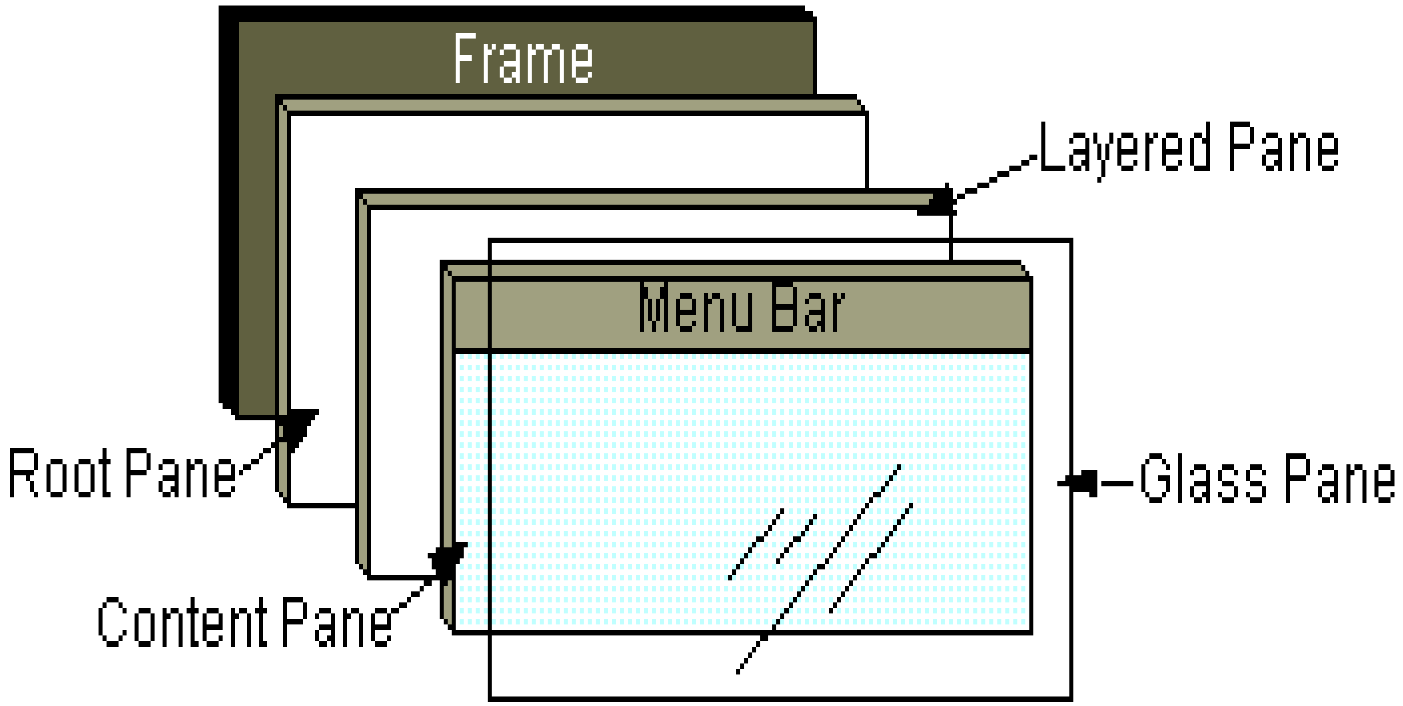
- get/setBounds()
- get/setSize()
- get/setLocation()
- get/setWidth()
- get/setHeight()
- get/setMaximumSize()
- get/setMinimumSize()
- get/setPreferredSize()
- get/setBorder()
- is/setDoubleBuffered()
- getGraphics()
- get/setToolTipText()
- add()
- remove()
- pack()

Swing Component and Containment Hierarchy





Root Panes


Frame
Layered Pane
Menu Bar
Root Pane
Glass Pane
Content Pane

Layout Managers

- arrange widgets according to a pattern
- can update containers to handle resizing of the container or internal widgets
- make complex UIs possible

Default Layout Managers

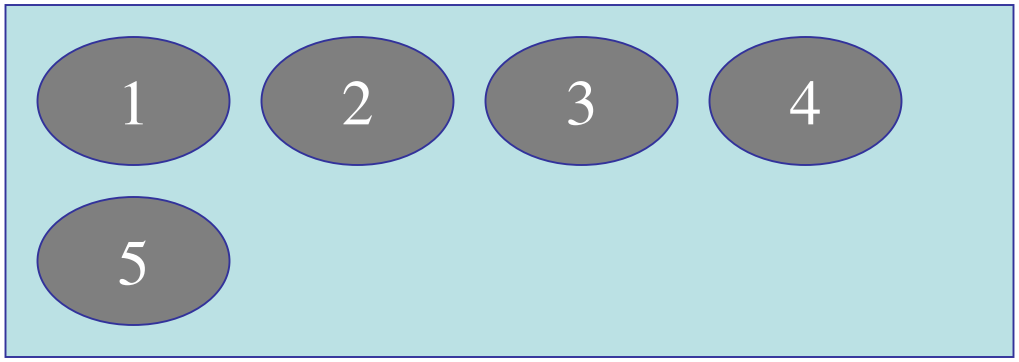
In AWT, the default layout for applets was FlowLayout.

Container	Layout Manager
JApplet	BorderLayout (on its content pane)
JBox	BoxLayout
JDialog	BorderLayout (on its content pane)
JFrame	BorderLayout (on its content pane)
JPanel	FlowLayout
JWindow	BorderLayout (on its content pane)

Top-level windows use BorderLayout

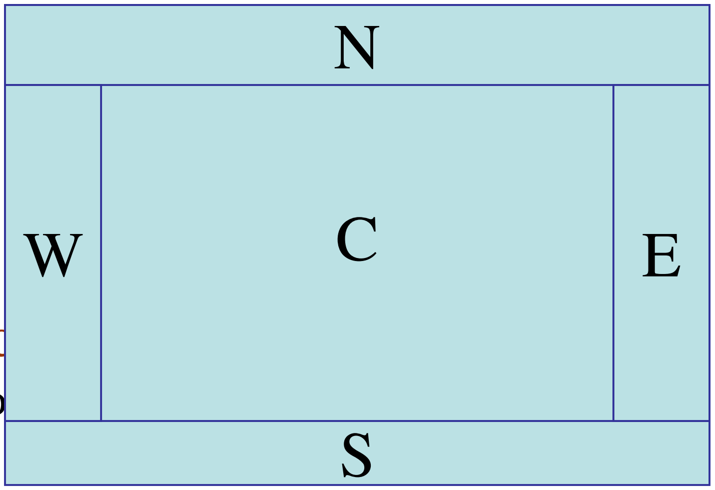
Flow Layout

- Arranges components from left to right and top to bottom
- Fits as many components in a row as possible before making a new row
- lets you specify alignment, horizontal and vertical spacing



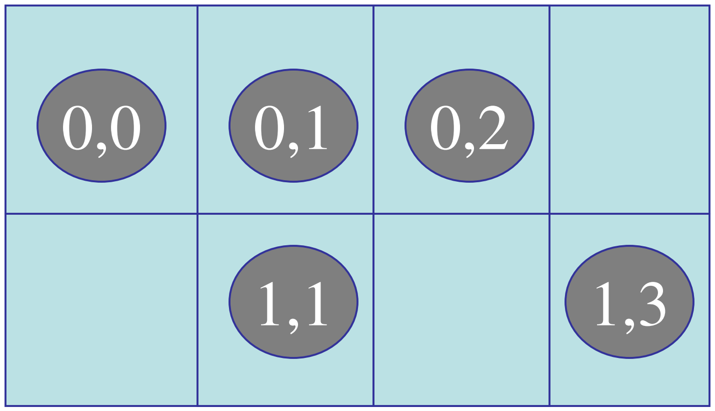
Border Layout

- Arranges components according to specified edges or the middle
  - NORTH
  - SOUTH
  - EAST
  - WEST
  - CENTER
- lets you specify horizontal and vertic
  - contentPanel.setLayout(new BorderLayo
  - contentPanel.add("Center", oPanel);
  - contentPanel.add("South", controlPanel);



Grid Layout

- Arranges components in a grid with specified rows and columns
- rows have same height and columns have same width



```
contentPanel.setLayout(new GridLayout(2, 4));
contentPanel.add(startButton);
contentPanel.add(stopButton);
```

Workshop

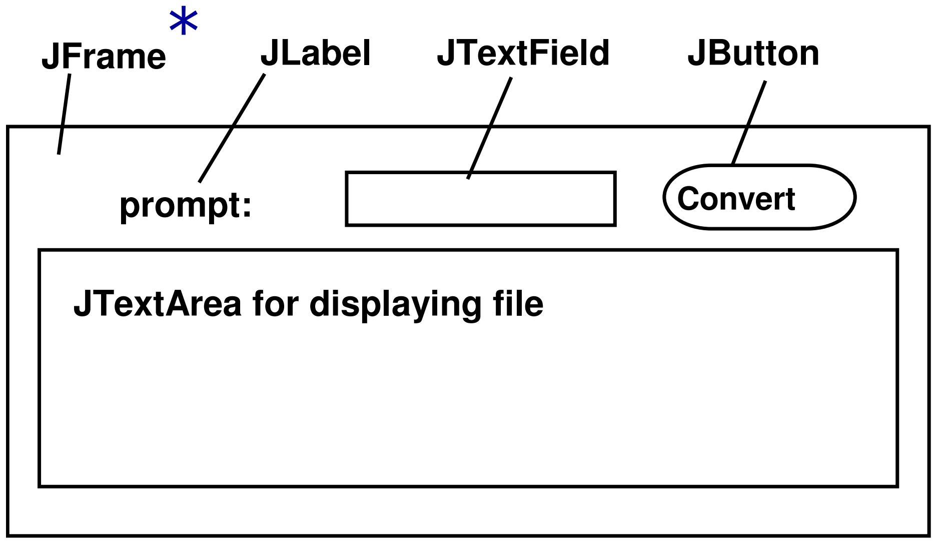
Case Study: Designing a Basic GUI

- Basic User Interface Tasks:
  - Provide help/guidance to the user.
  - Allow input of information.
  - Allow output of information.
  - Control interaction between the user and device.

- Problem Description: Design a GUI for a Java application that converts miles to kilometers. Write the class that performs the conversions.

  Swing objects for:
  - **Guidance**
  - **Input**
  - **Output**
  - **Control**

GUI Design: Layout



Containment Hierarchy **

JFrame
- Prompt JLabel
- Input JTextField
- Display JTextArea
- Convert JButton

Implementing the Converter Class

```
import javax.swing.*;  import java.awt.*;  port java.awt.event.*;
public class Converter extends JFrame implements ActionListener{
   private JLabel prompt = new JLabel("Distance in miles: ");
   private JTextField input = new JTextField(6);
   private JTextArea display = new JTextArea(10,20);
   private JButton convert = new JButton("Convert!");
   public Converter() {
      getContentPane().setLayout(new FlowLayout());
      getContentPane().add(prompt);
      getContentPane().add(input);
      getContentPane().add(convert);
      getContentPane().add(display);
      display.setLineWrap(true);
      display.setEditable(false);
      convert.addActionListener(this);   } // Converter()
   public void actionPerformed( ActionEvent e ) {
      double miles = Double.valueOf(input.getText()).doubleValue();
      double km = MetricConverter.milesToKm(miles);
      display.append(miles + " miles equals " + km + " kilometers\n"); } // actionPerformed()}
```

Instantiating the Top-Level JFrame

```
public static void main(String args[]) {
      Converter f = new Converter();
      f.setSize(400, 300);
      f.setVisible(true);
                                      // Quit the application
      f.addWindowListener(new WindowAdapter() {
          public void windowClosing(WindowEvent e) {
              System.exit(0);
          }
      });
} // main()
```

Summary

In this session, you learnt to:

- Describe the limitations of AWT
- Identify the need for creating platform-neutral GUI-based applications
- Describe the history, evolution and role of Swing in creating platform-neutral GUI-based applications
- Describe the Model-View-Controller framework on which Swing is based
- Describe various controls and layout managers in Swing
- Create a GUI-based application using Swing

Reference

1. Patrick Naughton and Herbert Schildt. Java 2: The Complete Reference
2. Bruce Eckel, *Thinking in Java*™, Prentice Hall
3. Deitel and Deitel, Java How to Program

Spirit of Wipro

Intensity to Win

- Make customers successful
- Team, Innovate, Excel

Act with Sensitivity

- Respect for the individual
- Thoughtful and responsible

Unyielding Integrity

- Delivering on commitments
- Honesty and fairness in action

WIPRO
Applying Thought