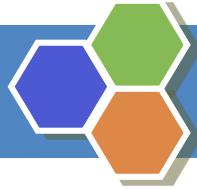


# Java SE

## Java





# History

- 1991 - Oak
- 1995 - Java
- Developers - James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan
- @ Sun Microsystems, Inc.





# Why Java ???

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic





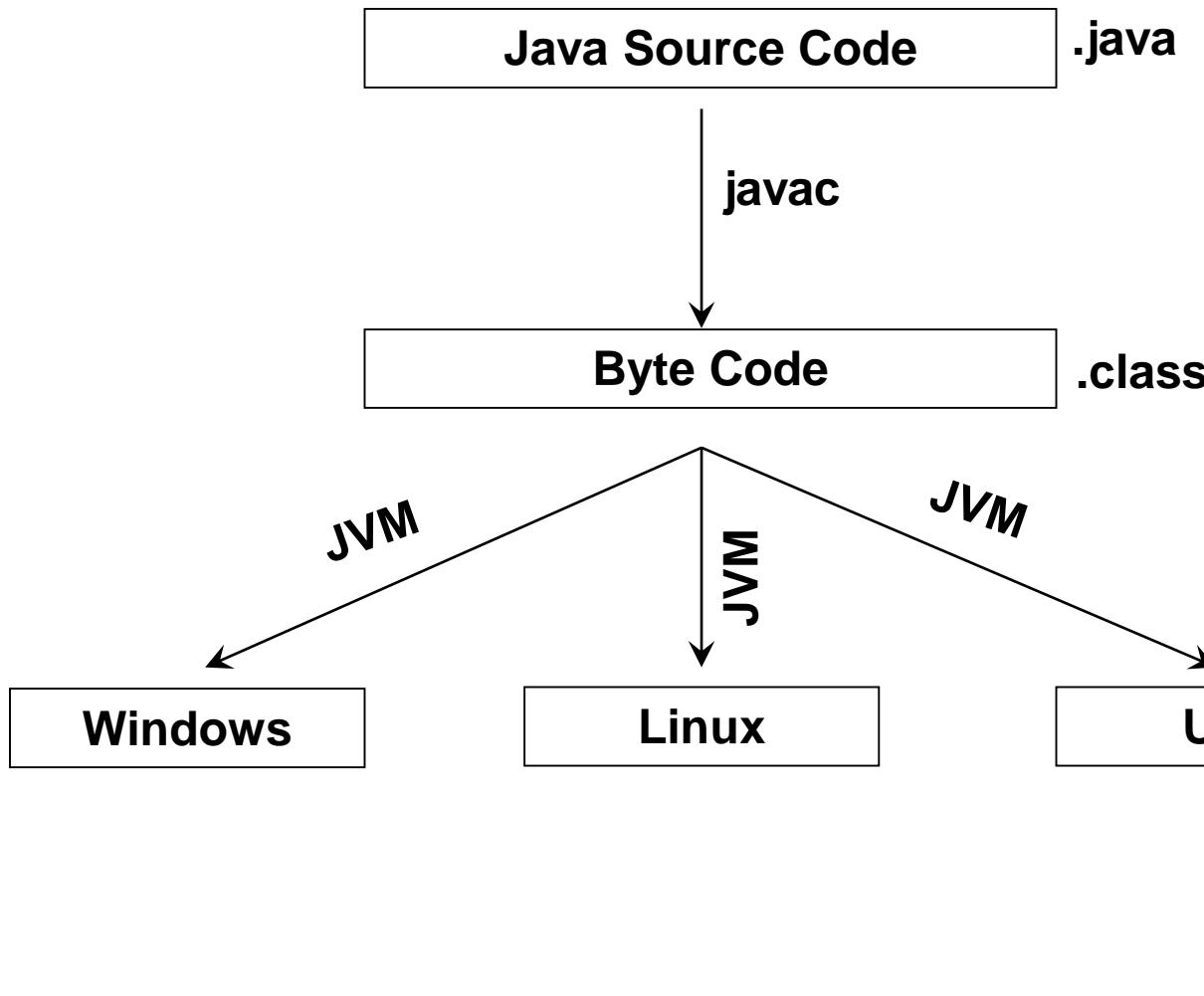
# Types of Java

- Java SE
  - Desktop application development
- Java EE
  - Web application development
- Java ME
  - Mobile application development





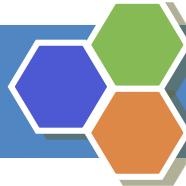
# How Java Works





- JDK – Java Development Kit
- JVM – Java Virtual Machine
- JRE – Java Runtime Environment
- API – Application Programming Interface

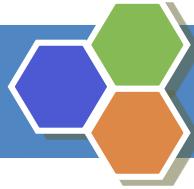




# Keyword

abstract	continue	goto	package	synchronized
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
cass	float	native	super	while
const	for	new	switch	

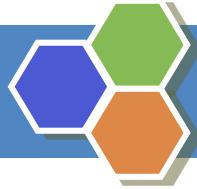




# First Java Program

```
public class FirstPrg {  
    public static void main (String[] args) {  
        System.out.println("My first JAVA  
program.");  
    }  
}
```





# Compile & Run Program

- Compile
  - javac FirstPrg.java
- Run
  - java FirstPrg

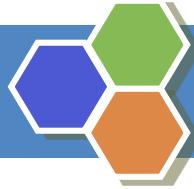




# Naming Conventions

- Class/Interface - ProperCase
- Variable/Object /Method - camelCase
- Package - lowercase
- Constant - UPPER\_CASE





# File Naming Rules & Comment

- Public class name MUST be filename
- Comment
  - ```
/* This is
 * Documentation
 * Comment
 */
```
  - `// Single line Comment`
  - `/* Multi
 Line
 Comment
 */`





# Java identifiers

- Class name
- Method name
- Variable name
- Identifier must start with alphabet, \$, \_

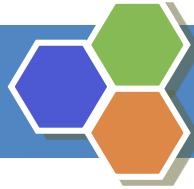




# Literal

- Character
  - 'A', '9', '\$'
- Numeric
  - 3, 4.5, -99.8
- String
  - "Hello", "55.5", "6.7\$"
- Logical
  - true, false





# Data types

| Type    | Byte | Range                                           |
|---------|------|-------------------------------------------------|
| byte    | 1    | $-2^7$ to $2^7-1$                               |
| short   | 2    | $-2^{15}$ to $2^{15}-1$                         |
| char    | 2    |                                                 |
| int     | 4    | $-2^{31}$ to $2^{31}-1$                         |
| float   | 4    | $-1.4 \times 10^{45}$ to $3.4 \times 10^{38}$   |
| long    | 8    | $-2^{63}$ to $2^{63}-1$                         |
| double  | 8    | $-4.9 \times 10^{324}$ to $1.8 \times 10^{308}$ |
| boolean |      | true / false                                    |

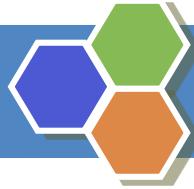




# Working with variable

- Variable declaration
  - Datatype varName;
- Value assignment
  - varName = value;
- JVM will not initialise local variables.
- Local variable's life span – block in which declared

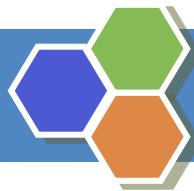




# Hands on System.out.println

```
public class SopClass {  
    public static void main(String[] args) {  
        int a=10, b=20;  
        System.out.println("a+b =" + a+b);  
        System.out.println(a + " + " + b + " = " + a+b);  
        System.out.println(a + b + " = " + a+b);  
    }  
}
```





# Operators

- Arithmetic (+, -, \*, /)
- Modulus (%)
- Assignment (=)
- Relational (>, >=, <, <=, ==, !=)
- Logical (&&, ||, !)
- Increment (++)
- Decrement (--)
- Binary (~, &, |, >>, <<, >>>)
- Ternary (? : )

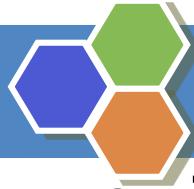




# Selection

- ```
if( condition) {  
    statement1;  
    statement2;  
}  
  
• if(condition)  
    statement;  
else  
    statement2;
```





## Selection contd...

- ```
switch(var) {  
    case const1 :      statement1;  
                        statement2;  
                        break;  
    case const2 :      statement3;  
                        statement4;  
                        break;  
  
    .....  
  
    .....  
    default : statement  
}
```





# Iteration / Loop

- ```
while (condition) {  
    statement1;  
    statement2;  
}
```
- ```
do {  
    statement1;  
    statement2;  
}while(condition);
```





# Iteration / Loop

- ```
for (initial value; condition; expression) {  
    statement1;  
    statement2;  
}
```
- ```
for (datatype var : array/collection) {  
    statement1;  
    statement2;  
}
```

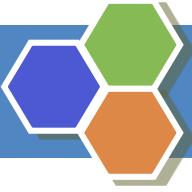




# Working with loops

- continue
- break





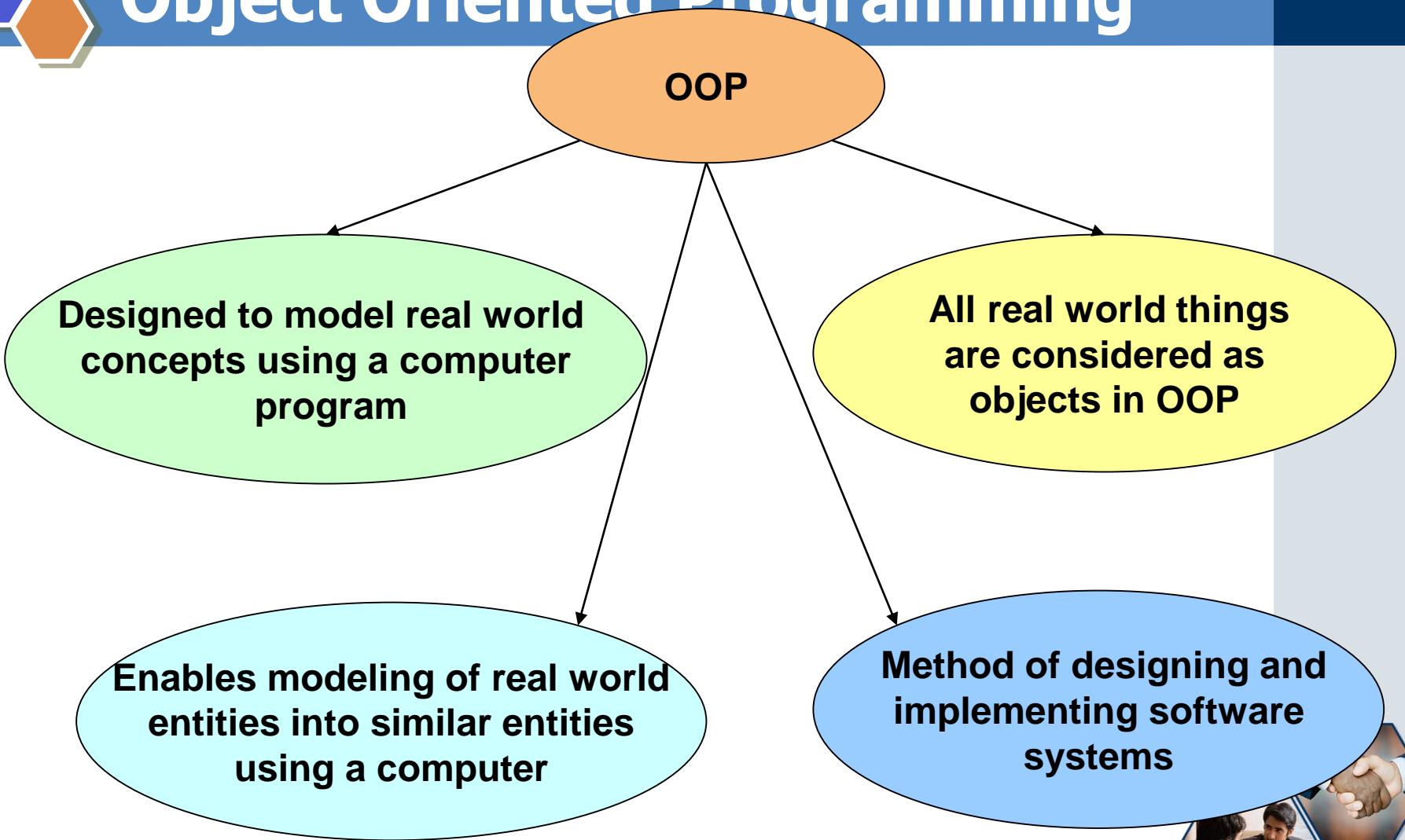
# OOP

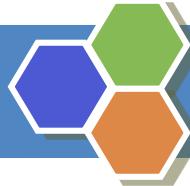
## Object Oriented Concepts



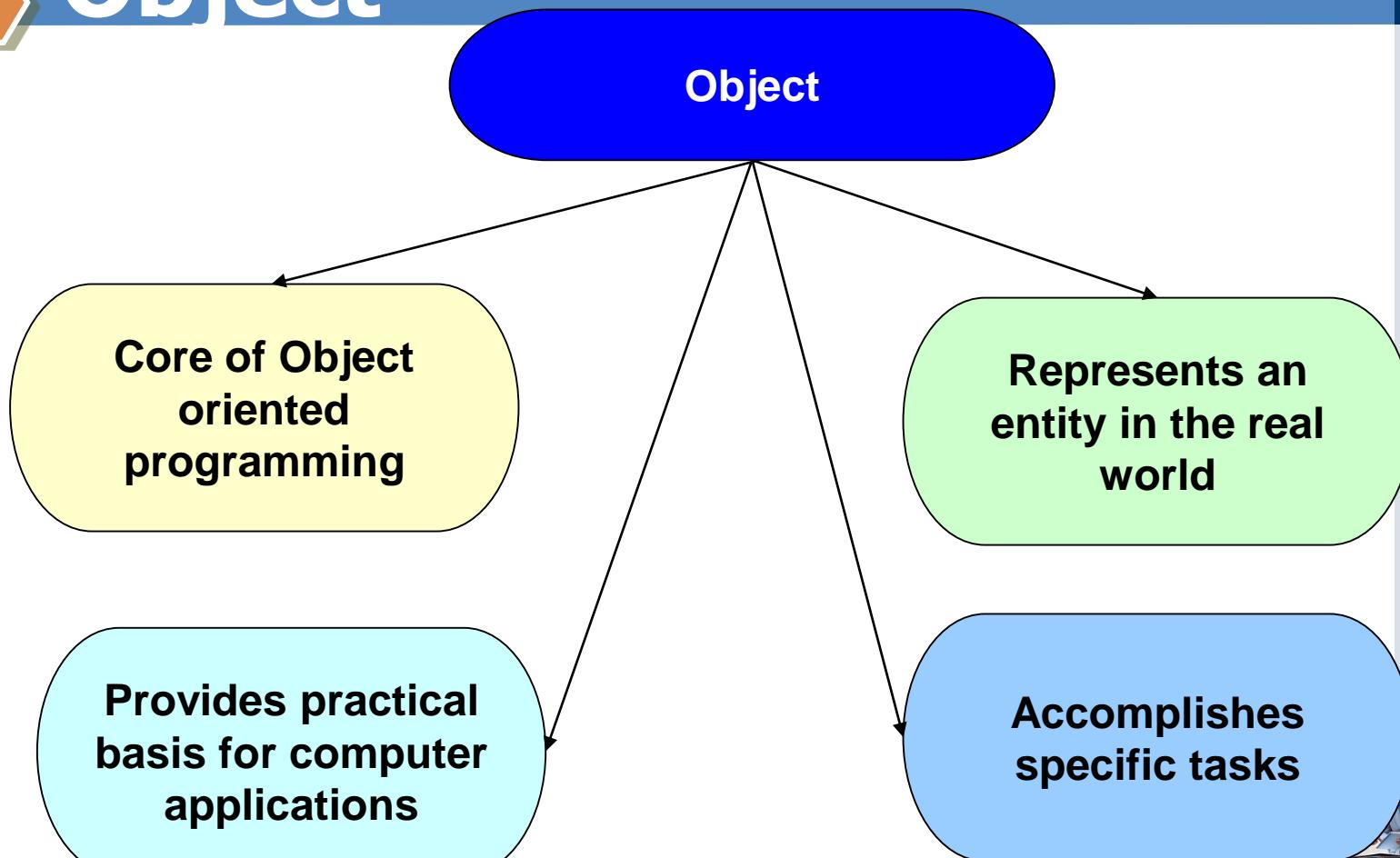


# Object Oriented Programming





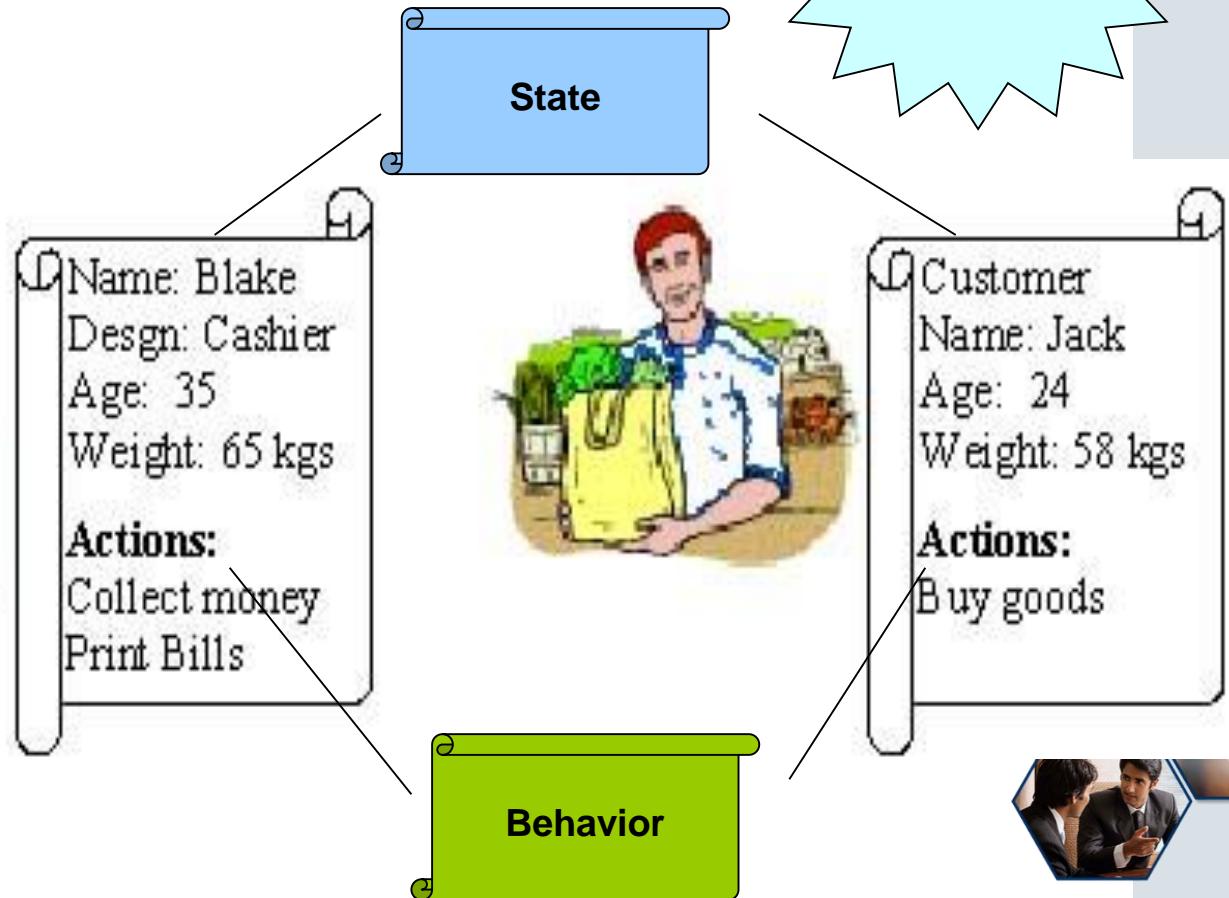
# Object

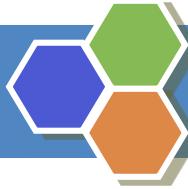


***“An object is a concrete entity that exists and which has a well defined state and behavior.”***



# Example of an Object





# Message Passing

## Message Passing

Objects communicate with each other by passing messages

When a particular operation is to be performed, it is requested for by sending a message to the object for which the operation is define

**"A *message* is a request sent by one object to another object to carry out certain actions."**





# Class

- ❑ A Class defines an entity in terms of common characteristics and actions.
- ❑ Class is a mechanism used to group properties of actions common to various objects.

## Examples of Classes

Class of Cars

Class of Shapes

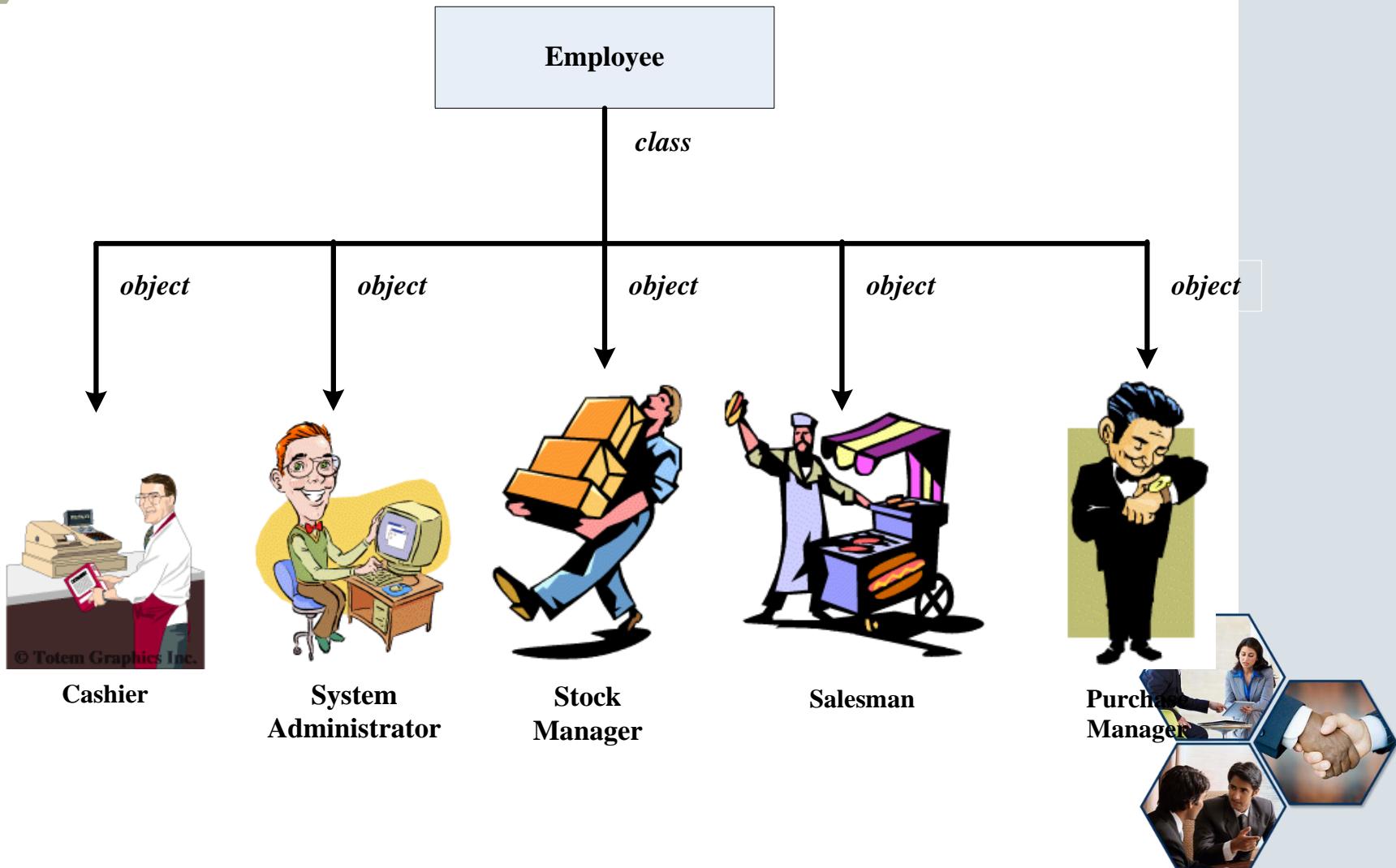
Class of Animals

**“A class is a blueprint for a group of objects that have common properties and behavior.”**





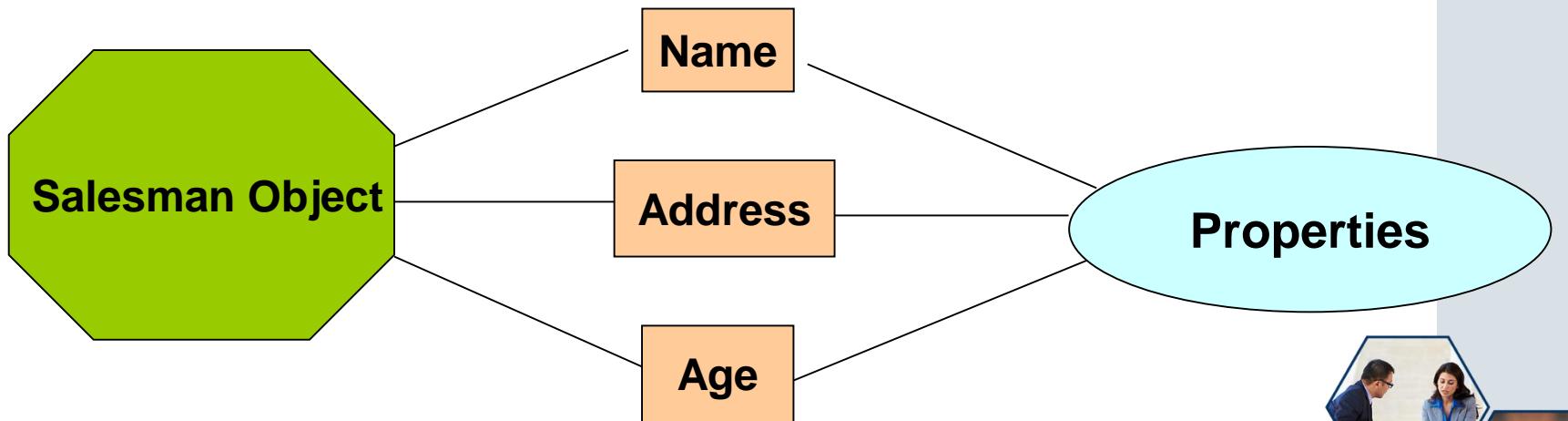
# Example of Class and Object





# Properties or Attributes

- Characteristics of objects represented as variables in a class.
- Each object has its own value for each of its properties.
- The property names are shared by all instances of a class.



**“A characteristic possessed by an object or entity when represented in a class is called a *property*.”**





# Methods

## Methods

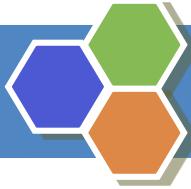
Actual implementation of an operation

Methods specify the manner in which the data of an object is manipulated

Specification of how the requested operation is carried out

**“An action performed by an object is known as a *method.*”**





# Difference between Class and Objects

## Class and Objects

**Class defines an entity**

**Object is the actual entity**

**Class is a conceptual model that defines all the characteristics and actions required of an object**

**All objects belonging to the same class have the same characteristics and actions**



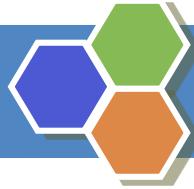


# Encapsulation

- Information Hiding
- Hiding implementation details of an object from its user
- Packing things together and presenting them in their new integrated form.
- For example, two or more chemicals form a capsule
- Packing the methods and attributes together in a single unit.
- Units are implemented in the form of classes

**“The process of hiding attributes, methods or details of implementation is called *Encapsulation*.”**





# Abstraction

**Technique of dealing  
with the  
complexity of an  
object.**

**Focussing only on the  
essential details  
and overlooking the  
non-essential  
details of an object.**





# Data Abstraction

- Concept of abstraction applied to the attributes (data) of a class.
- Implemented in programming languages using Abstract Data Types (ADT).

**“The process of identifying and grouping attributes and actions related to a particular entity as relevant to the application is *Data Abstraction*. ”**





# Implementing Classes in Java

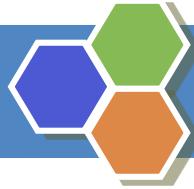
## □ Syntax

```
class <classname> {  
    <body of the class>  
}
```

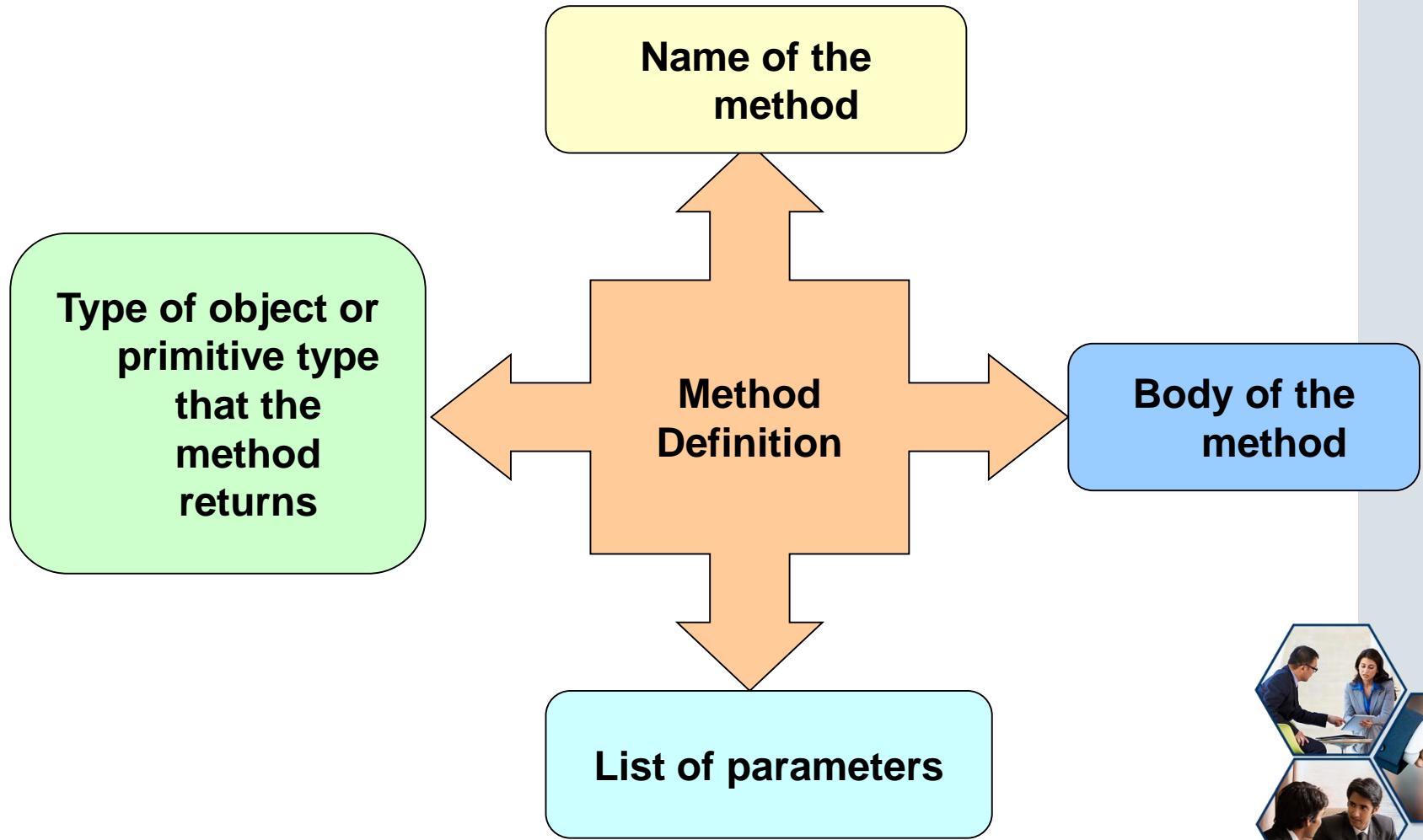
*where,*

**class** is the keyword used for creating a class,  
**<classname>** is the name of the class, and  
**<body of the class>** consists of declaration of  
attributes and methods.





# Methods in classes





# Methods in classes

## Syntax

```
<returntype> <methodname> (<type1> <arg1>, <type2> <arg3,...> {  
    <set of statements>  
}
```

*where,*

**returntype** is the data type of the value returned by the method,  
**<methodname>** is the user-defined name of the method, and method's  
parameter list is a set of variable declarations.





# Methods in Classes

- Methods are accessed using dot notation.
- Object whose method is called is on the left of the dot, while the name of the method is on the right.
- For Example,

```
Obj.isAvailable();
```

- Java provides class methods, which are similar to instance methods.
- Class method declaration is preceded with a static keyword.





# this Keyword

- Used inside any instance method to refer to the current object.
- The value of `this` refers to the object on which the current method has been called.
- The `this` keyword can be used where a reference to an object of the current class type is required.

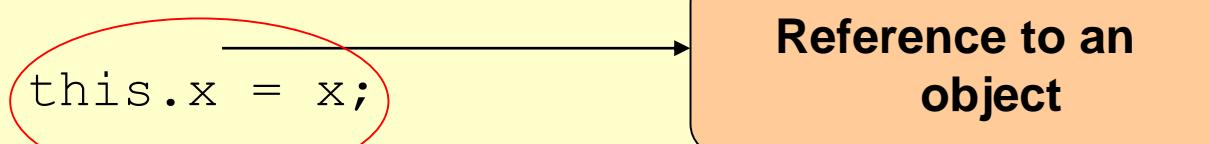


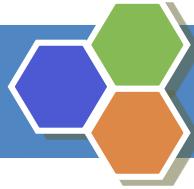


# Example of this Keyword

```
class Bank {  
    int x,y;  
    void init (int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}  
  
public static void main (String args[])  
{  
    Bank p = new Bank();  
    p.init (4,3);  
}
```

The program initializes x = 4 and y = 3.





# Constructors

- Method invoked whenever an instance of a given class is created.
- Same name as the class and no return type.
- Java allocates memory for the object, initializes the instance variables and calls the constructor methods.
- Two types of constructors
  - Parameterized constructors
  - Implicit constructors





## Example of Parameterized Ctor

```
Sdate(int m,int d,int y) {  
    month=m;  
    day=d;  
    year=y;  
    System.out.println("The Date is " + m + "/" + d  
    + "/" + y + ".");  
}  
  
public static void main(String args[])  
{  
    Sdate S1,S2;  
    S1=new Sdate(11,27,1969);  
    S2=new Sdate(3,3,1973);  
}
```

Parameterized  
Constructor



# Example of Implicit Constructors

```
Sdate ()  
{  
    month=11;  
  
    day=27;  
  
    year=1969;  
}  
  
public static void main(String args[])  
{  
    Sdate S1,S2;  
  
    S1=new Sdate();  
  
    S2=new Sdate();
```



## Implicit Constructors





# Analyzing the Program

- The symbol `/* */` indicates that the statements that follow are comments in that program. The multi line comment begins with `/*` and ends with `*/`. A single line comment begins with a `//` and ends at the end of the line.
- The keyword `class` declares the definition of the class. It also helps the compiler to understand that it is a class declaration.
- The entire class definition, including all its members, is done within the open `{ }` and closed `{ }` curly braces. This marks the beginning and end of the class definition block.

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





# Analyzing the Program

- The program execution begins from `main()` method.
- The `public` keyword is an access specifier, which controls the visibility and scope of class members.
- The `static` keyword allows the `main()` method to be called, without needing to create an instance of the class.
- The `void` keyword tells the compiler that the `main()` method does not return any value when it is executed.





# Analyzing the Program 3-3

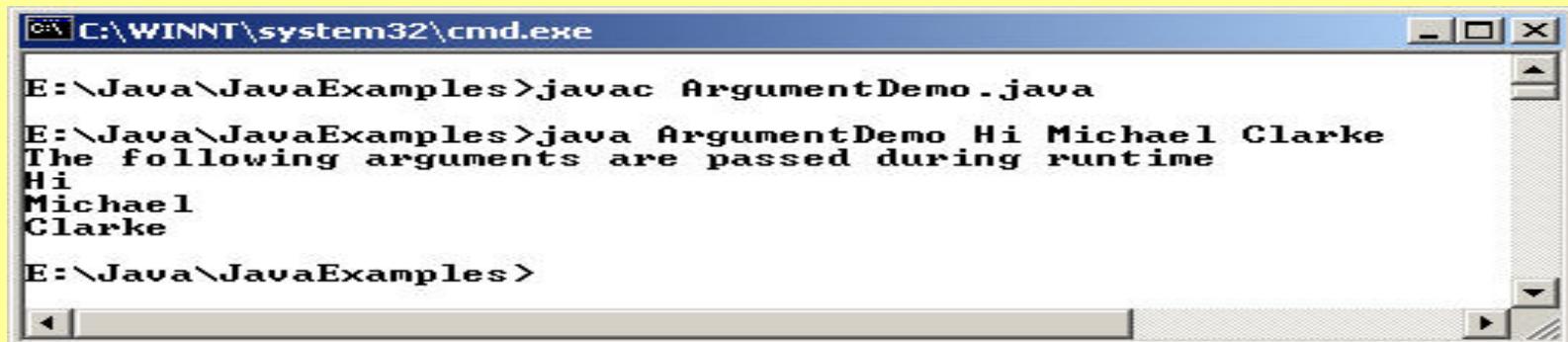
- The `main()` method is the starting point for all Java applications.
- `Args[]` is an array of type `String`.
- The `println()` method displays the string that is passed to it as an argument with the help of `System.out`.





# Command-Line Argument

```
class ArgumentDemo {  
    public static void main(String[] args) {  
        System.out.println("The following arguments are  
passed during runtime");  
        /* Passing the command line arguments */  
        System.out.println(args[0]);  
        System.out.println(args[1]);  
        System.out.println(args[2]);  
    }  
}
```



The screenshot shows a Windows command-line interface (cmd.exe) window. The title bar reads "C:\WINNT\system32\cmd.exe". The command line shows the user navigating to "E:\Java\JavaExamples" and running "javac ArgumentDemo.java". After compilation, the user runs "java ArgumentDemo Hi Michael Clarke". The application's output is displayed below, showing the message "The following arguments are passed during runtime" followed by the arguments "Hi", "Michael", and "Clarke".

```
E:\Java\JavaExamples>javac ArgumentDemo.java  
E:\Java\JavaExamples>java ArgumentDemo Hi Michael Clarke  
The following arguments are passed during runtime  
Hi  
Michael  
Clarke  
E:\Java\JavaExamples>
```





# Type Casting

- Type Casting causes the program to treat a variable of one type as though it contains data of another type.

Example:

```
float c = 34.89675f;
```

```
int b = (int) c + 10; // Convert c to an integer
```





# Types of Data Conversion

## Automatic Type Conversion

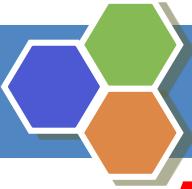
When one type of data is assigned to a variable of another type, automatic type conversion takes place provided it meets the following conditions:

- The two types are compatible.
- The destination type is larger than the source type.

## Casting

Casting is used for explicit type conversion. It loses information above the magnitude of the value being converted.





# Example

## Automatic Type Casting

- Declare and initialize the variables
- Add the variables and assign the value to the variable of different data type without explicit type casting.
- No ERROR!!!

```
long sum;  
  
int first = 20,  
second = 30;  
  
sum = first +  
second;
```

**Demonstration:** Additional  
Example 2

**NIIT**

## Casting

- Declare the variables
- The variables will hold the values which are passed during runtime.
- Add the variable after type casting.

```
int first, second, sum;  
first = Integer.parseInt(args[0]);  
second = Integer.parseInt(args[1]);  
sum = first + second;
```

**Demonstration:** Additional  
Example 1

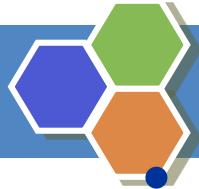




# Type Promotion Rules

- All `byte` and `short` values are promoted to `int` type.
- If one operand is `long`, the whole expression is promoted to `long`.
- If one operand is `float`, the whole expression is promoted to `float`.
- If one operand is `double`, the whole expression is promoted to `double`.





# Arrays

- An array is a data structure that stores data of same data type in consecutive memory locations.
- Three ways to declare an array are:
  - **datatype identifier [ ];**
  - **datatype identifier [ ] = new datatype[size];**
  - **datatype identifier [ ] = {value1,value2,...valueN};**
  - **ClassName identifier[] = new ClassName[size];**
- Array can have more than one dimension
- In Java, Arrays are collection of references





# Arrays

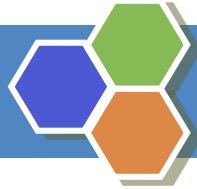
## One Dimensional Array

- It consists of a single column of data of the same type.
- An array is declared by specifying its name and size.

## Multi Dimensional Array

- Multi Dimensional arrays are arrays of arrays.
- To declare a Multi Dimensional array, additional index has to be specified by using another set of square brackets.





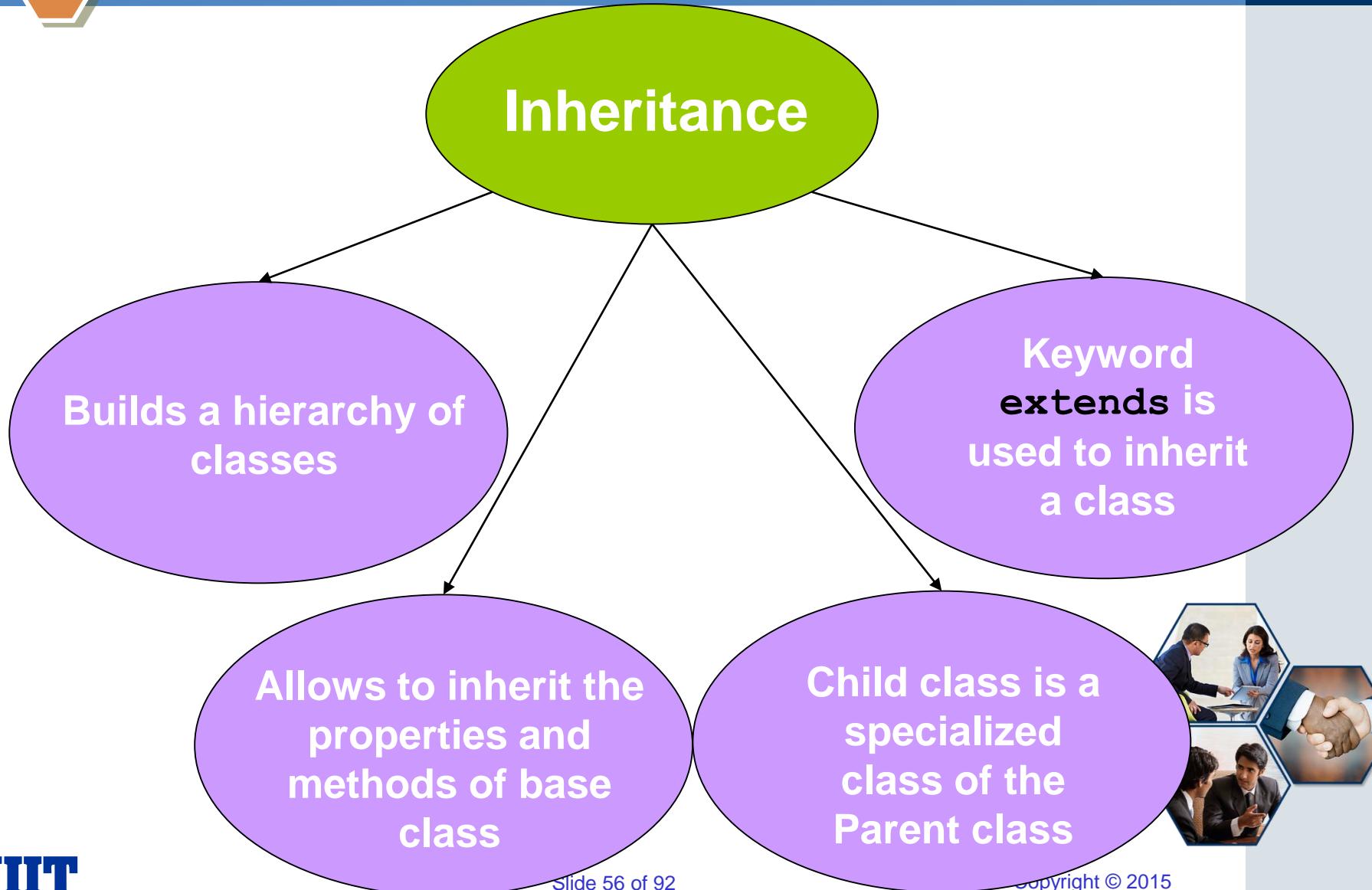
# Example

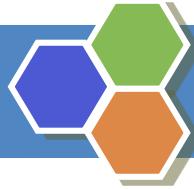
- ```
/* Array Initialization */  
double[] nums = {2, 0, 1};
```
- ```
int[] numbers = {8, 18, 5, 2, 1, 10};
```
- ```
System.out.println("Total number of elements  
in an array is: " + numbers.length);
```
- ```
System.out.println("First element in an  
array is: " + numbers[0]);
```
- ```
System.out.println("Last element in an array  
is: " + numbers[numbers.length - 1]);
```





# Inheritance





# Advantages of Inheritance

## Advantages

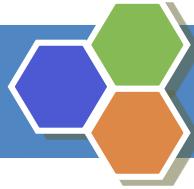
**Reusability of code**

**Data and methods of superclass are available to subclass**

**Subclass can be customized easily**

**Designing applications becomes simpler**





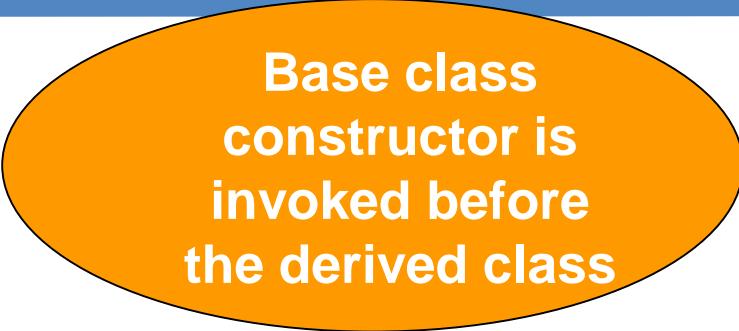
# Inheritance

- ❑ Keyword **extends** is used.
- ❑ Accessing members of the superclass in the subclass.

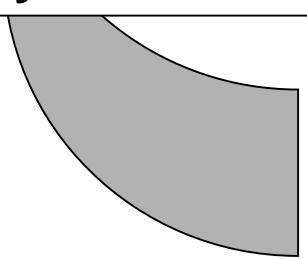
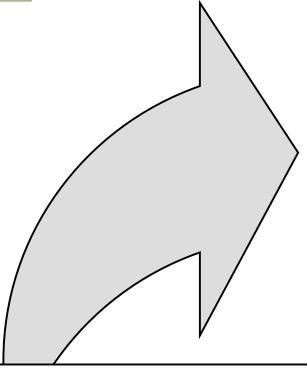




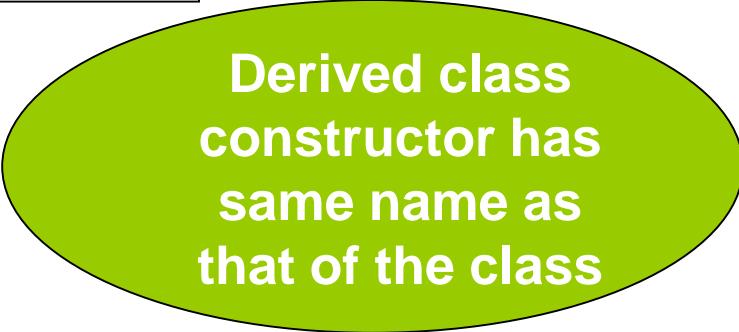
# Derived Class Constructors



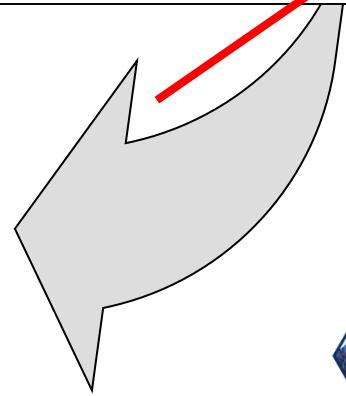
Base class constructor is invoked before the derived class



To invoke base class constructor **super** keyword is used

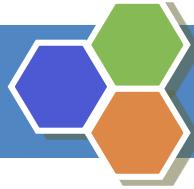


Derived class constructor has same name as that of the class



**Constructor Inheritance**





# Derived Class Constructors 2-2

- The syntax to invoke the superclass constructor is:

```
super(parameter_list) or super();
```

- The parameters required by the constructor of the superclass are specified in the

**parameter\_list**

- The `super()` method always refers to the superclass immediately above the calling class.

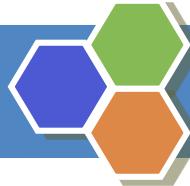




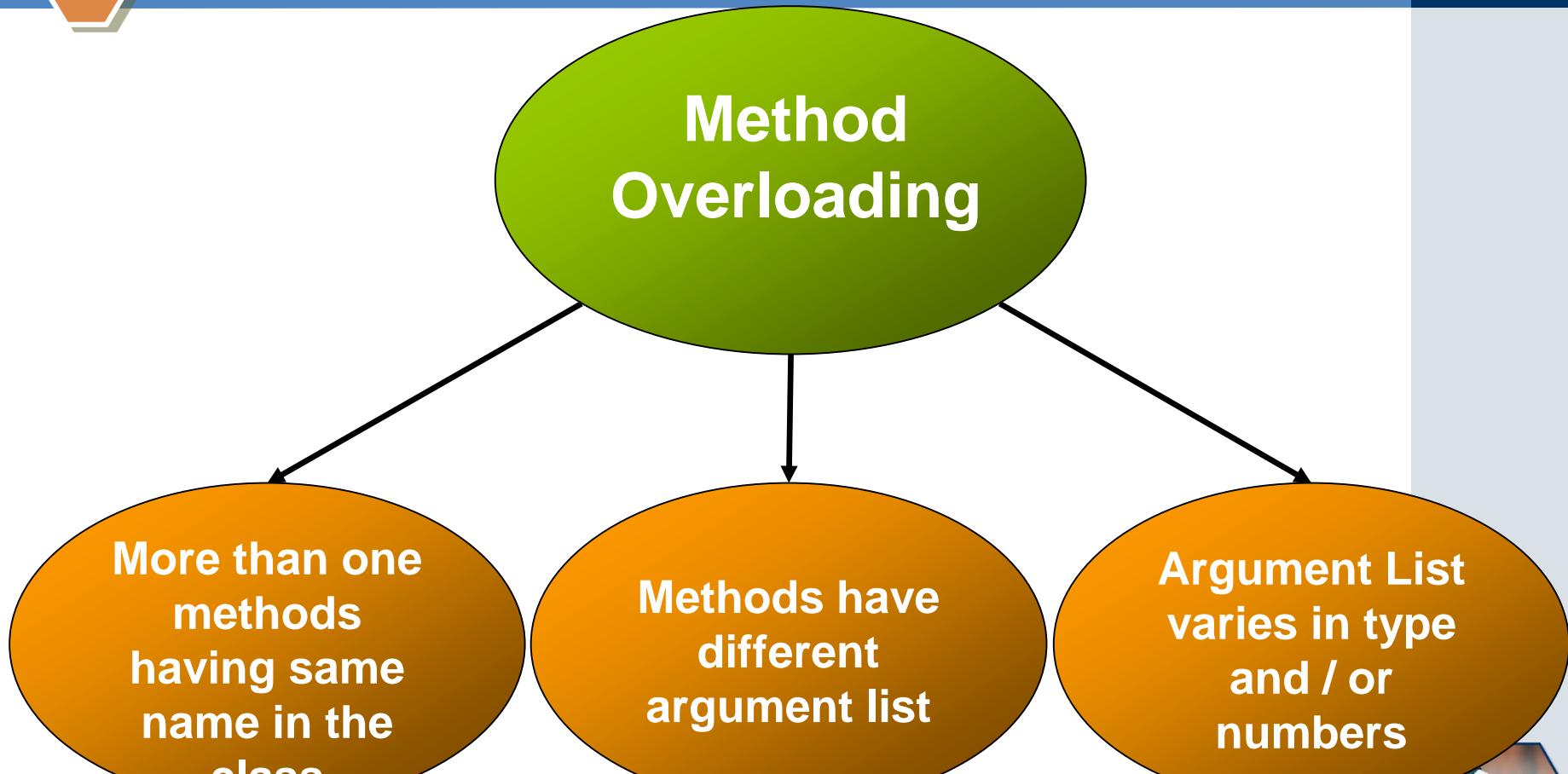
# Polymorphism

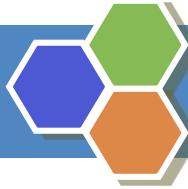
- Polymorphism means “**many forms**”.
- It is described as “one interface, many implementations”.
- It helps to write code that does not depend on specific types.
- It is the capability of a method to do different things based on the object that is acting upon it.





# Method Overloading





# Method Overriding

## Method Overriding

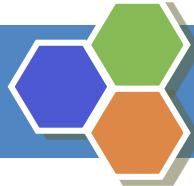
Subclass and Superclass have methods with same name and type signature

`super.methodname()` is used to call the superclass version of the method

Overridden method of the subclass will be invoked when called by a subclass object

Method in the superclass will be hidden





# Dynamic Binding

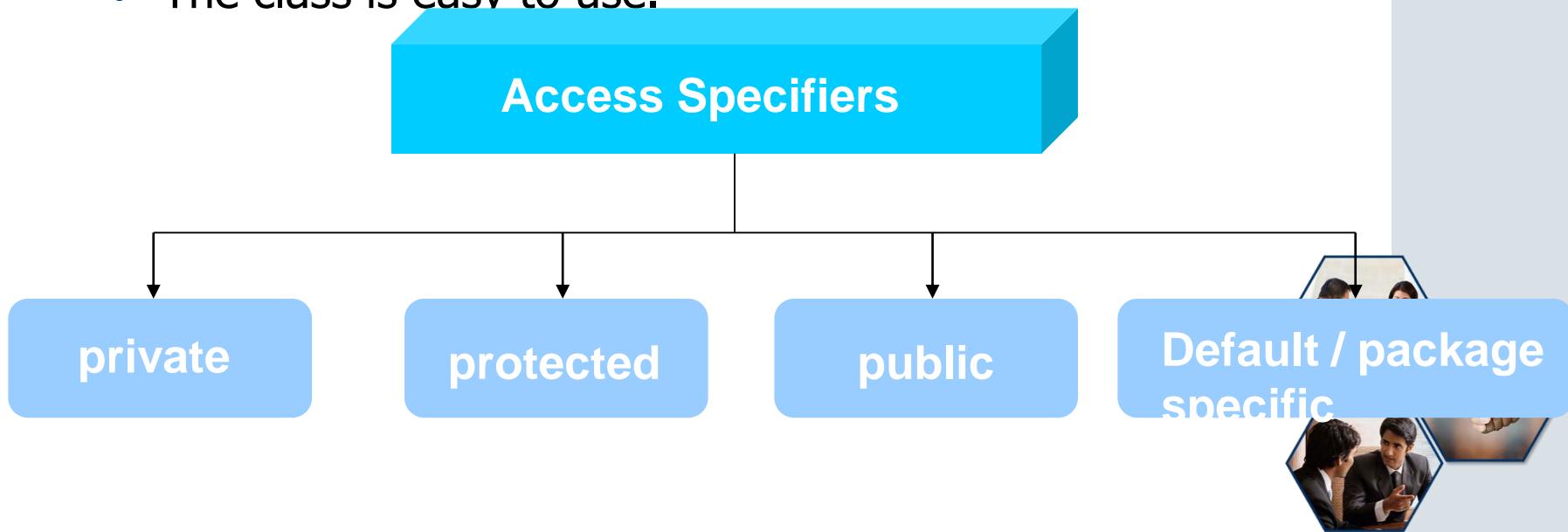
- Super class can refer to sub class.
- But super class reference can't refer to sub class members.
- Sub class can not refers to super class
- Super class will refer to sub class method if it refers to sub class and method is overridden.





# Access Specifier

- Information hiding is one of the most important features of OOPs.
- Reasons for information hiding are:
  - Changes made to any implementation details will not affect code that uses this class.
  - Prevents accidental erasure of data by users.
  - The class is easy to use.





# Access Specifier

**public**

**Accessible to members and non members of the class**

**private**

**Accessible only to members of the class**

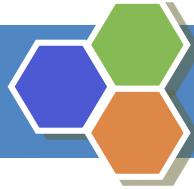
**protected**

**Accessible to members of the class and members of its subclass**

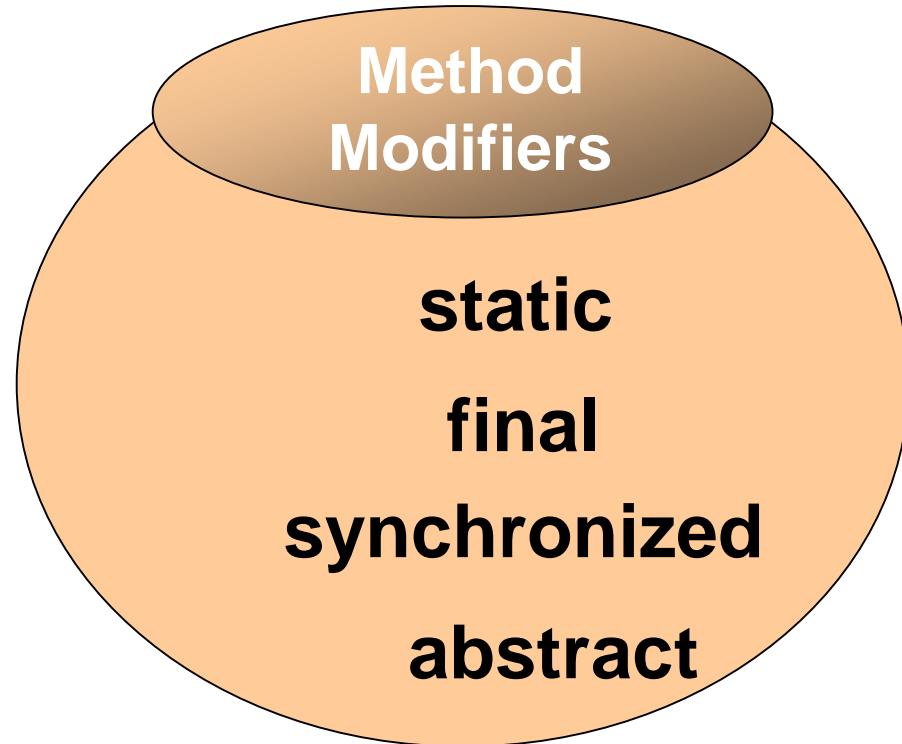
**default**

**Accessible to members of the class in the same package**





# Method Modifiers





# static Modifier

- Programmer might need to access a class member independent of any object of the class.
- The keyword `static` is used to create such a member.
- Such a member is accessed before any object of the class is created.
  - For example, `main()` method is declared static as it can be invoked by the Java runtime system without creating an instance of the class.





# static modifier

## Rules

Can call other static methods

Must access static data

Cannot use **super** or **this** keyword

- Syntax for invoking a static method is:
  - `Classname.methodname( ) ;`





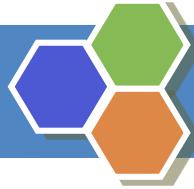
# static modifier

static methods are invoked using class names.

```
public static void main(String [] arg) {  
    //initializing a variable  
    double inch = 66;  
    double feet = InchesToFeet.convert(inches);  
    System.out.println(inch + " inch is " + feet +  
" feet.");  
}
```

```
class InchesToFeet {  
    /** Initializes a static variable. */  
    private static final int inches = 12;  
    /** Constructor. */  
    protected InchesToFeet() {  
    }  
    public static double convert(double in) {  
        return (in / inches);  
    }  
    .....  
}
```





# final Modifier

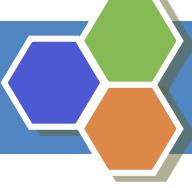
**final**

Variable's content  
cannot be modified

Methods cannot be  
overridden by  
subclass

Reference of  
object should not  
change





# abstract Modifier 3-1

- Certain methods in the superclass do not contain any logic and need to be overridden in the subclass.
- The keyword `abstract` is used for such methods.
- The subclass provides implementation details of such abstract methods.
- The syntax is:
  - `abstract type method_name(parameter_list);`
- Any class that contains one or more abstract methods must be declared as abstract.
- The `class` keyword should be preceded by the `abstract` keyword when declaring an abstract class.





# abstract Modifier 3-2

## abstract

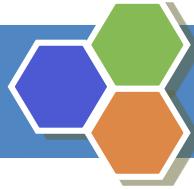
Abstract class  
cannot be  
instantiated

Constructors and  
static methods  
cannot be abstract

Abstract method  
of superclass has  
to be  
implemented in  
subclass

- An abstract class can have object references that can point to a subclass object.

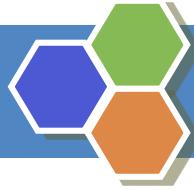




# Interface

- Interface is an contract with class that class has to implement functionality
- Interface tells only about functionality
- Interface doesn't tell about implementation
- By default all methods are “public abstract”
- By default all members are “public final”
- To implement an interface, “implements” keyword is being used.
- Interface can extends one or more interfaces

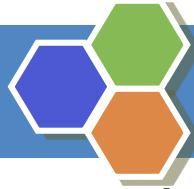




# Interface

```
interface MyInterface {  
    void call(String name);  
    void wish(String name);  
}  
  
class MyClass implements MyInterface {  
    public void call (String name) {  
        System.out.println ("Hi " + name); }  
  
    public wish(String name) {  
        System.out.println ("Have a great day " + name); }  
}
```





# Summary

- Inheritance allows the creation of hierarchical classifications.
- Inheritance allows the reusability of code.
- All methods and properties of the base class are inherited by objects of the derived class except the constructors.
- Polymorphism is the ability of different objects to respond to the same message in different ways.
- Overloaded methods are examples of static polymorphism and overridden methods are examples of dynamic polymorphism.





# Summary

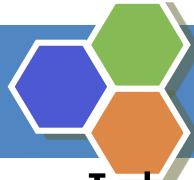
- Access specifiers are used to determine how the class members are accessed.
- Some of the method modifiers in Java are static, final, abstract.
- Interface is a contract for class to provide certain functionality without telling about its implementation.





# Exceptions





# Review

- Inheritance allows the creation of hierarchical classifications.
- Inheritance allows the reusability of code.
- All methods and properties of the base class are inherited by objects of the derived class except the constructors.
- Polymorphism is the ability of different objects to respond to the same message in different ways.
- Overloaded methods are examples of static polymorphism and overridden methods are examples of dynamic polymorphism.
- Access specifiers are used to determine how the class members are accessed.
- Some of the method modifiers in Java are `static`, `final`,





# Objectives

## *Define Exception*

- *Explain exception handling*
- *Describe the try, catch and finally blocks*
- *Examine multiple catch blocks*
- *Explore nested try/catch blocks*
- *Explain the use of throw and throws keywords*



# What is an exception?

```
class ExceptionRaised {  
    /** Constructor. */  
    protected ExceptionRaised() {}  
    /**  
     * This method generates an exception.  
     * @param operand1 is numerator in division  
     * @param operand2 is denominator in division  
     * @return It will return the remainder of the division.  
     */  
    static int remainder(int operand1, final int operand2) {  
        int result = operand1 / operand2; // user defined method  
        return result;  
    }  
    /**  
     * Sole entry point to the class and application.  
     * @param args Array of String arguments.  
     */  
    public static void main(final String[] args) {  
        System.out.println("Division:  
        ");  
        int[] g = new int[2];  
        g[0] = 10;  
        g[1] = 0;  
        System.out.println("The variable result is defined to store the result.  
        ");  
        System.out.println(result);  
        System.out.println("Exception object occurred : " + e.toString());  
        e.printStackTrace();  
    }  
}
```

- Can be generated manually in a program

OR

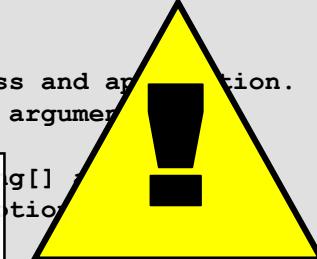
Generated by Java Runtime

Abnormal Condition

Exception

## Error Handling Benefits

- Fixes Error
- Prevents automatic termination



Program Terminates Abruptly and control is given to OS

OS





# Handling Exceptions

## A pseudo code handling a runtime error

```
.....  
IF B IS ZERO GO TO ERROR  
C = A / B  
PRINT C  
GO TO EXIT
```

**ERROR:**

BLOCK THAT  
HANDLES THE  
EXCEPTION

“CODE CAUSING ERROR DUE TO DIVISION BY ZERO”

**DISPLAY**

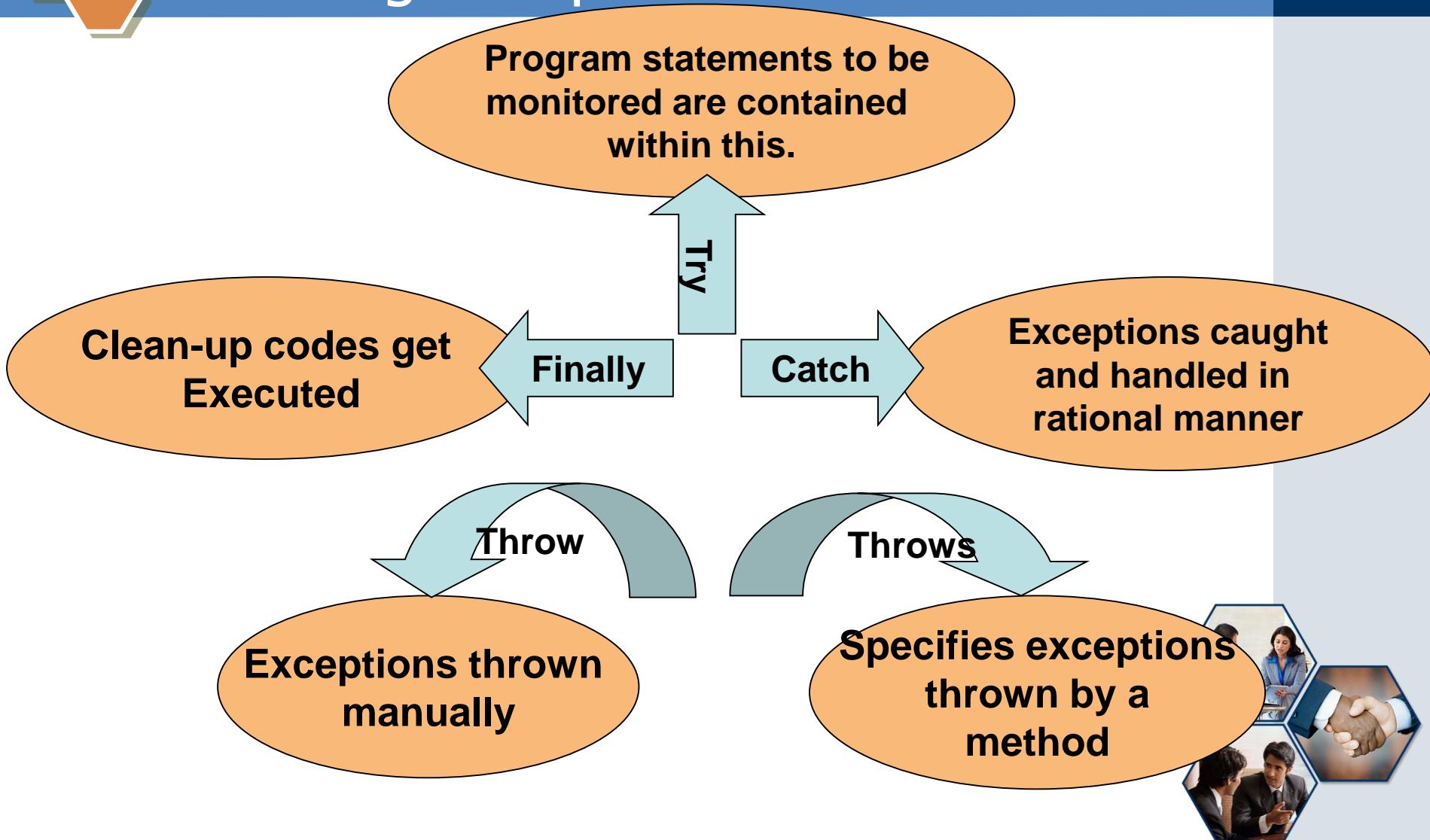
**EXIT:**

**END**

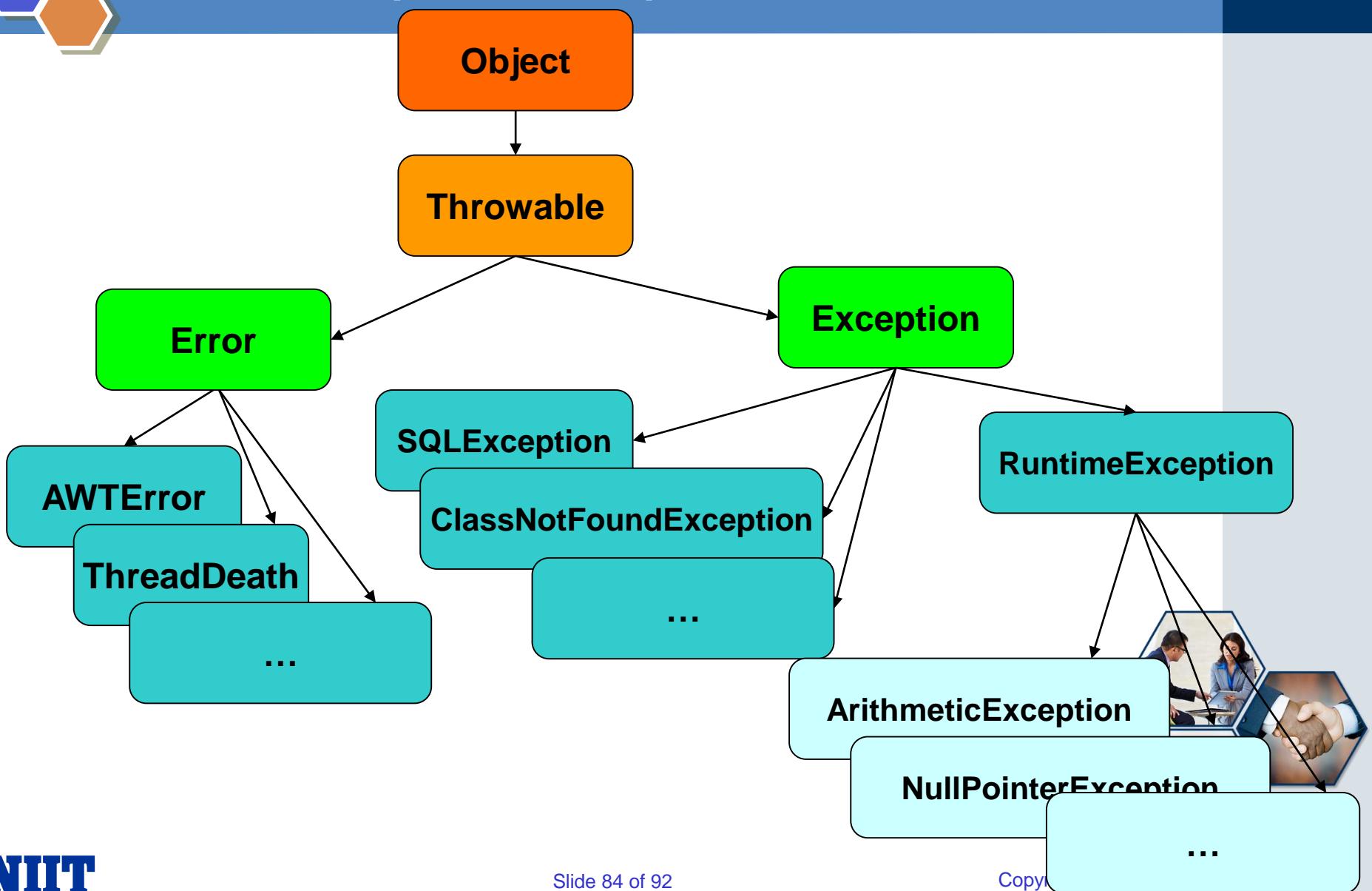




# Handling Exceptions



# Hierarchy of Exception classes

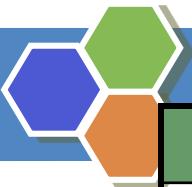




# Hierarchy of Exception classes

- ❑ All exception types are subclasses of the built-in class **Throwable**.
- ❑ **Throwable** has two subclasses, they are:
  - ❑ **Exception**: To handle exceptional conditions that user programs should catch.
    - ❑ An important subclass of **Exception** is **RuntimeException**, which includes division by zero and invalid array indexing.
  - ❑ **Error**: To handle exceptional conditions that are not expected to be caught under normal circumstances. i.e. stack overflow





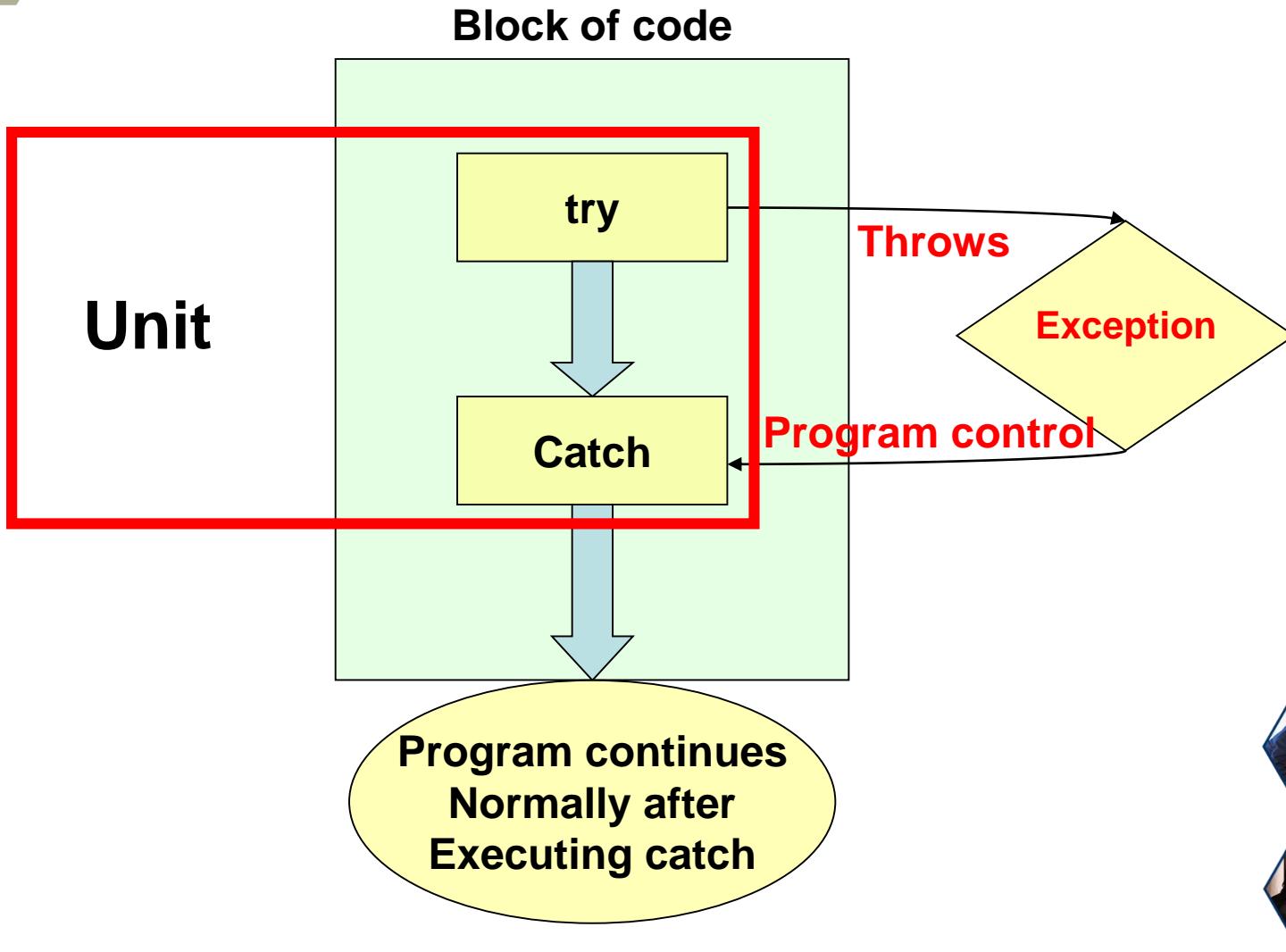
# Hierarchy of Exception classes

Exception	Description
<b>Exception</b>	<b>Root class of exception hierarchy</b>
<b>RuntimeException</b>	<b>Base class for many java.lang exceptions</b>
<b>ArithmaticException</b>	<b>Arithmatic error condition, such as divide by zero</b>
<b>IllegalArgumentException</b>	<b>Method received illegal argument</b>
<b>ArrayIndexOutOfBoundsException</b>	<b>Array size is less or greater than actual array size</b>
<b>NullPointerException</b>	<b>Attempt to access <i>null</i> object member</b>
<b>SecurityException</b>	<b>Security settings do not allow operation</b>
<b>ClassNotFoundException</b>	<b>Unable to load requested class</b>
<b>NumberFormatException</b>	<b>Invalid conversion of a string to a numeric float</b>
<b>IOException</b>	<b>Root class for I/O exceptions</b>
<b>FileNotFoundException</b>	<b>Unable to locate a file</b>
<b>EOFException</b>	<b>End of file</b>
<b>IllegalAccessException</b>	<b>Access to a class denied</b>
<b>NoSuchMethodException</b>	<b>Requested method does not exist</b>
<b>InterruptedException</b>	<b>Thread interrupted</b>





# try and catch blocks 2-1





# try and catch blocks 2-2

```
class ExceptionRaised {  
    /** Constructor. */  
    protected ExceptionRaised() {  
    }  
    /**  
     * This method generates an exception.  
     * @param operand1 is numerator in division  
     * @param operand2 is denominator in division  
     * @return It will return the remainder of the  
division.  
     */  
    static int calculate(final int operand1, final int  
operand2) {  
  
        int result = operand1 / operand2;          // user  
defined method  
        return result;  
    }  
}
```





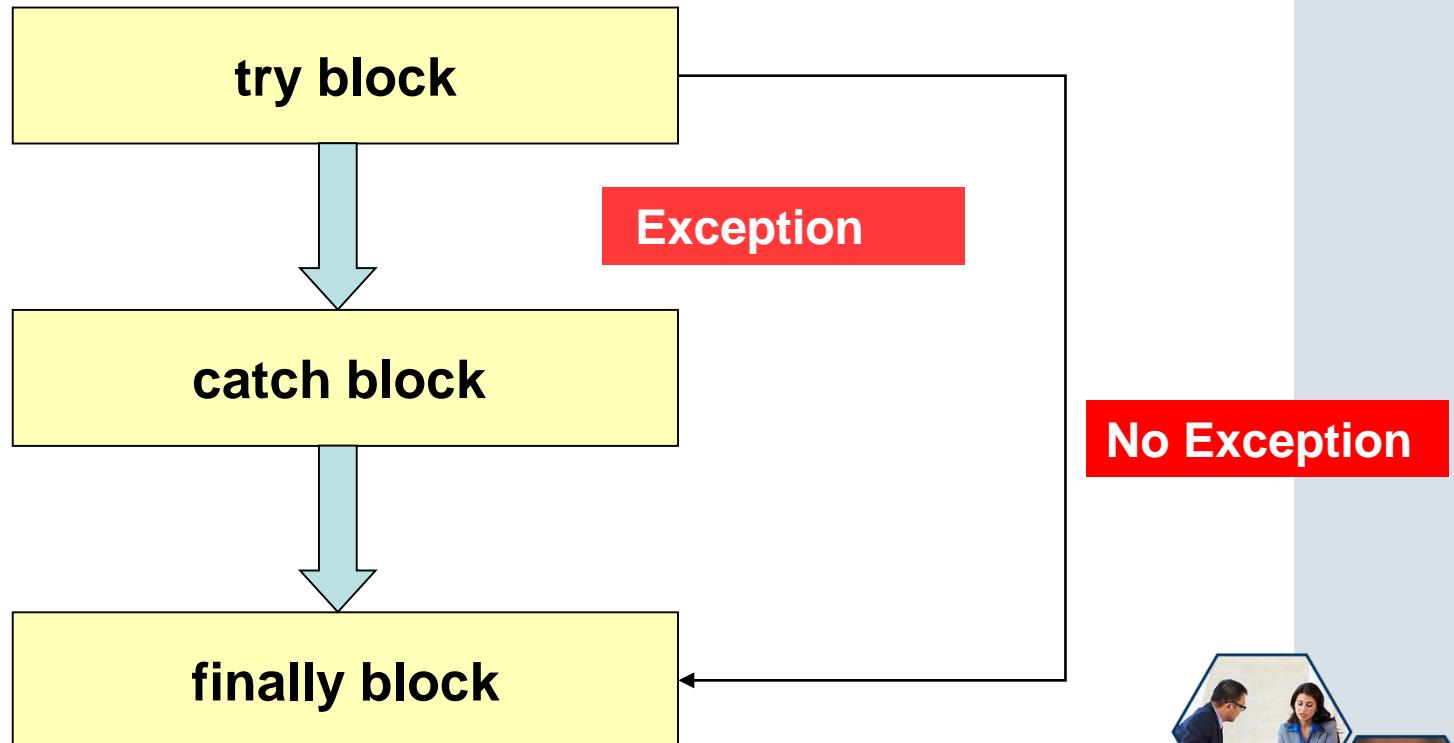
```
public class ArithmeticException {  
    /** Constructor. */  
    protected ArithmeticException() {  
    }  
    public static void main(final String[] args) {  
        ExceptionRaised obj = new ExceptionRaised();  
        try {  
            /* The variable result is define to store the  
result. */  
            int result = obj.calculate(9, 0);  
            System.out.println(result);  
        } catch (Exception e) { // Exception object  
            System.err.println("Exception occurred : " +  
e.toString());  
            e.printStackTrace();  
        }  
    }  
}
```





# finally Block

## Execution flow of try, catch and finally blocks





# General form of exception-handling block

```
try{
    // block of code to monitor for errors
    methodGeneratingException();

}

catch (Exception e) {
    // exception handler for Exception e
}

finally{
    // block of code to be executed before
try ends
    cleanup();
}
```





# Multiple catch blocks

- Single piece of code can generate more than one error.
- 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.

```
....  
try{  
    }  
    catch (ArrayIndexOutOfBoundsException e) {  
    }  
    catch (Exception e) {  
    }  
.... .
```



# Nested try - catch blocks

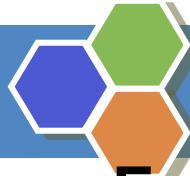
```
* All Rights Reserved This class demonstrate the nested try-catch statements.  
* class NestedException { /* Constructor. */  
protected NestedException() {  
} /** This method test the format of the number  
* @param argument is used to store the value of args.  
*/  
public test(String argument) {  
try {  
int num = args.length;  
/* Nested try block. */  
try {  
int numValue = Integer.parseInt(args[0]);  
System.out.println("The square of " + args[0] +  
+ numValue * numValue);  
} catch (NumberFormatException nb) {  
/** Displaying the appropriate message, if exception  
* has occurred.  
*/  
System.out.println("Not a number! ");  
}  
} catch (ArrayIndexOutOfBoundsException ne) {  
System.out.println("Please enter the number!!!!");  
}  
}  
public static void main(final String[] args) {  
NestedException obj = new NestedException();  
obj.test(args[0]);  
} }
```

Inner try  
executed first

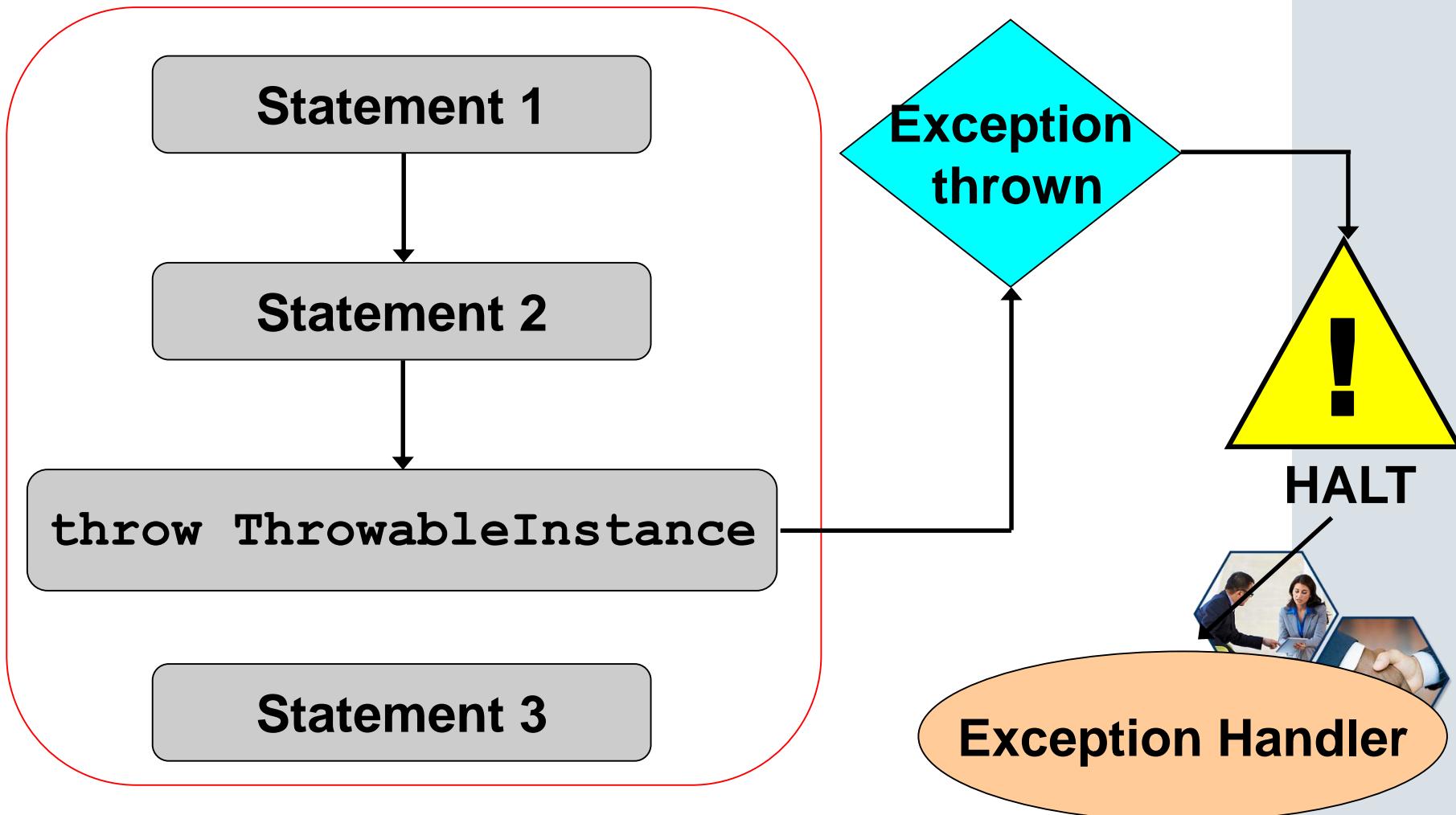
This is why  
exception  
handlers need  
to be nested

Outer catch inspected in  
case inner try does not  
have a catch





# Using throw & throws Executable Program Statements





# Using throw & throws

## Called Method

Can cause exceptions

```
type calledmethod-name  
(parameter-list)  
throws exception-list  
{  
// body of method  
}
```

~~Handle exceptions~~

## Calling Method

Guards against called  
method exceptions  
and handles them

```
type callingmethod-name {  
try {  
// statements  
Calledmethod-name();}  
catch(Exception e) {  
//statements}  
}
```

Handles exceptions

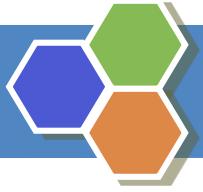




# Summary

- Whenever an error is encountered during run time, an Exception occurs.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error.
- Java exception handling is managed using try, catch, throw, throws, and finally.
- Program statements to monitor are contained within a try block. Code within catch block catches the exception and handles it.
- Any code that absolutely must be executed before a method returns is put in a finally block.
- To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified by a throws clause.





# Collection Framework

java.util Package

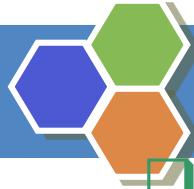




# Objectives

- *Set Interface*
  - *HashSet*
  - *TreeSet*
  - *LinkedHashSet*
  - *Special Purpose implementation*
    - *EnumSet and CopyOnWriteArraySet*
  - *List Interface*
    - *ArrayList*
    - *LinkedList*
  - *Special Purpose implementation*
    - *CopyOnWriteArrayList*
  - *Map*
    - *HashMap*
    - *TreeMap*
    - *LinkedHashMap*

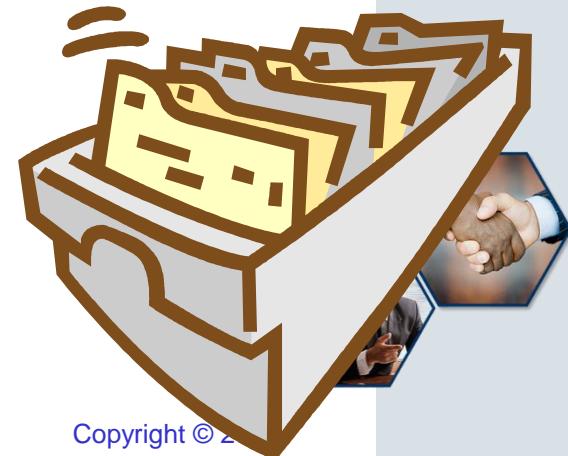
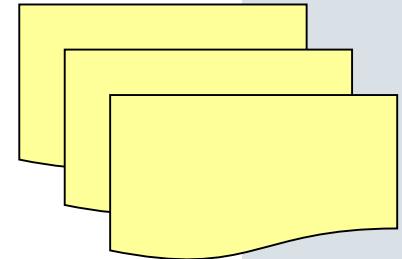


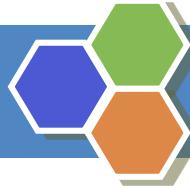


# Collection Classes

 An object of the Collection class groups multiple elements/objects into a single unit.

- Collections are used to store, retrieve and manipulate data and to transmit data from one method to another.
- Collection Framework is an unified architecture for representing and manipulating collections.





# Collection Classes

**Collection Framework is composed of three components.**

## Interface

Is abstract  
data types  
representing  
collections

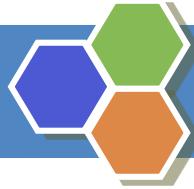
## Implementation

Is actual  
execution of  
interfaces

## Algorithms

Are methods that  
perform  
computations on  
objects that  
implement the  
interface

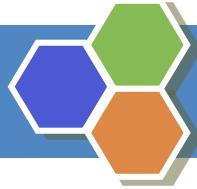




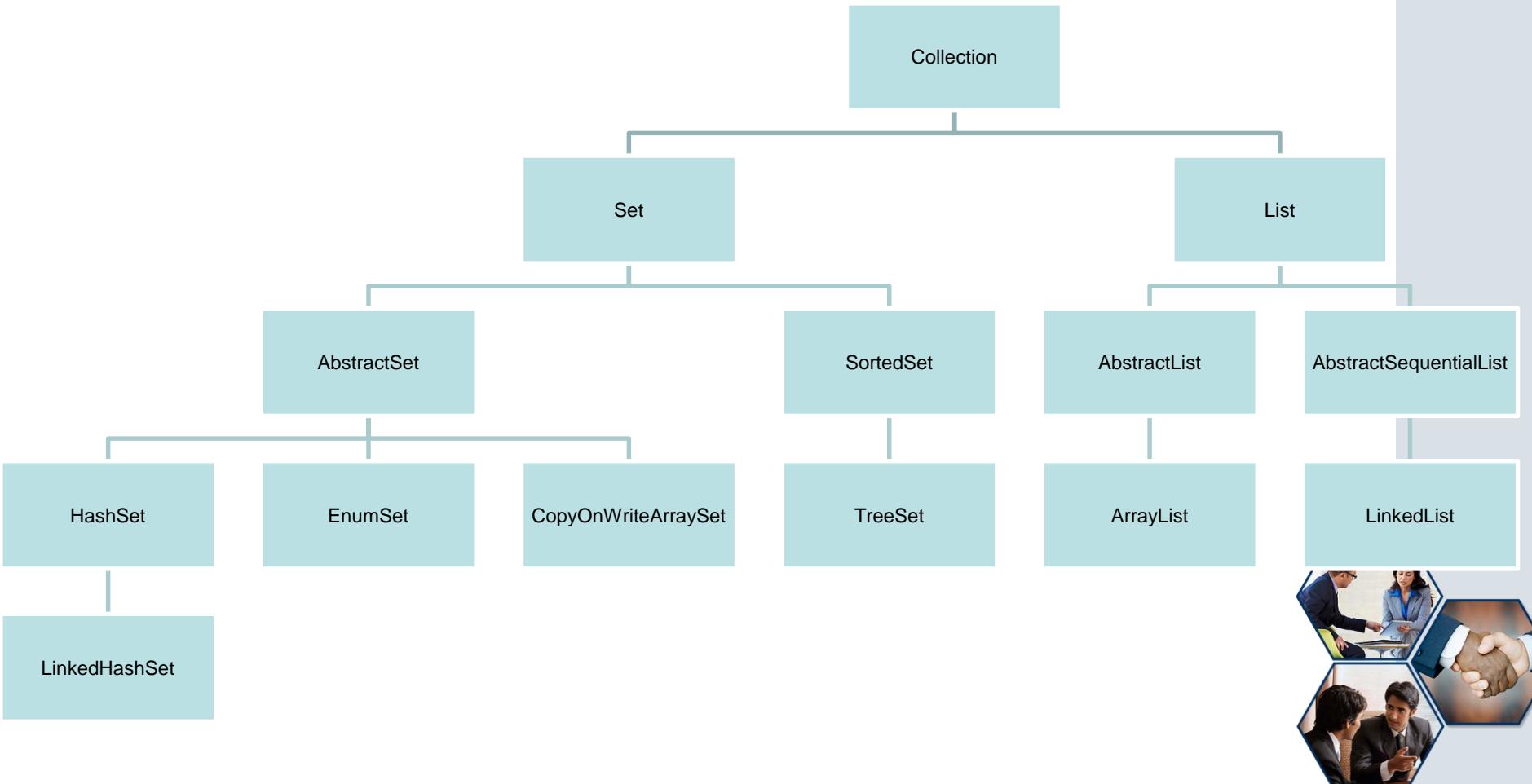
# Advantages of Collection Framework

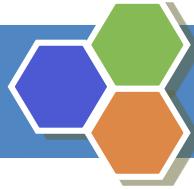
- Reduces programming effort by providing useful data structures and algorithms.
- Increases program speed and quality since the implementation of each interface is interchangeable.
- Allows interoperability among different APIs.
- Extending or adapting a collection is easy.





# Collection Framework





# HashSet

- Use Hash table for storage
- Store unique objects
- Can store various objects
- Doesn't arrange/sort objects
- Information stored using *Hashing*
- Same execution time irrespective of size of collection
- Methods
  - add()
  - contains()
  - remove()
  - size()

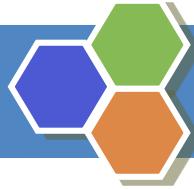




# LinkedHashSet

- Extends HashSet
- Same as HashSet
- Data retrieval is in the manner as it inserted
- Introduce from Java 1.4

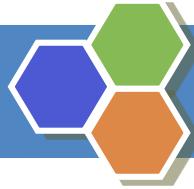




# TreeSet

- Store unique objects
- Information stored in Tree
- Store similar types of objects
- Arrange/sort objects
- Methods
  - add()
  - contains()
  - remove()
  - size()
  - first()
  - last()





# EnumSet

- Introduced in Java 1.5
- Specialized Set used with enum
- All basic operations execute in constant time





# CopyOnWriteArrayList

- Available since Java 1.5
- Work with snap shot of CopyOnWriteArrayList
- Thread safe





# ArrayList

- An `ArrayList` object is a variable length array of object references.
- It is used to create dynamic arrays.
- Extends `AbstractList` and implements `List` interface.
- ArrayLists are created with an initial size.
- As elements are added, size increases and the array expands.
- It gives better performance when accessing and iterating through objects.





# LinkedList

- LinkedList class is used to create a linked-list data structure.
- Extends AbstractSequentialList and implements the List interface.
- Constructors of LinkedList class are:

Constructor	Description
LinkedList()	Creates an empty linked list
LinkedList(Collection c)	Creates a linked list based on the elements of a given collection.

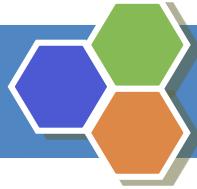




# CopyOnWriteArrayList

- Available since Java 1.5
- Same as ArrayList
- Work with snap shot of array

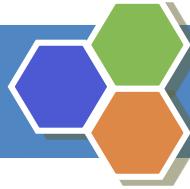




# Iterator Interface

- Used to navigate in Collection
- Methods
  - `hasNext()`
  - `next()`
  - `remove()`





# Map Interfaces and Classes

## Map Interfaces and Classes

Map is an object and stores data in the form of a relationship between keys and values

Keys should be unique but values can be duplicated

Some maps can store null object for keys and values





# Map Interfaces and Classes

- It maps keys to values.
- Each key can map to one value only.
- It contains methods for basic operations, bulk operations and collection views.
- `SortedMap` interface extends `Map` interface.
- `SortedMap` interface maintains its entries in ascending order.





# Map Interfaces and Classes 4-3

The classes that implement the map interfaces are:

**AbstractMap**

**Implements  
most of the  
map  
interfaces.**

**TreeMap**

**Guarantees  
that its  
elements will  
be sorted in  
ascending key  
order**

**HashMap**

**Used for  
Hashtables**

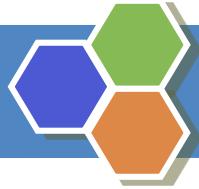




# HashMap

- Provides optional map operations.
- Two parameters affect the performance of the instance of the HashMap. They are:
  - initial capacity: determines the capacity of the hash table when it is created
  - load factor: determines how full the hash table can become before its capacity is automatically increased.
- It does not guarantee the order of its elements, that is, the order in which the iterator reads the HashMap is not constant.
- Allows null values





# TreeMap

- Same as HashMap
- It guarantees the order of its elements.
- Allows null values

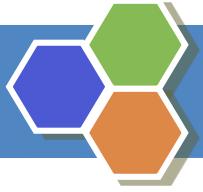




# LinkedHashMap

- Extends HashMap
- Same as HashMap
- Data retrieval is in the manner as it inserted
- Introduce from Java 1.4

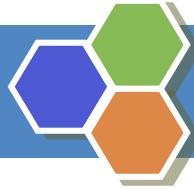




# Java Data Base Connectivity

`java.sql` package

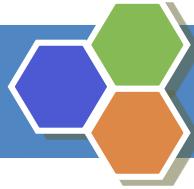




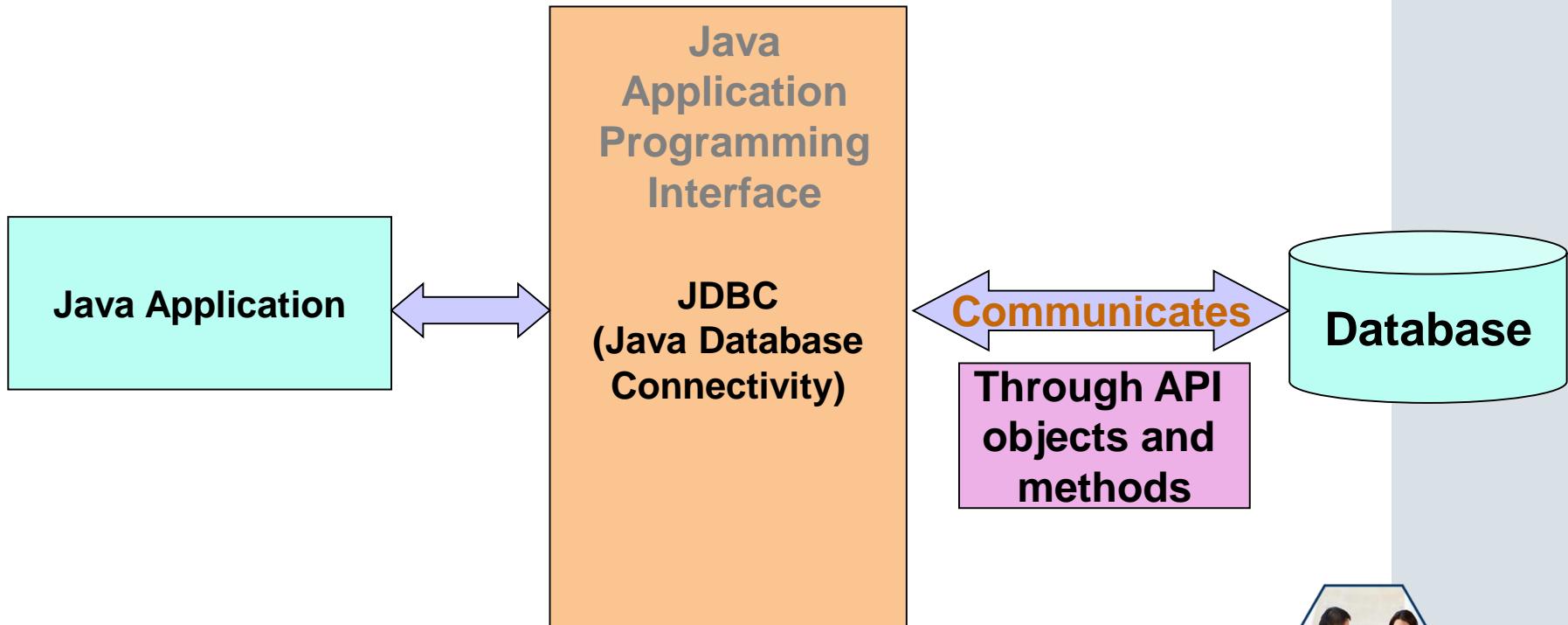
# Objectives

- *Discuss Java Database Connectivity (JDBC)*
- *Describe the java.sql package in brief*
- *Discuss types of JDBC drivers*
- *Explain the anatomy of a JDBC program*
- *Describe the PreparedStatement interface*
- *Use ResultSet Interface*



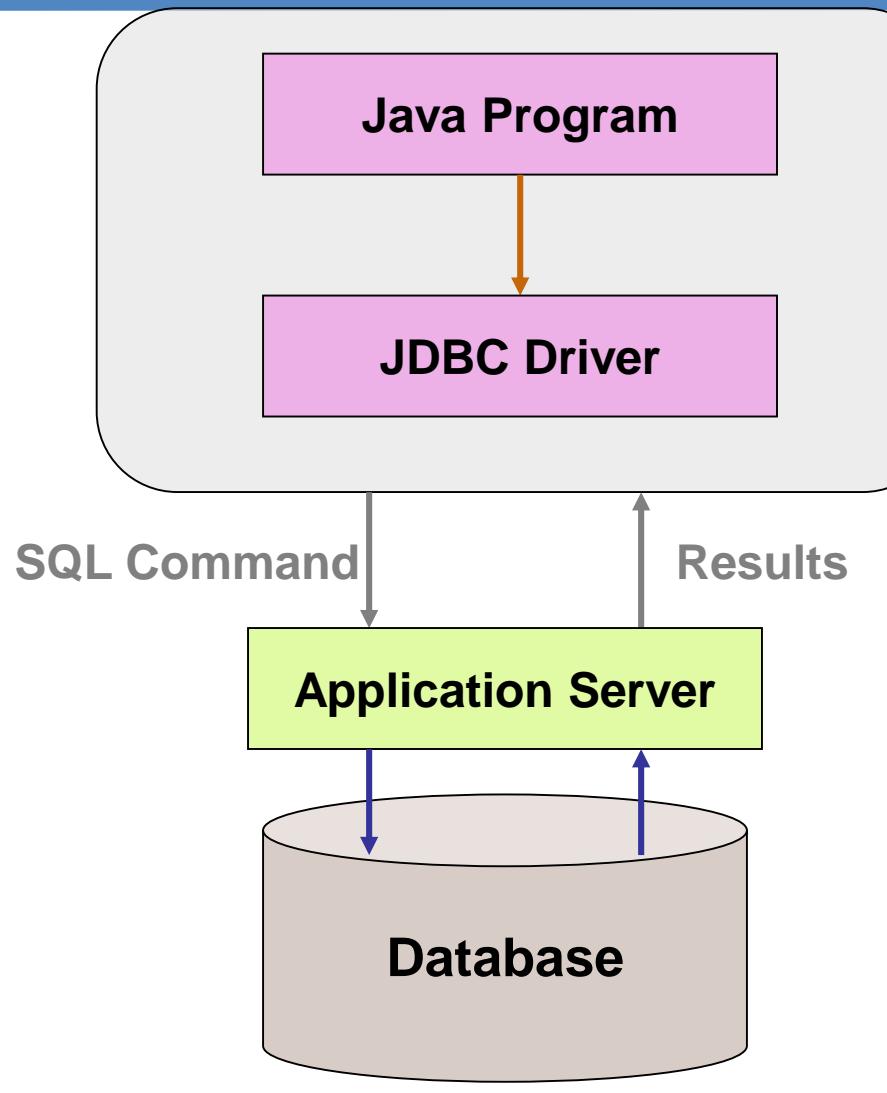


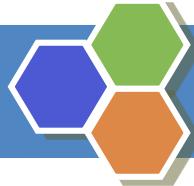
# JDBC



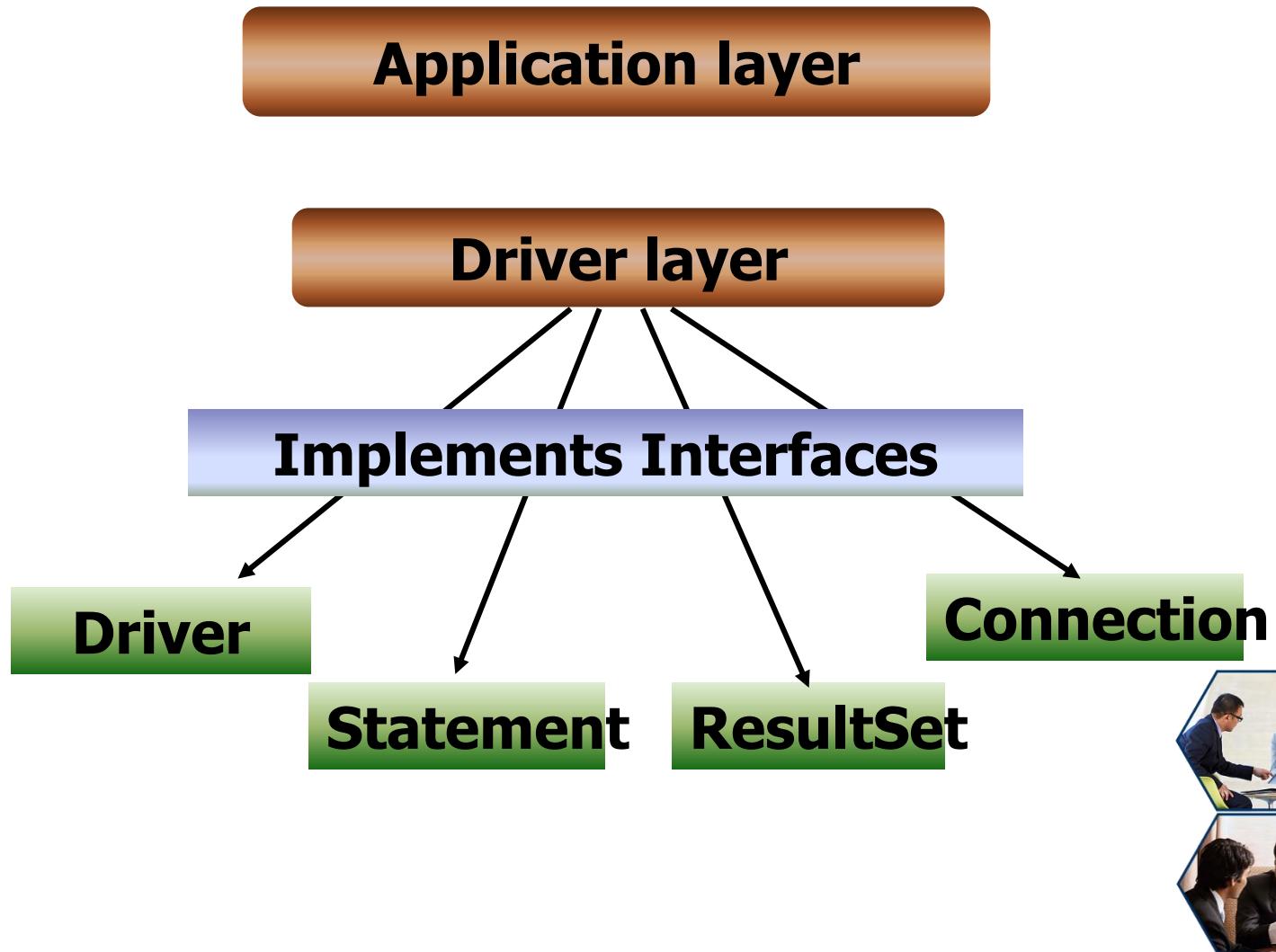


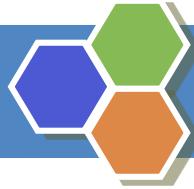
# JDBC Architecture





# JDBC Architecture





# Driver Types

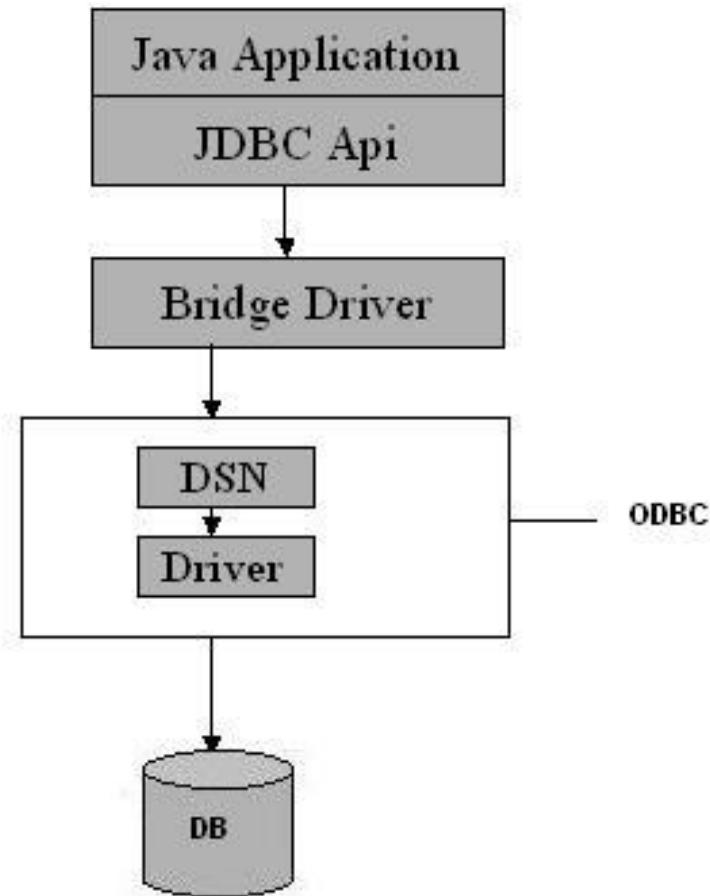
- Type 1 – JDBC ODBC Bridge
- Type 2 – Native API partly Java
- Type 3 – All Java Net Protocol
- Type 4 – All Java Native Protocol





# JDBC ODBC Driver

- driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available





# Type -1 Driver

## Advantage

- The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

## Disadvantage

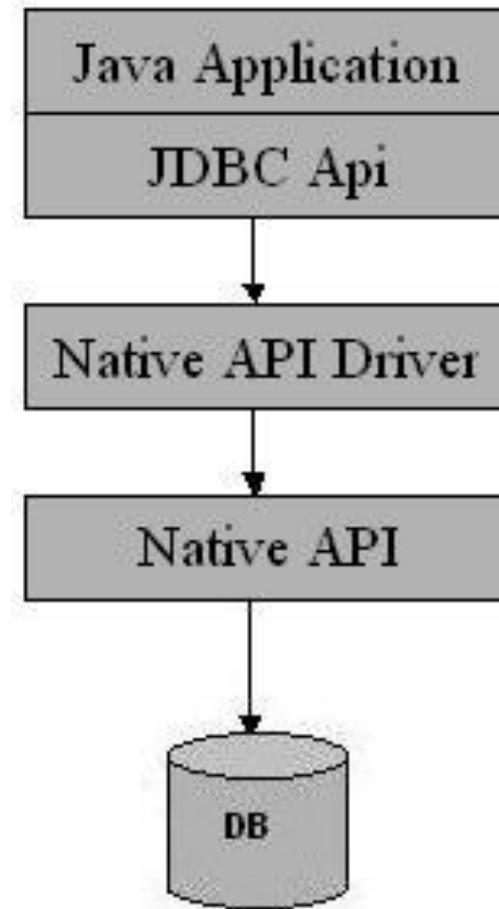
- Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
- A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
- The client system requires the ODBC Installation to use the driver.
- Not good for the Web





# Native API/partly Java

- The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api





# Adv/Disadv

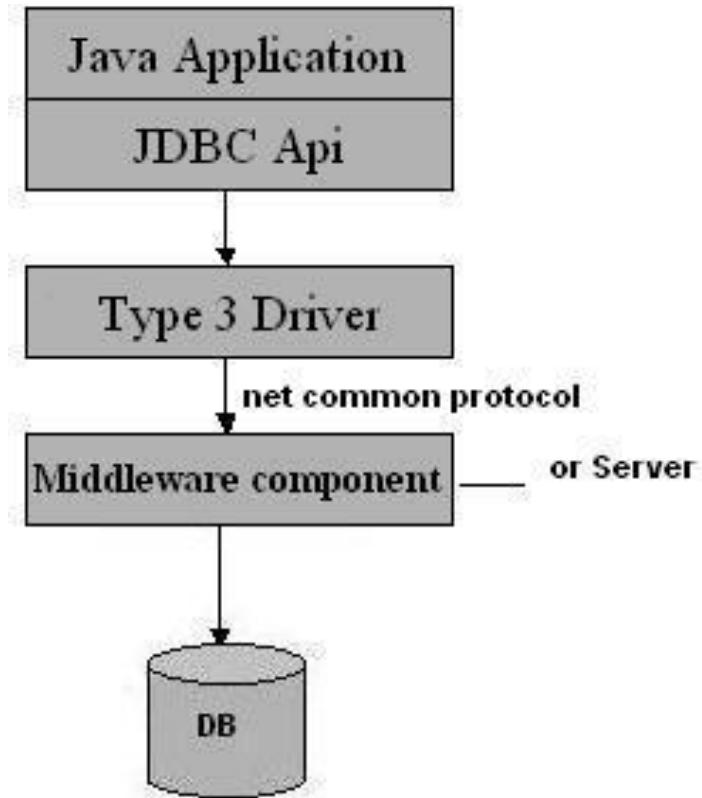
- **Advantage**
- The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.
- **Disadvantage**
- Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
- Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
- If we change the Database we have to change the native api as it is specific to a database
- Mostly obsolete now.
- Usually not thread safe.





# All Java Net Protocol

- Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers





## Advantage

- This driver is server-based, so there is no need for any vendor database library to be present on client machines.
- This driver is fully written in Java and hence Portable. It is suitable for the web.
- There are many opportunities to optimize portability, performance, and scalability.
- The net protocol can be designed to make the client JDBC driver very small and fast to load.
- The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
- This driver is very flexible allows access to multiple databases using one driver.
- They are the most efficient amongst all driver types.

## Disadvantage

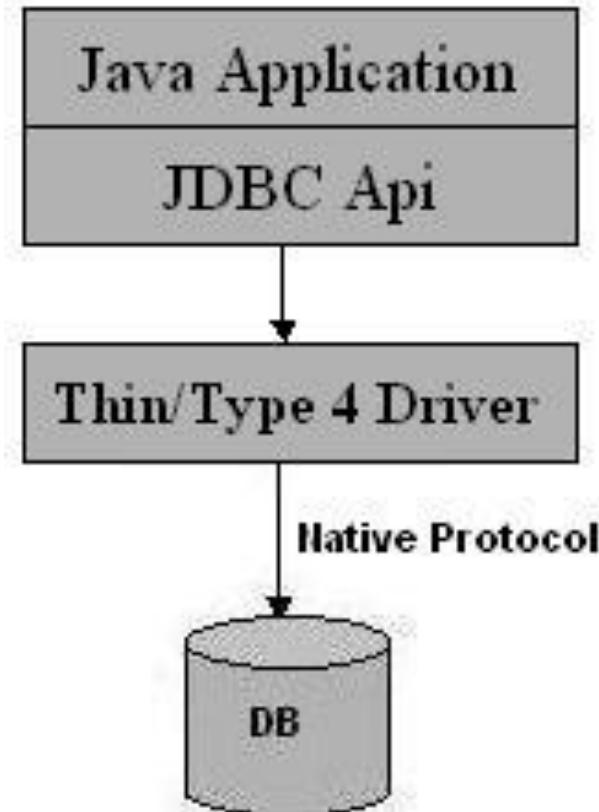
- It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.





## Native Protocol/All Java Protocol

- The Type 4 uses java networking libraries to communicate directly with the database server





# Adv / Disadv

## Advantage

- The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
- Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
- You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

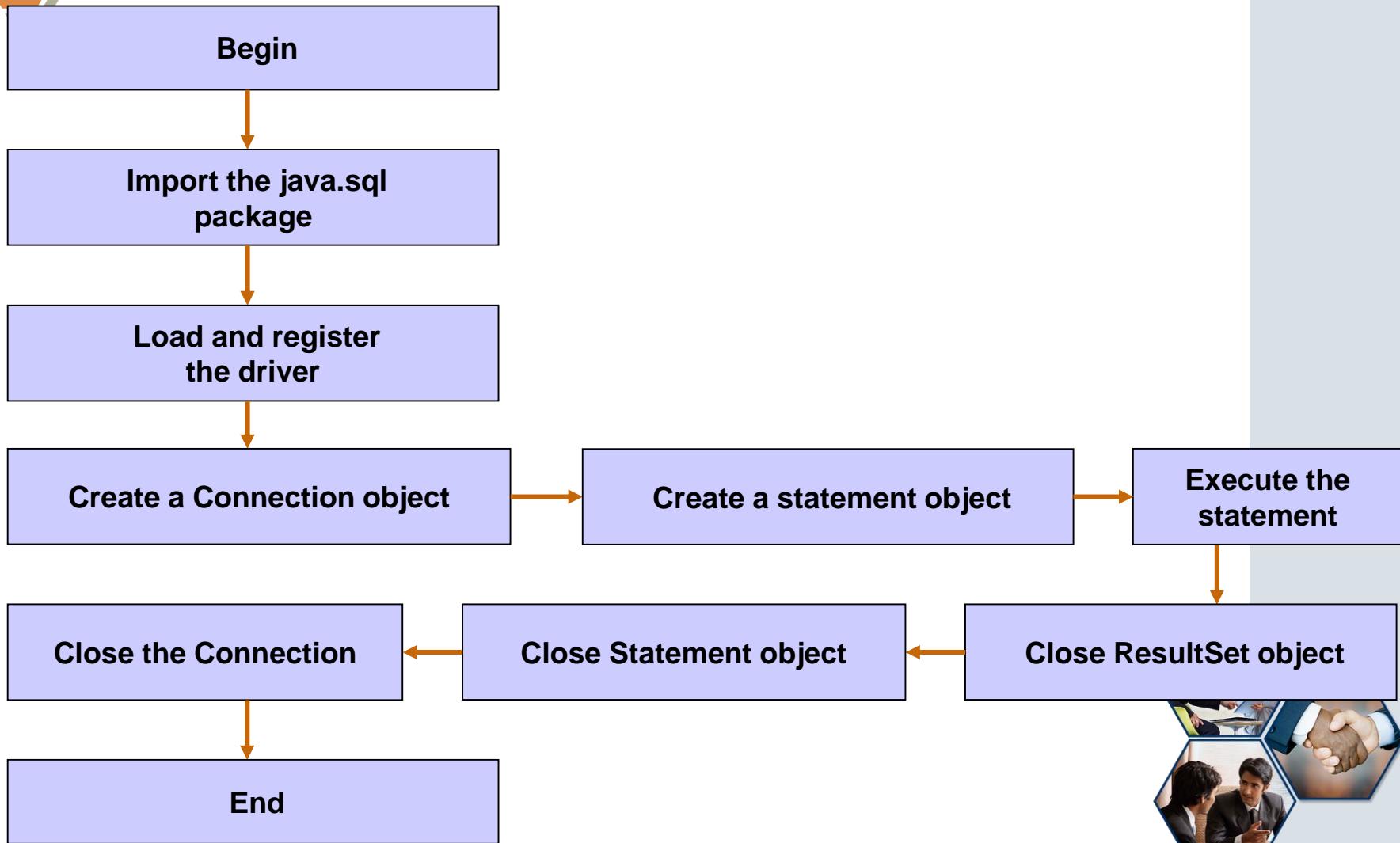
## Disadvantage

- With type 4 drivers, the user needs a different driver for each database





## Steps for Database Access





# JDBC program

- import java.sql.\*;
- class JdbcTest {
- public static void main (String args []) {
- // Load JDBCODBCDriver
- Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- // Connect to the local database
- Connection conn = DriverManager.getConnection  
        ("jdbc:odbc:EMPDSN","sa", "niit011p");





## Contd...

```
// Query the student names  
Statement stmt = conn.createStatement ();  
ResultSet rset = stmt.executeQuery ("SELECT * FROM Bank");  
// Print the name out  
//name is the 2nd attribute of Student  
while (rset.next ())  
    System.out.println (rset.getString (2));  
  
//close the result set, statement, and the connection  
rset.close();  
stmt.close();  
conn.close();  
} }
```

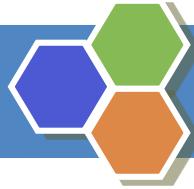




# Statement type

- Statement
- PreparedStatement
- CallableStatement

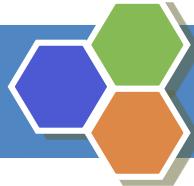




# ResultSet

- ResultSet
- ResultSetMetaData



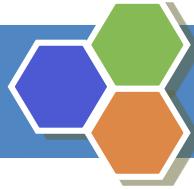


# PreparedStatement with Select

```
PreparedStatement ps = conn.prepareStatement( "Select *\nfrom Bank where city=? And balance>?");\n\nps.setInt(2, 500000);\n\nps.setString(1, "Bangalore");
```

```
ResultSet rs = ps.executeQuery();
```





# DML

- Insert
  - `INSERT INTO table (col1, col2, ....) VALUES (val1, val2, ....)`
- Update
  - `UPDATE table SET col1=val1, col2=val2 WHERE col=val`
- Delete
  - `DELETE FROM table WHERE col=val`





## Update using PreparedStatement

```
PreparedStatement ps = conn.prepareStatement( "UPDATE  
BankDetails SET city= ? WHERE name = ?");
```

```
ps.setString(1, "Bangalore");
```

```
ps.setString(2, "Harish");
```

```
ps.executeUpdate();
```





# Using Transaction

- When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

```
conn.setAutoCommit(false);
```

....

```
transaction
```

...

```
con.commit();
```

```
con.setAutoCommit(true);
```





# Using Transaction

```
con.setAutoCommit(false);  
PreparedStatement updateCity = con.prepareStatement(  
    "UPDATE BankDetails SET City = ? WHERE name  
    LIKE ?");
```

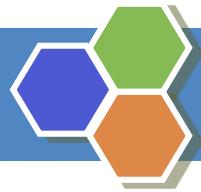
```
updateCity.setString(1, "Bangalore");  
updateCity.setString(2, "Harish");
```

```
int res = updateCity.executeUpdate();  
If(res == 0)  
    con.rollback();  
else  
    con.commit();  
con.setAutoCommit(true);
```



# java.sql package

Interface Name	Description
CallableStatement	This contains methods that are used for execution of SQL stored procedures.
Connection	This is used to maintain and monitor database sessions. Data access can also be controlled using the transaction locking mechanism.
DatabaseMetaData	This interface provides database information such as the version number, names of the tables and functions supported.
Driver	This interface is used to create Connection objects.
PreparedStatement	This is used to execute pre-compiled SQL statements.
ResultSet	This interface provides methods for retrieving data returned by an SQL statement.
ResultSetMetaData	This interface is used to collect the meta data information associated with the last ResultSet object.
Statement	It is used to execute SQL statements and retrieve data into the ResultSet.

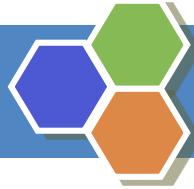


# An Introduction to Design Patterns



143

Copyright © 2015



# Overview

## Part I: Motivation & Concept

- the issue
- what design patterns are
- what they're good for
- how we develop & categorize them





# Overview (cont'd)

## Part II: Application

- use patterns to design a document editor
- demonstrate usage & benefits

## Part III: Wrap-Up

- observations, caveats, & conclusion





## Part I: Motivation & Concept

OOD methods emphasize design notations

Fine for specification, documentation

But OOD is more than just drawing diagrams

Good draftsmen  good designers

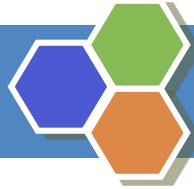
Good OO designers rely on lots of experience

At least as important as syntax

Most powerful reuse is *design* reuse

**NIIT** Match problem to design experience





## Part I: Motivation & Concept (cont'd)

OO systems exhibit recurring structures that promote

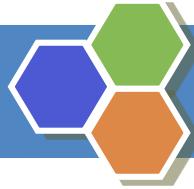
- abstraction
- flexibility
- modularity
- elegance

Therein lies valuable design knowledge

**Problem:**

**capturing, communicating, & applying this  
knowledge**

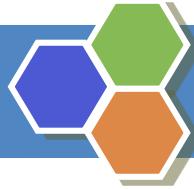




## Part I: Motivation & Concept (cont'd)

- abstracts a recurring design structure
- comprises class and/or object
  - dependencies
  - structures
  - interactions
  - conventions
- names & specifies the design structure explicitly
- distills design experience





## Part I: Motivation & Concept (cont'd)

1. Name
2. Problem (including “forces”)
3. Solution
4. Consequences & trade-offs of application

**Language- & implementation-independent**

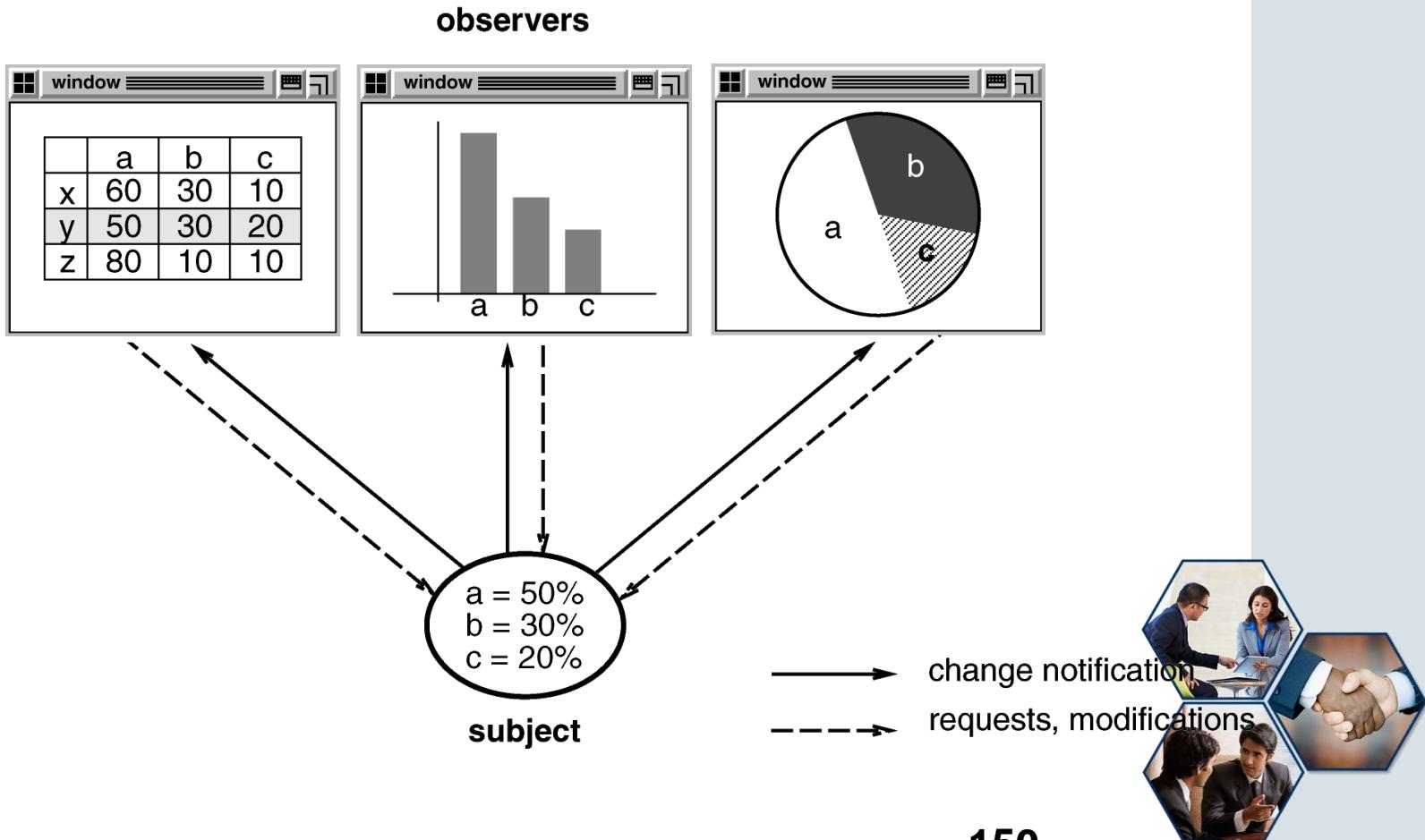
**A “micro-architecture”**

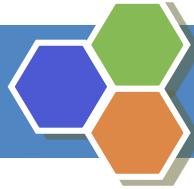
**Adjunct to existing methodologies (RUP, Fusion,  
SCRUM, etc.)**





## Part I: Motivation & Concept (cont'd)





## Part I: Motivation & Concept (cont'd)

### Codify good design

- distill & generalize experience
- aid to novices & experts alike

### Give design structures explicit names

- common vocabulary
- reduced complexity
- greater expressiveness

### Capture & preserve design information

- articulate design decisions succinctly
- improve documentation

### Facilitate restructuring/refactoring

- patterns are interrelated
- additional flexibility





## Part I: Motivation & Concept (cont'd)

		<b>Purpose</b>		
		<b>Creational</b>	<b>Structural</b>	<b>Behavioral</b>
<b>Scope</b>	<b>Class</b>	Factory Method	Adapter (class)	Interpreter Template Method
	<b>Object</b>	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

**Scope: domain over which a pattern applies**  
**Purpose: reflects what a pattern does**





## Part I: Motivation & Concept (cont'd)

### Design Pattern Template (1st half)

NAME scope purpose  
Intent

short description of the pattern & its purpose

### Also Known As

Any aliases this pattern is known by

### Motivation

motivating scenario demonstrating pattern's use

### Applicability

circumstances in which pattern applies

### Structure

graphical representation of the pattern using modified UML notation

### Participants

participating classes and/or objects & their responsibilities

Slide 153 of 92



...

### **Collaborations**

how participants cooperate to carry out their responsibilities

### **Consequences**

the results of application, benefits, liabilities

### **Implementation**

pitfalls, hints, techniques, plus language-dependent issues

### **Sample Code**

sample implementations in C++, Java, C#, Smalltalk, C, etc.

### **Known Uses**

examples drawn from existing systems

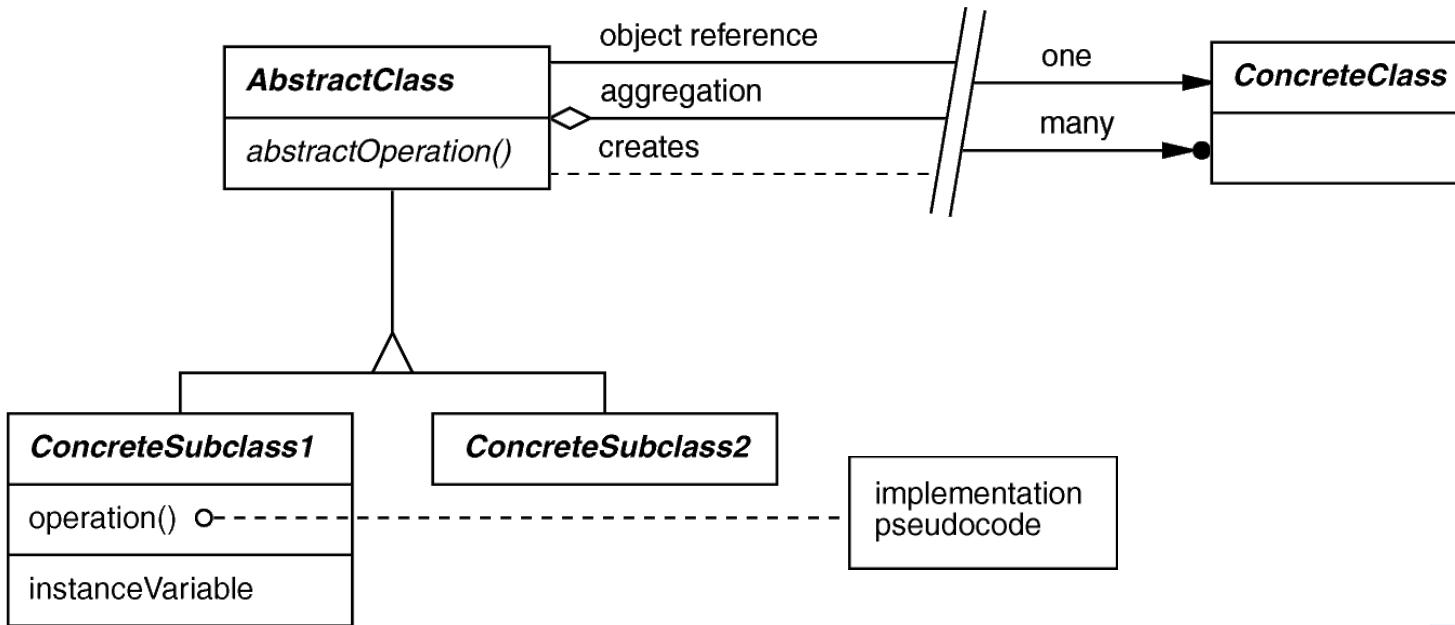
### **Related Patterns**

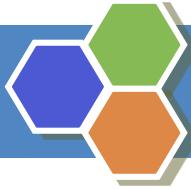
discussion of other patterns that relate to this one





## Part I: Motivation & Concept (cont'd)





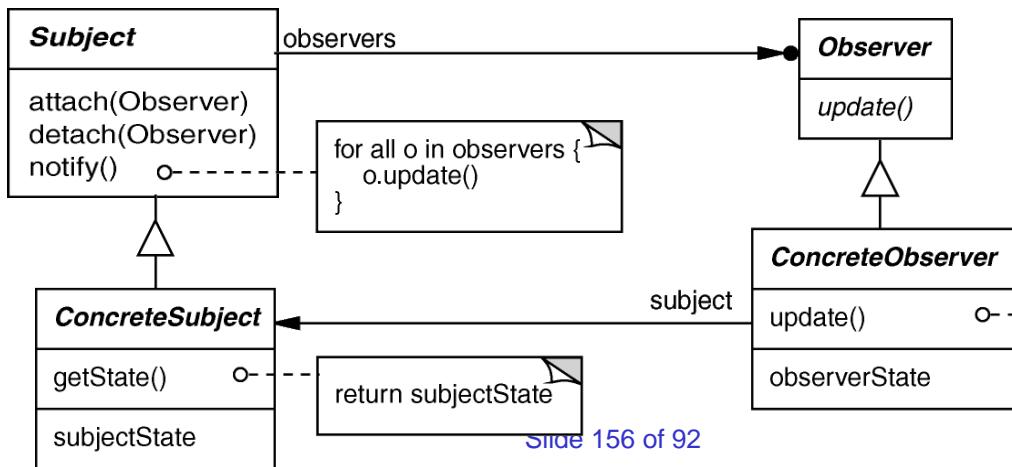
#### Intent

define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated

#### Applicability

- an abstraction has two aspects, one dependent on the other
- a change to one object requires changing untold others
- an object should notify unknown other objects

#### Structure





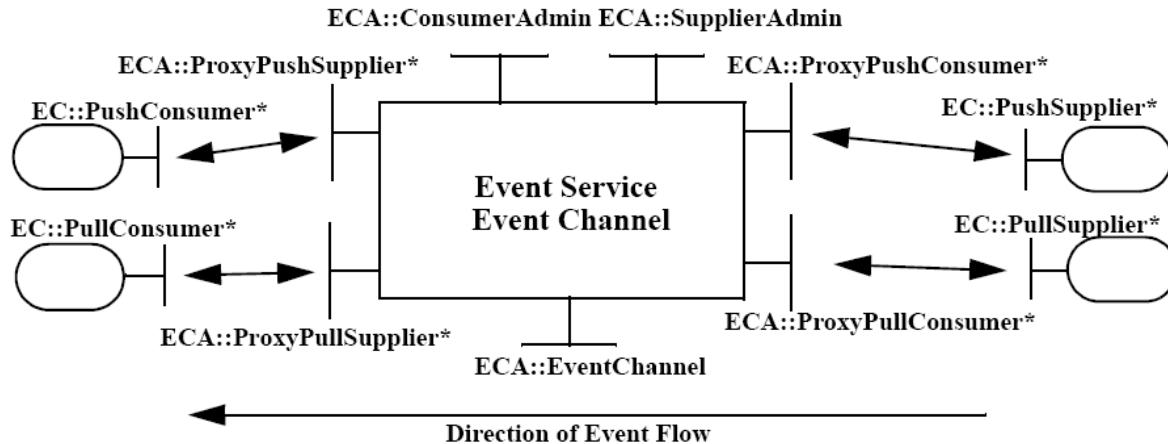
## Motivation & Concept (cont'd)

OBSERVER

object behavioral

```
class ProxyPushConsumer : public ...  
    virtual void push (const CORBA::Any &event) {  
        for (std::vector<PushConsumer>::iterator i  
            (consumers.begin ()); i != consumers.end (); i++)  
            (*i).push (event);  
    }  
}
```

```
class MyPushConsumer : public ...  
    virtual void push  
    (const CORBA::Any &event) { /* consume the event. */ }
```



**CORBA Notification Service**  
example using C++ Standard  
Template Library (STL) iterators  
(which is an example of the  
Iterator pattern from GoF)





## Consequences

- + modularity: subject & observers may vary independently
- + extensibility: can define & add any number of observers
- + customizability: different observers offer different views of subject
- unexpected updates: observers don't know about each other
- update overhead: might need hints or filtering

## Implementation

- subject-observer mapping
- dangling references
- update protocols: the push & pull models
- registering modifications of interest explicitly

## Known Uses

- Smalltalk Model-View-Controller (MVC)
- InterViews (Subjects & Views, Observer/Observable)
- Andrew (Data Objects & Views)
- Pub/sub middleware (e.g., CORBA Notification Service, Java Messaging Service)
- Mailing lists

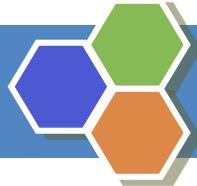




## Part I: Motivation & Concept (cont'd)

- *Design* reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details



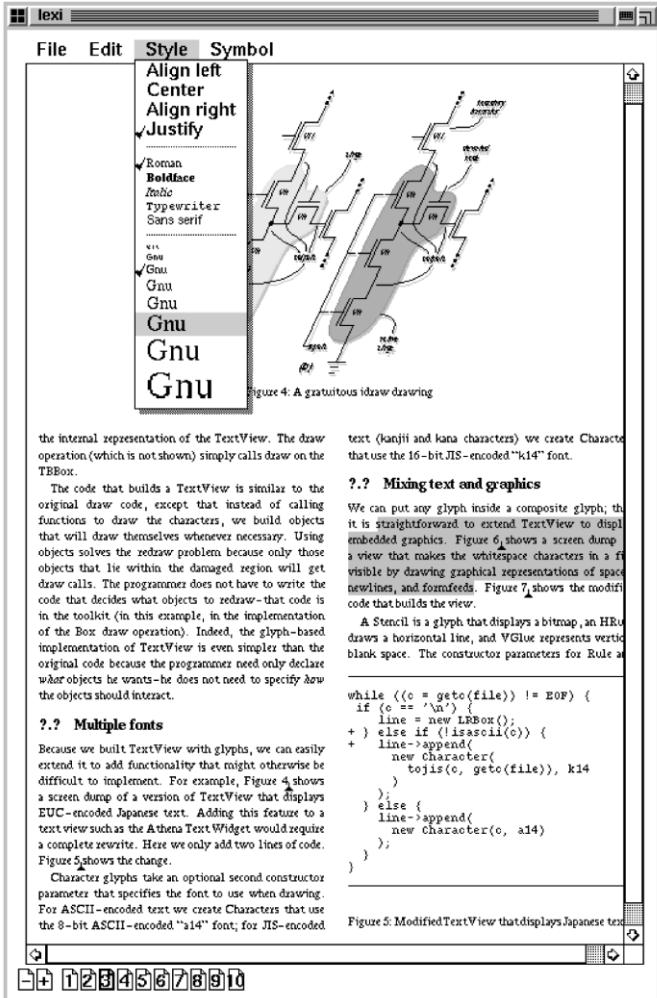


## Part I: Motivation & Concept (cont'd)

- Require significant tedious & error-prone human effort to handcraft pattern implementations *design reuse*
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves some important details unresolved



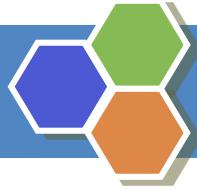
## Part II: Application: Document Editor (Lexi)



## 7 Design Problems

1. Document structure
2. Formatting
3. Embellishment
4. Multiple look & feels
5. Multiple window systems
6. User operations
7. Spelling checking & hyphenation





# Document Structure

## Goals:

- present document's visual aspects
- drawing, hit detection, alignment
- support physical structure  
(e.g., lines, columns)

## Constraints/forces:

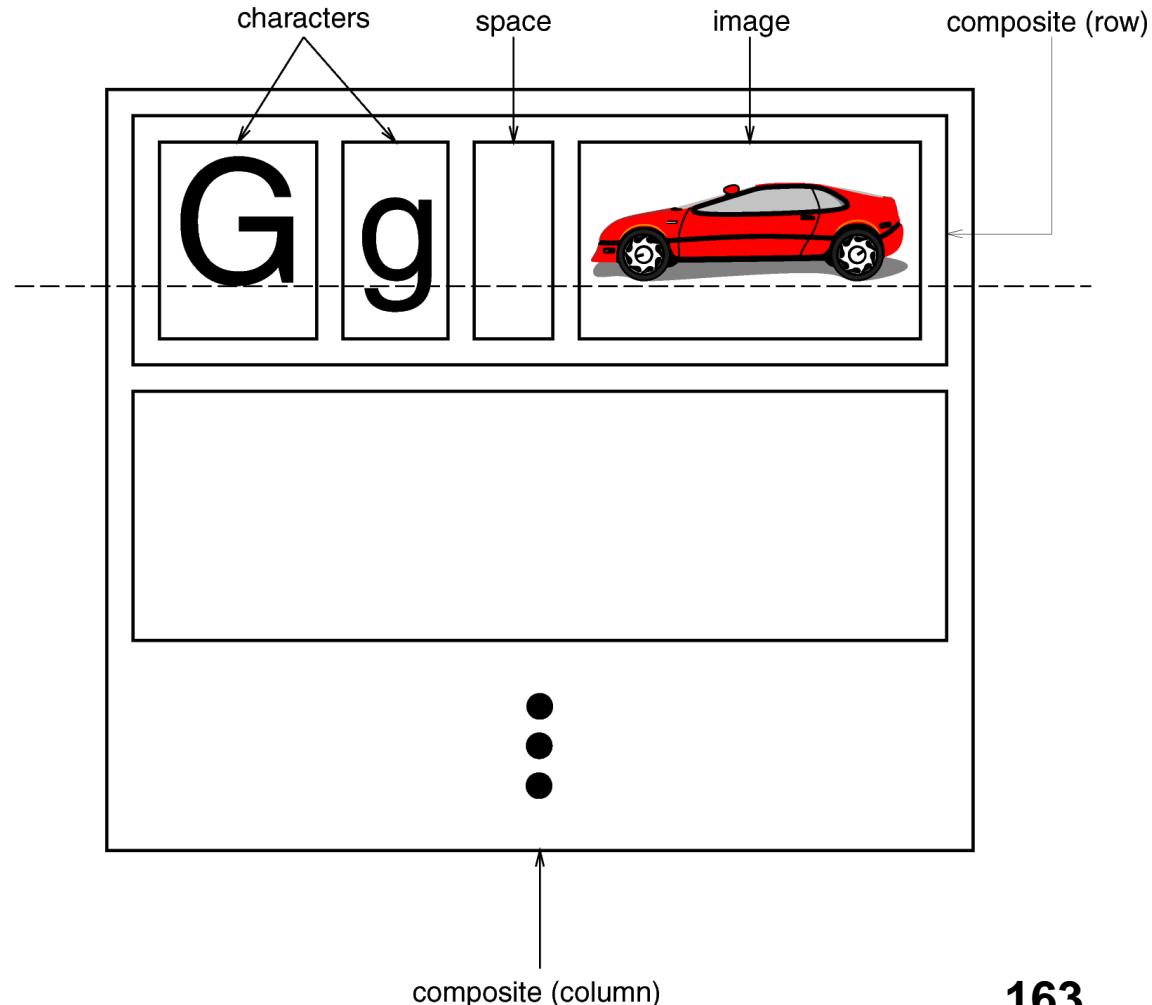
- treat text & graphics uniformly
- no distinction between one & many

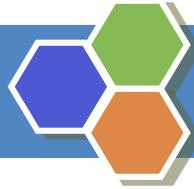




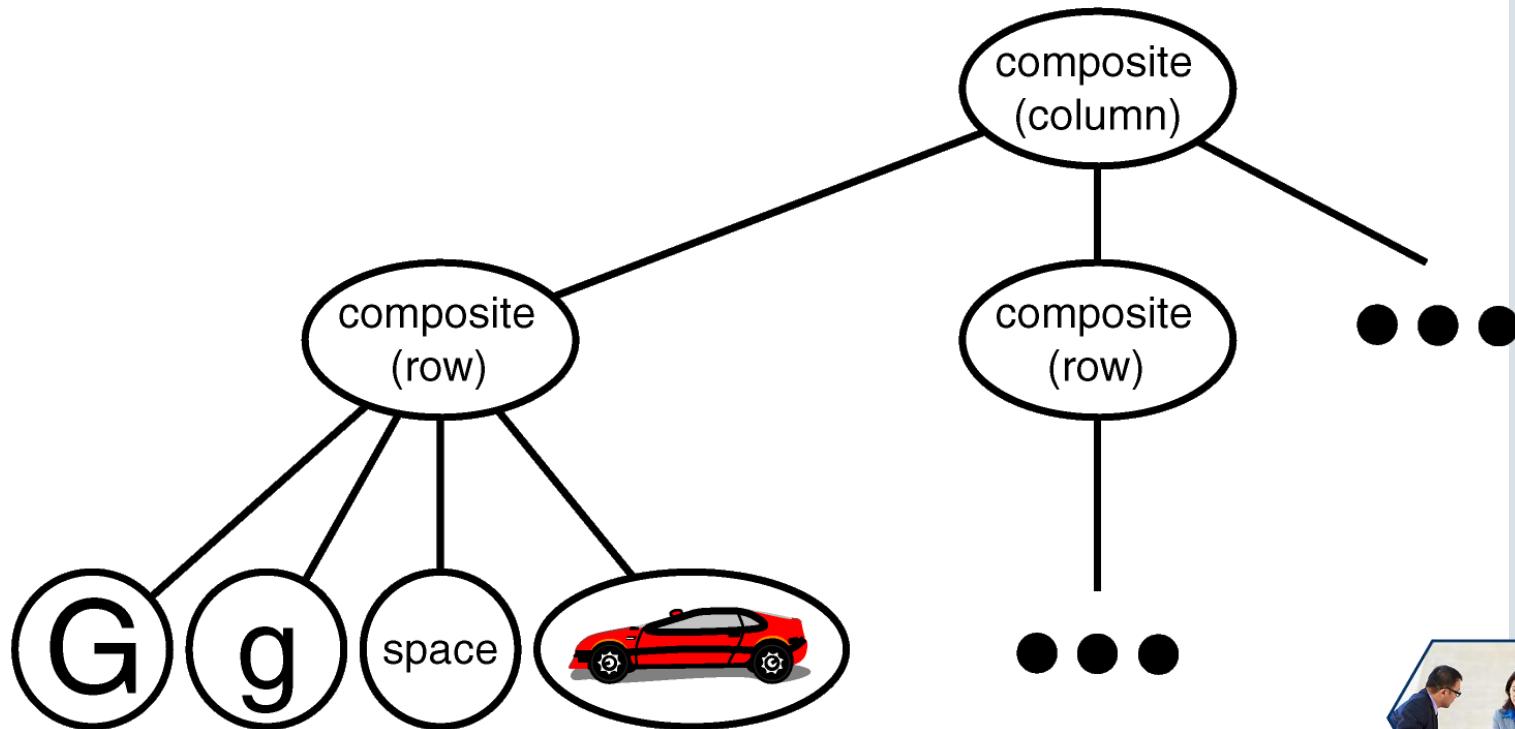
## Document Structure (cont'd)

### Solution: Recursive Composition





## Document Structure (cont'd)





## Document Structure (cont'd) Glyph

Base class for composable graphical objects

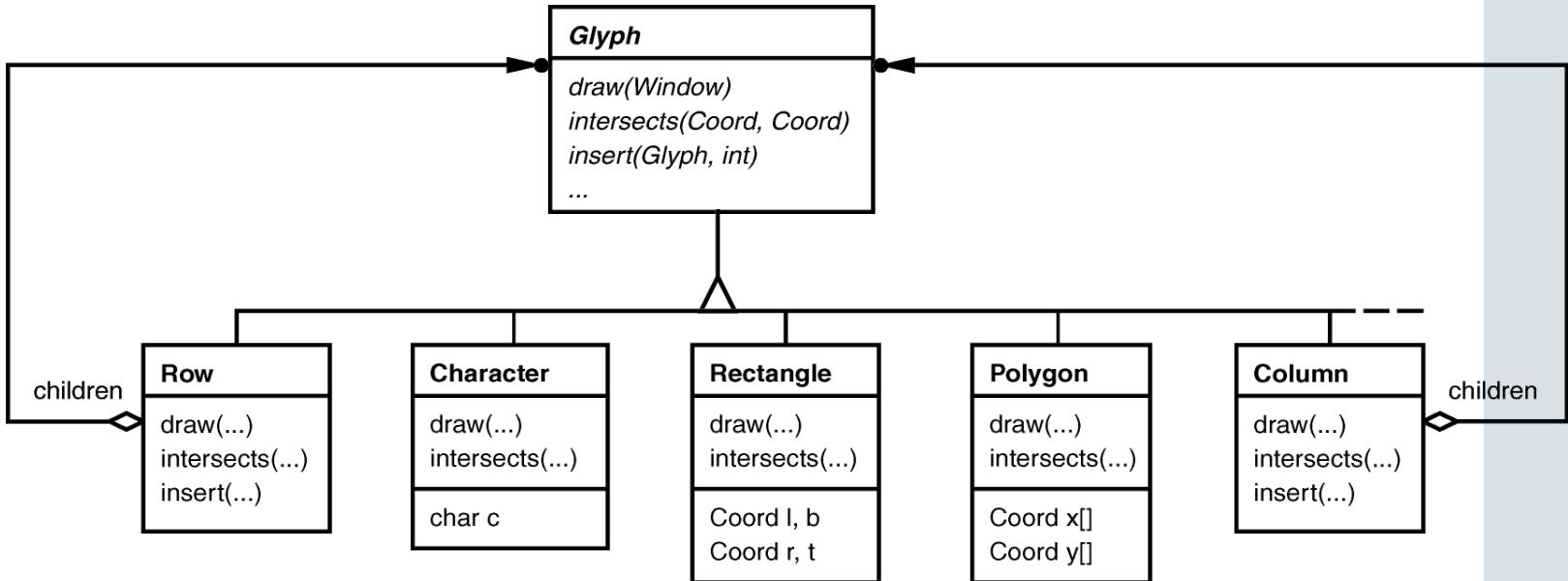
### Basic interface:

Task	Operations
<b>appearance</b>	<code>void draw(Window)</code>
<b>hit detection</b>	<code>boolean intersects(Coord, Coord)</code>
<b>structure</b>	<code>void insert(Glyph)</code> <code>void remove(Glyph)</code> <code>Glyph child(int n)</code> <code>Glyph parent()</code>

Subclasses: Character, Image, Space, Row, Column

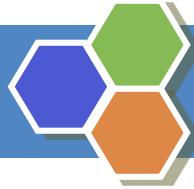


## Document Structure (cont'd)



**Note the inherent recursion in this hierarchy**  
◆ i.e., a Row *is a Glyph* & a Row *also has Glyphs!*





## Intent

treat individual objects & multiple, recursively-composed objects uniformly

## Applicability

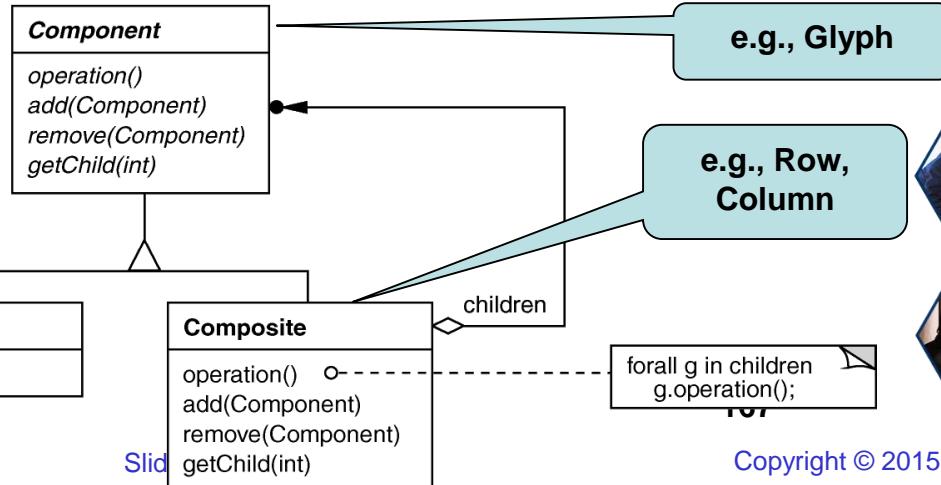
objects must be composed recursively,

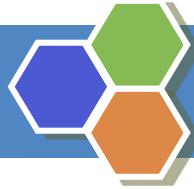
*and* no distinction between individual & composed elements,

*and* objects in structure can be treated uniformly

### Structure

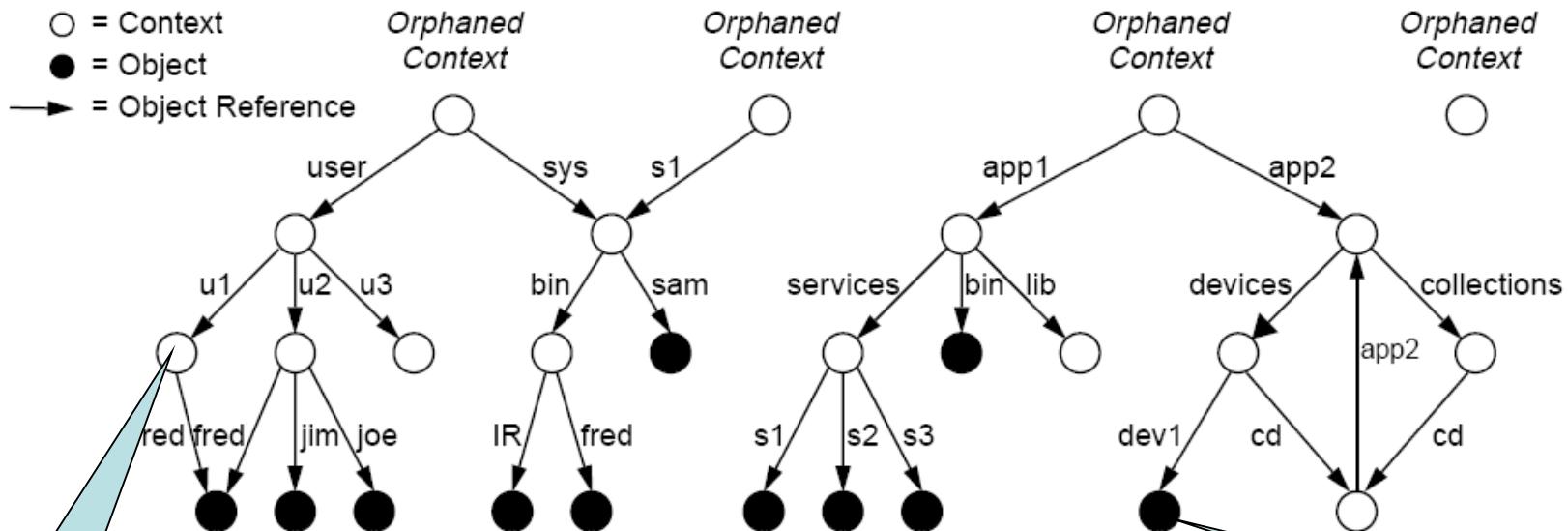
e.g., Character,  
Rectangle, etc.





## Document Structure (cont'd) COMPOSITE

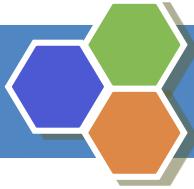
object structural



**Leaf Node**



CORBA Naming Service example using `CosNaming::BindingIterator`  
(which is an example of the “Batch Iterator” pattern compound from  
POSA5)



```
class Glyph {  
public:  
    virtual void  
        draw (const Drawing_Region &) = 0;  
    // ...  
protected:  
    int x_, y_; // Coordinate position.  
};
```

Component

```
class Character : public Glyph {  
public:  
    Character  
        (const std::string &name);  
    // ...  
    virtual void  
        draw (const Drawing_Region &c)  
    { c.draw_text (x_, y_, name_); }  
private:  
    std::string name_;  
};
```

Leaf

```
class Row : public Glyph {  
public:  
    Row (std::vector<Glyph*> children);  
    // ...  
    virtual void  
        draw (const Drawing_Region &c){  
        for (std::vector<Glyph*>::iterator  
            i (children_);  
            i != children_.end ();  
            i++)  
            (*i)->draw (c);  
    }  
    // ...  
private:  
    std::vector<Glyph*> children_;  
    // ...  
};
```

Composite



```

void list_context (CosNaming::NamingContext_ptr nc) {
    CosNaming::BindingIterator_var it; // Iterator reference
    CosNaming::BindingList_var bl; // Binding list
    const CORBA::ULong CHUNK = 100; // Chunk size

    nc->list (CHUNK, bl, it); // Get first chunk
    show_chunk (bl, nc); // Print first chunk
    if (!CORBA::is_nil(it)) { // More bindings?
        while (it->next_n(CHUNK, bl)) // Get next chunk
            show_chunk (bl, nc); // Print chunk
        it->destroy(); // Clean up
    }
}

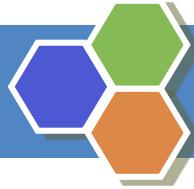
void show_chunk (const CosNaming::BindingList_ptr &bl, // Helper function
                CosNaming::NamingContext_ptr nc) {
    for (CORBA::ULong i = 0; i < bl.length (); ++i) {
        cout << bl[i].binding_name[0].id << "." << bl[i].binding_name[0].kind;

        if (bl[i].binding_type == CosNaming::ncontext) {
            cout << ": context" << endl;
            CORBA::Object_var obj = nc->resolve (bl[i].binding_name);
            list_context (CosNaming::NamingContext::_narrow (obj));
        }
        else cout << ": reference" << endl;
    }
}

```

Handle  
Composite  
Node

Handle  
Leaf  
Node



## Consequences

- + uniformity: treat components the same regardless of complexity
- + extensibility: new Component subclasses work wherever old ones do
- overhead: might need prohibitive numbers of objects

## Implementation

- do Components know their parents?
- uniform interface for both leaves & composites?
- don't allocate storage for children in Component base class
- responsibility for deleting children

## Known Uses

- ET++ Vobjects
- InterViews Glyphs, Styles
- Unidraw Components, MacroCommands
- Directory structures on UNIX & Windows
- Naming Contexts in CORBA
- MIME types in SOAP





# Formatting

## Goals:

- automatic linebreaking, justification

## Constraints/forces:

- support multiple linebreaking algorithms
- don't tightly couple these algorithms with the document structure





## Formatting (cont'd)

### Composer

- base class abstracts linebreaking algorithm
- subclasses for specialized algorithms,  
e.g., **SimpleComposer**, **TeXComposer**

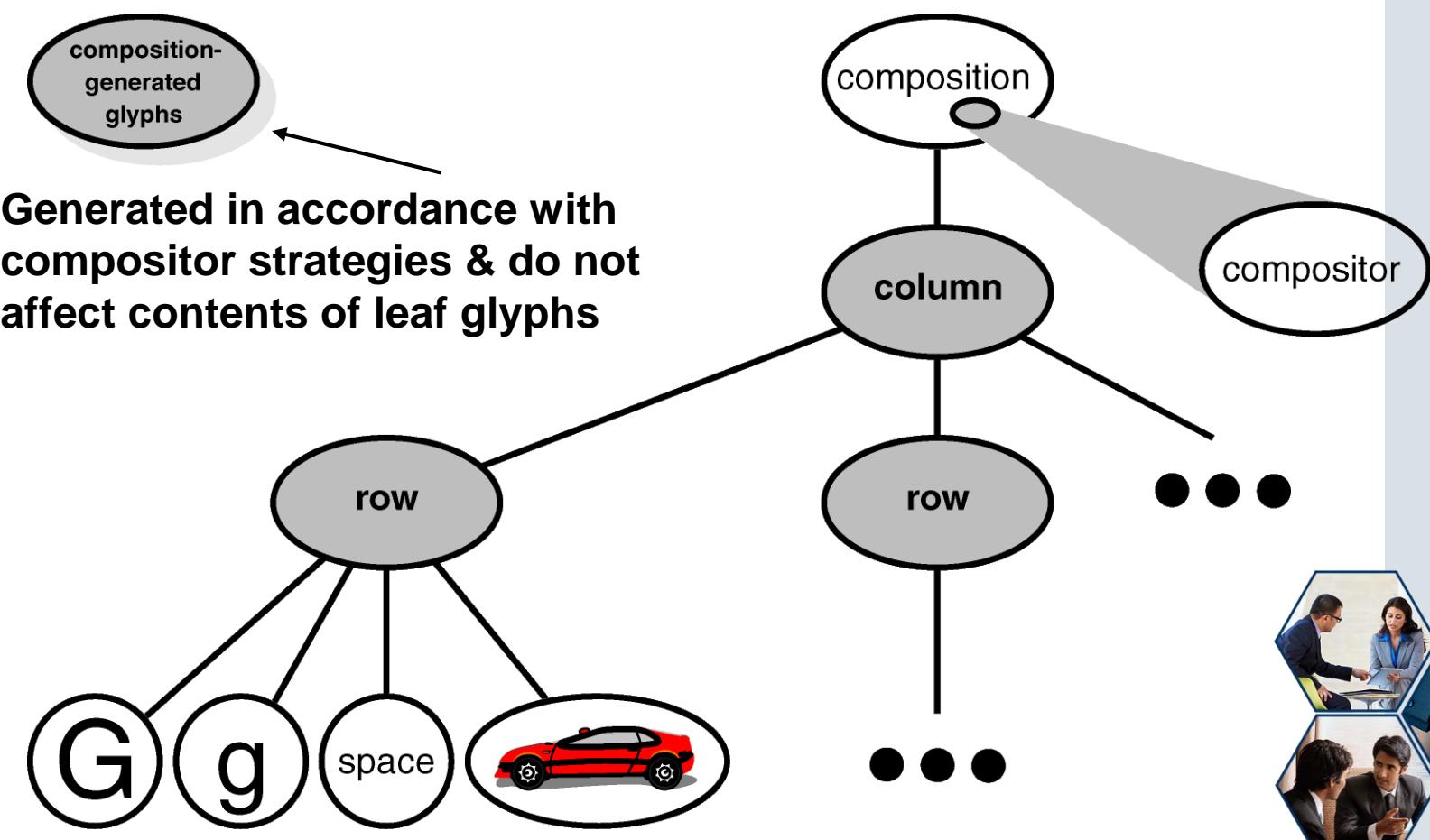
### Composition

- composite glyph (typically representing a column)
- supplied a compositor & leaf glyphs
- creates row-column structure as directed by compositor





## Formatting (cont'd) New Object Structure





Intent

## Formatting (cont'd) STRATEGY

object behavioral

define a family of algorithms, encapsulate each one, & make them

interchangeable to let clients & algorithms vary independently

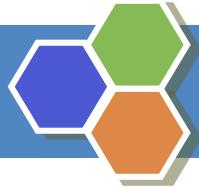
### Applicability

when an object should be configurable with one of many algorithms,

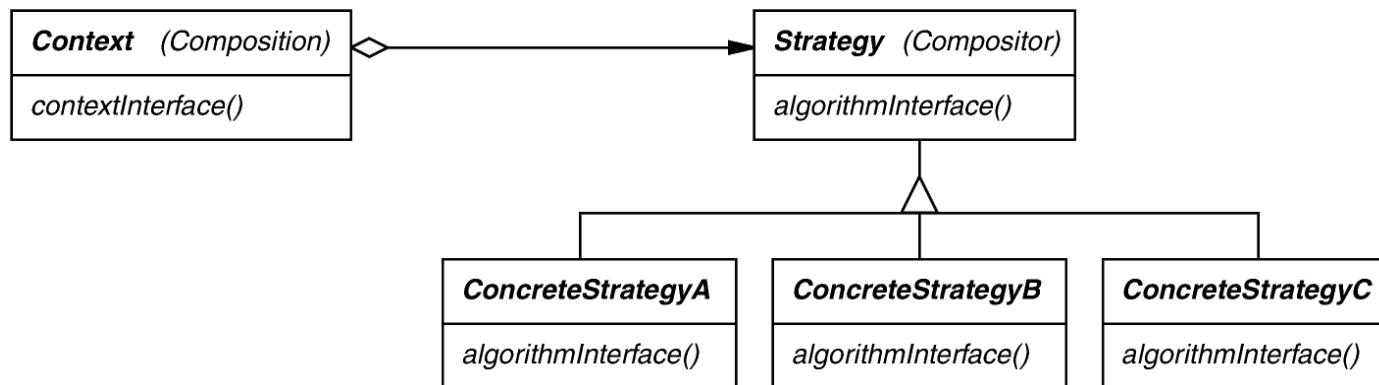
*and* all algorithms can be encapsulated,

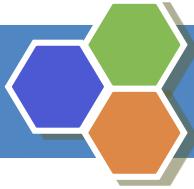
*and* one interface covers all encapsulations





## Structure





```
class Composition : public Glyph {  
public:  
    void perform_composition (const Compositor &compositor,  
                             const std::vector<Glyph*> &leaf_glyphs) {  
        compositor.set_context (*this);  
        for (std::vector<Glyph*>::iterator  
             i (leaf_glyphs);  
             i != leaf_glyphs.end ();  
             ++i) {  
            s->insert (*i);  
            compositor.compose ();  
    };
```

creates row-column structure as directed by compositor

Simple algorithm

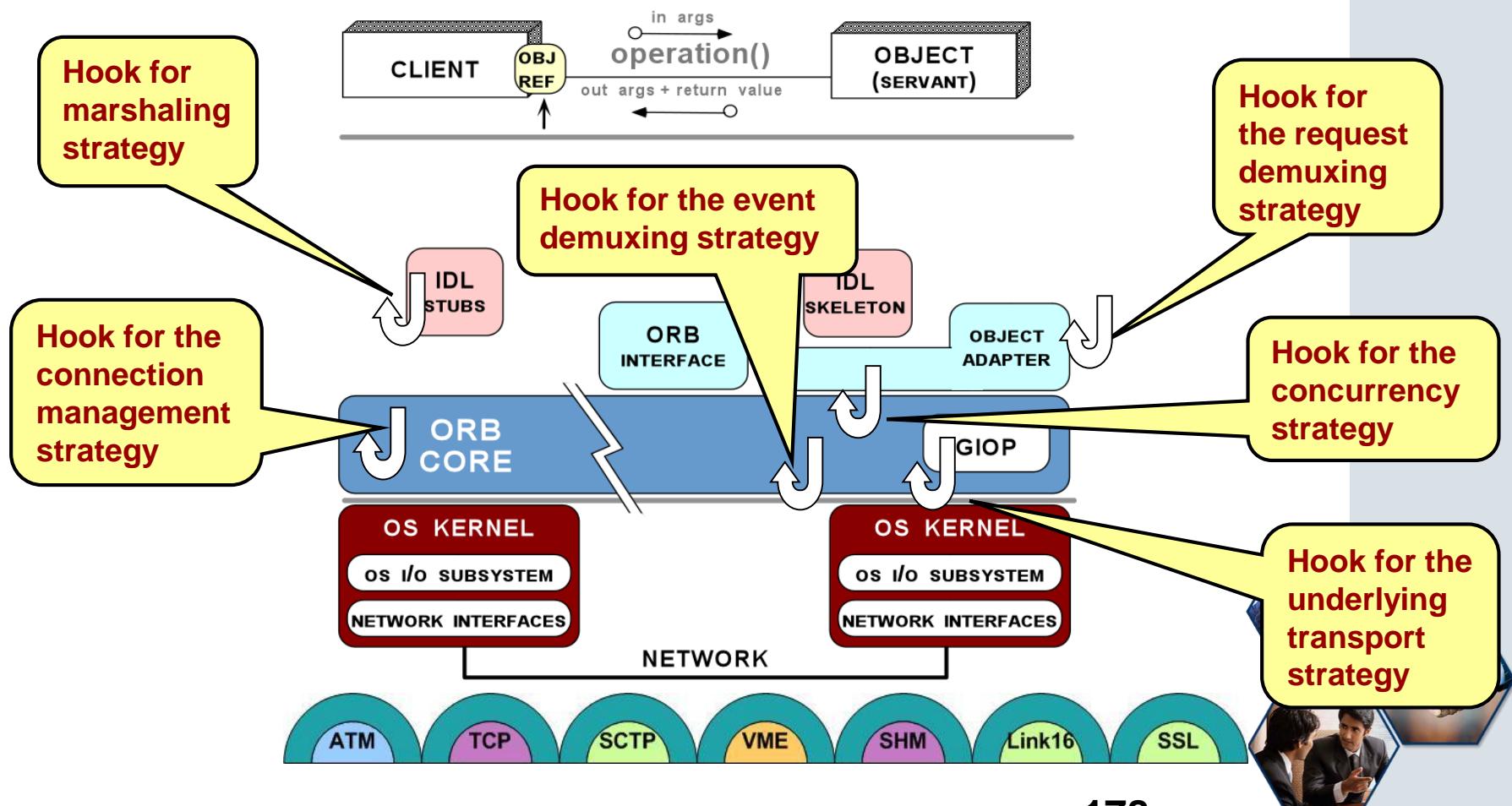
```
class Compositor {  
public:  
    void set_context  
        (Composition &context);  
    virtual void compose () = 0;
```

```
class SimpleCompositor  
    : public Compositor {  
public:  
    virtual void compose ()  
    { /* ... */  
    };
```

Complex algorithm

```
class TexCompositor  
    : public Compositor {  
public:  
    virtual void compose () { /* ... */  
    };
```

## Formatting (cont'd)



N

Strategy can also be applied in distributed systems (e.g., middleware)



## Consequences

- + greater flexibility, reuse
- + can change algorithms dynamically
- strategy creation & communication overhead
- inflexible Strategy interface
- semantic incompatibility of multiple strategies used together

## Implementation

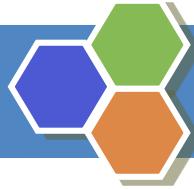
- exchanging information between a Strategy & its context
- static strategy selection via parameterized types

## Known Uses

- InterViews text formatting
- RTL register allocation & scheduling strategies
- ET++SwapsManager calculation engines
- The ACE ORB (TAO) Real-time CORBA middleware

## See Also





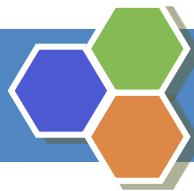
## Intent

- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

```
class Composition : public Glyph {  
public:  
    // Template Method.  
    void perform_composition (const std::vector<Glyph*> &leaf_glyphs) {  
        set_context (*this);  
        for (std::vector<Glyph*>::iterator i (leaf_glyphs);  
             i != leaf_glyphs.end (); i++) {  
            insert (*i);  
            compose ();  
        }  
    }  
    virtual void compose () = 0; // Hook Method.  
};  
  
class Simple_Composition : public Composition {  
    virtual void compose () { /* ... */ }  
};  
  
class Tex_Composition : public Composition {  
    virtual void compose () { /* ... */ }  
};
```

180



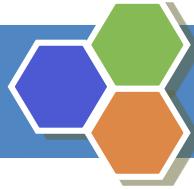


## Intent

- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses

```
class Base_Class {  
public:  
    // Template Method.  
    void template_method (void) {  
        hook_method_1 ();  
        hook_method_2 ();  
        // ...  
    }  
    virtual void hook_method_1 () = 0;  
    virtual void hook_method_2 () = 0;  
};  
  
class Derived_Class_1 : public Base_Class {  
    virtual void hook_method_2 () { /* ... */ }  
};  
  
class Derived_Class_2 : public Base_Class {  
    virtual void hook_method_1 () { /* ... */ }  
    virtual void hook_method_2 () { /* ... */ }  
};
```





# Embellishment

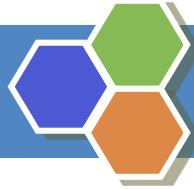
## Goals:

- add a frame around text composition
- add scrolling capability

## Constraints/forces:

- embellishments should be reusable without subclassing, i.e., so they can be added dynamically at runtime
- should go unnoticed by clients





# Embellishment (cont'd)

## Monoglyph

- base class for glyphs having *one* child
- operations on MonoGlyph (ultimately) pass through to child

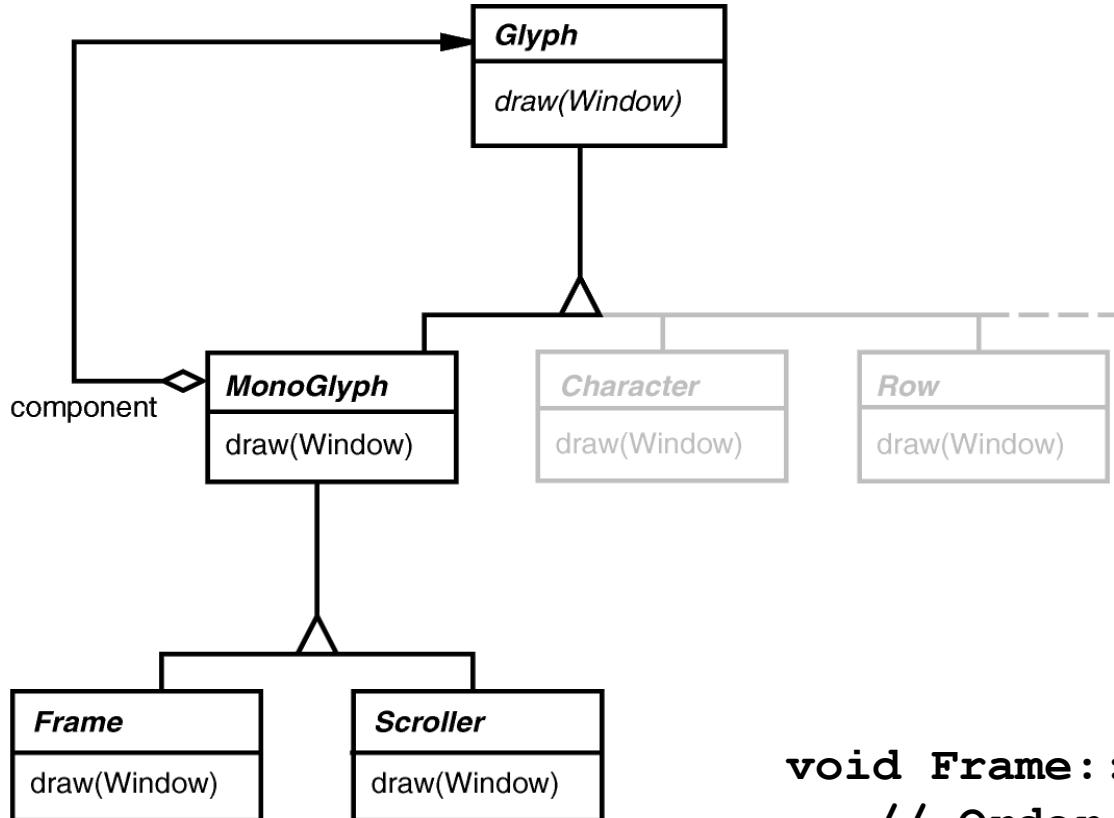
MonoGlyph subclasses:

- **Frame**: adds a border of specified width
- **Scroller**: scrolls/clips child, adds scrollbars





## Embellishment (cont'd) MonoGlyph Hierarchy



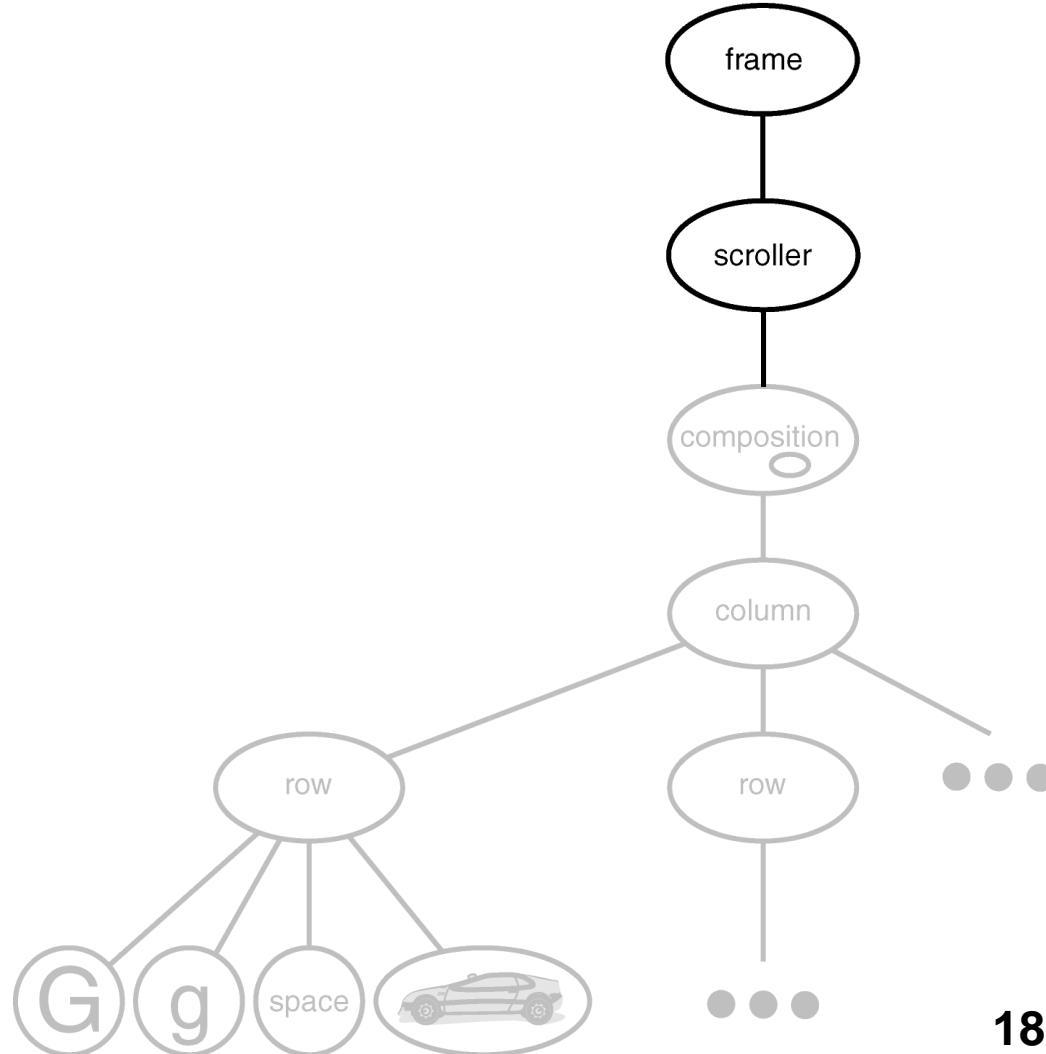
```
void MonoGlyph::draw (Window &w) {
    component->draw (w);
}
```

```
void Frame::draw (Window &w) {
    // Order may be important!
    MonoGlyph::draw (w);
    drawFrame (w);
}
```





## Embellishment (cont'd)



Slide 185 of 92



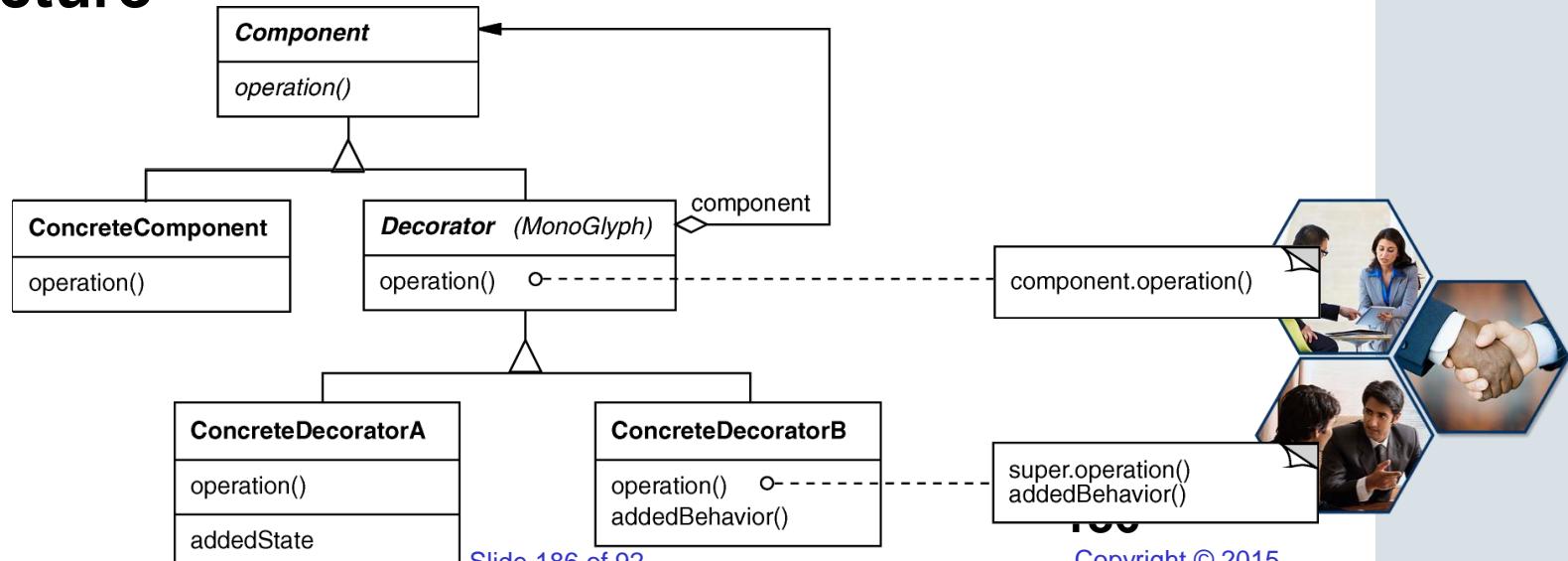
### Intent

Transparently augment objects with new responsibilities dynamically

### Applicability

- when extension by subclassing is impractical
- for responsibilities that can be added & withdrawn dynamically

### Structure





## Embellishment (cont'd)

```
size_t request_count;

void *worker_task (void *) {
    request_count++;

    // ... process the request
}

ACE_Thread_Mutex m;
size_t request_count;

void *worker_task (void *) {
{
    ACE_Guard <ACE_Thread_Mutex> g (m);
    request_count++;
}

    // ... process the request
}
```

```
ACE_Thread_Mutex m;
size_t request_count;

void *worker_task (void *) {
    m.acquire ();
    request_count++;
    m.release ();

    // ... process the request
}
```





## Embellishment (cont'd)

```
Atomic_Op<size_t, ACE_Thread_Mutex> request_count;

void *worker_task (void *) {
    request_count++;

    // ... process the request
}

template <typename T, typename LOCK>
class Atomic_Op {
public:
    void operator++ () {
        ACE_Guard <LOCK> g (m_);
        count_++;
    // ...
    }
private:
    T count_;
    LOCK m_;
};

```





### Consequences

- + responsibilities can be added/removed at run-time
- + avoids subclass explosion
- + recursive nesting allows multiple responsibilities
- interface occlusion
- identity crisis
- composition of decorators is hard if there are side-effects

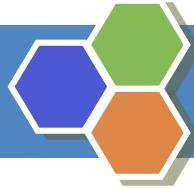
### Implementation

- interface conformance
- use a lightweight, abstract base class for Decorator
- heavyweight base classes make Strategy more attractive

### Known Uses

- embellishment objects from most OO-GUI toolkits
- ParcPlace PassivityWrapper
- InterViews DebuggingGlyph
- Java I/O classes
- ACE\_Atomic\_Op





# Multiple Look & Feels

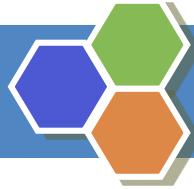
## Goals:

- support multiple look & feel standards
- generic, Motif, Swing, PM, Macintosh, Windows, ...
- extensible for future standards

## Constraints/forces:

- don't recode existing widgets or clients
- switch look & feel without recompiling





## Multiple Look & Feels (cont'd)

Instead of

```
MotifScrollbar *sb = new MotifScrollbar();
```

use

```
Scrollbar *sb = factory->createScrollbar();
```

where **factory** is an instance of **MotifFactory**

- BTW, this begs the question of who created the **factory**!





# Multiple Look & Feels (cont'd)

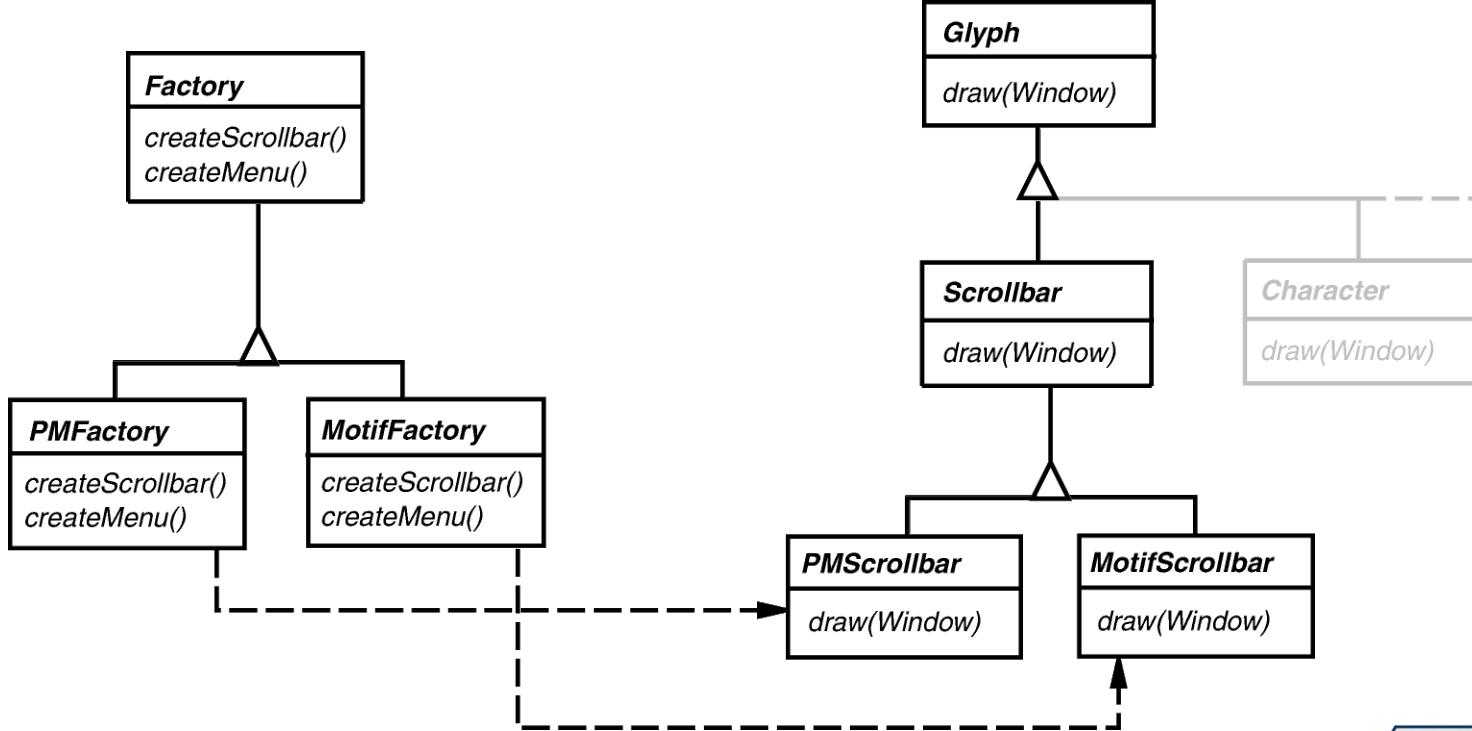
- defines “manufacturing interface”
- subclasses produce specific products
- subclass instance chosen at run-time

```
// This class is essentially a Java interface
class GUIFactory {
public:
    virtual Scrollbar *createScrollbar() = 0;
    virtual Menu *createMenu() = 0;
    ...
};
```





## Multiple Look & Feels (cont'd)



```
Scrollbar *MotifFactory::createScrollBar () {
    return new MotifScrollbar();
}
Scrollbar *PMFactory::createScrollBar () {
    return new PMSScrollbar();
```





# Multiple Look & Feels (cont'd)

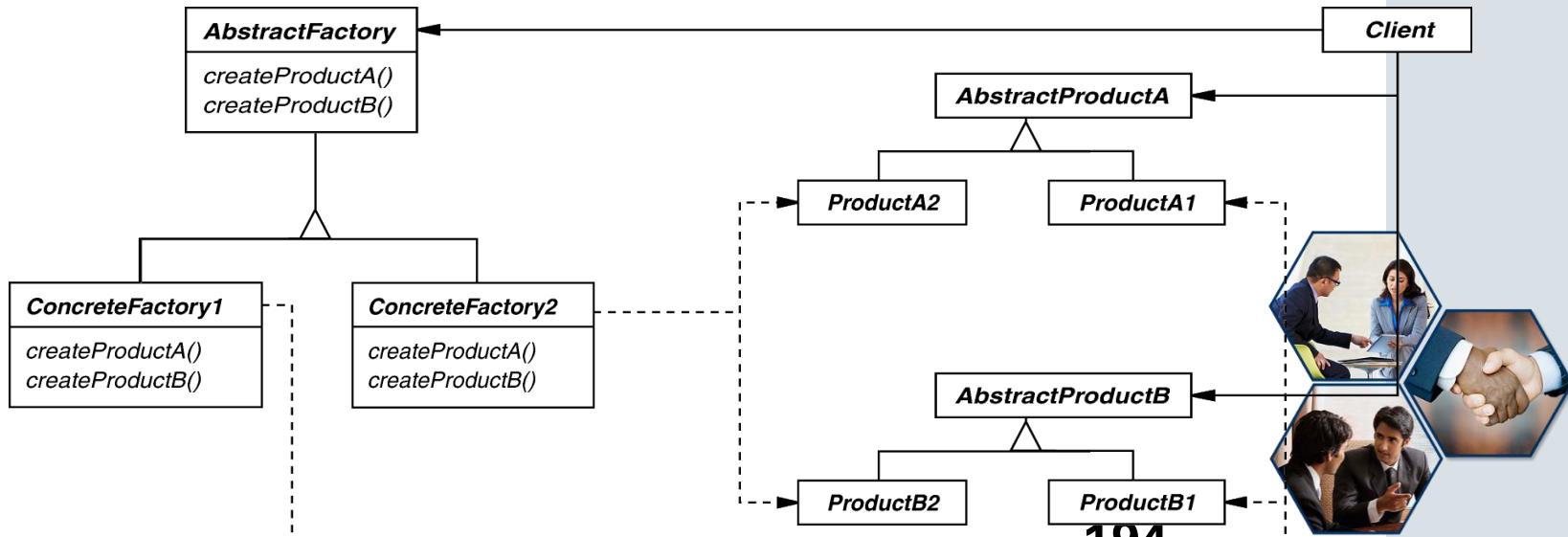
## Intent

create families of related objects without specifying subclass names

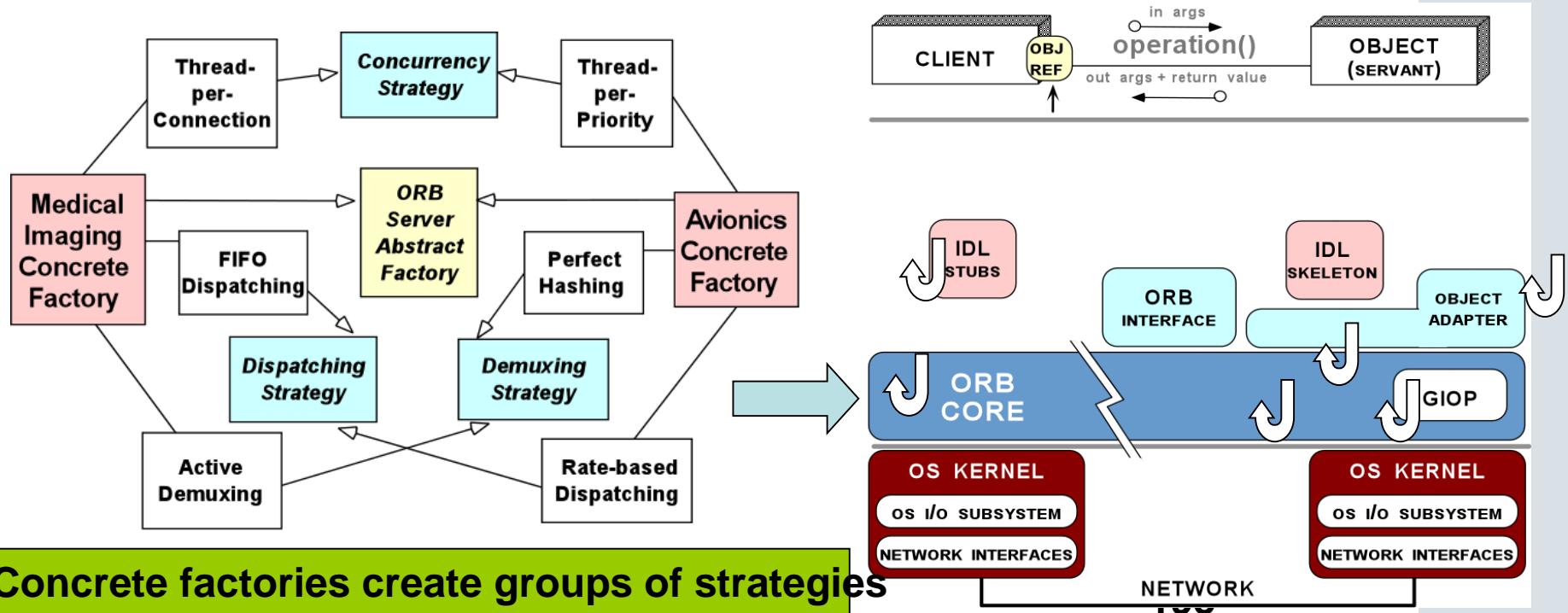
## Applicability

when clients cannot anticipate groups of classes to instantiate

## Structure

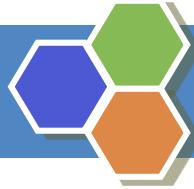


# Multiple Look & Feels (cont'd)



**Concrete factories create groups of strategies**





## Multiple Look & Feels (cont'd) ABSTRACT FACTORY (cont'd) object creational

### Consequences

- + flexibility: removes type (i.e., subclass) dependencies from clients
- + abstraction & semantic checking: hides product's composition
- hard to extend factory interface to create new products

### Implementation

- parameterization as a way of controlling interface size
- configuration with Prototypes, i.e., determines who creates the factories
- abstract factories are essentially groups of factory methods

### Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- The ACE ORB (TAO)





# Multiple Window Systems

## Goals:

- make composition appear in a window
- support multiple window systems

## Constraints/forces:

- minimize window system dependencies in application & framework code



# Window

- user-level window abstraction
- displays a glyph (structure)
- window system-independent
- task-related subclasses  
(e.g., IconWindow, PopupWindow)





## Multiple Window Systems (cont'd)

```
class Window {  
public:  
    ...  
    void iconify();           // window-management  
    void raise();  
    ...  
    void drawLine(...);      // device-independent  
    void drawText(...);      // graphics interface  
    ...  
};
```



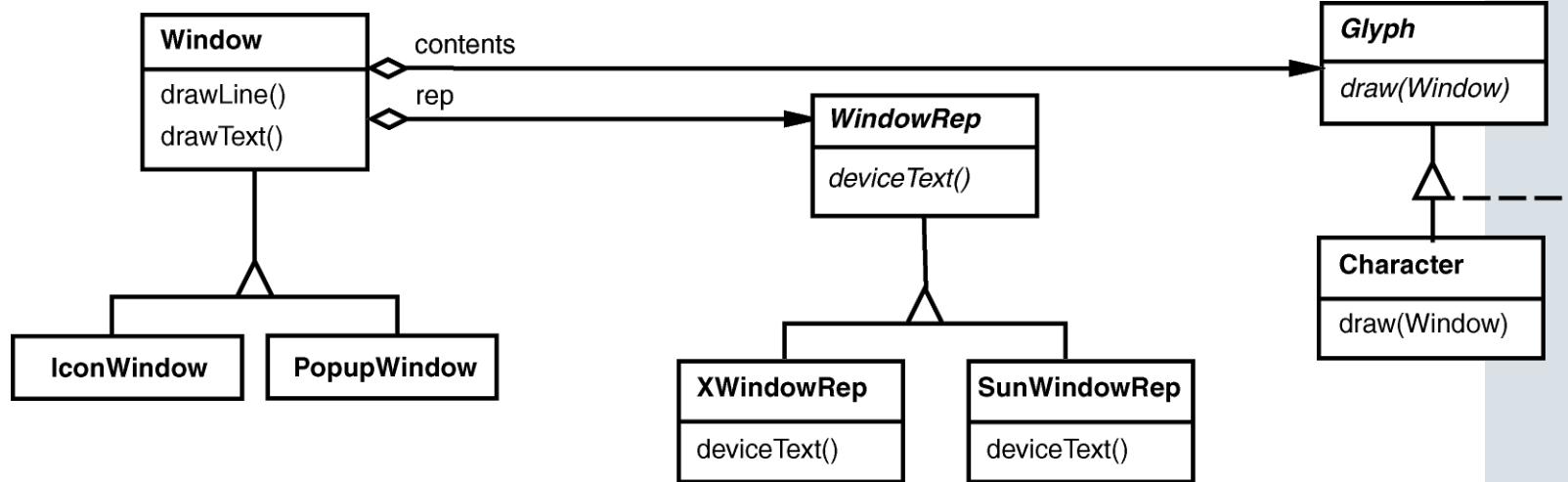
- abstract implementation interface
- encapsulates window system dependencies
- window systems-specific subclasses  
(e.g., XWindowRep, SunWindowRep)

**An Abstract Factory can  
produce the right  
WindowRep!**





# Multiple Window Systems (cont'd)



```
void Character::draw (Window &w) {
    w.drawText(...);
}

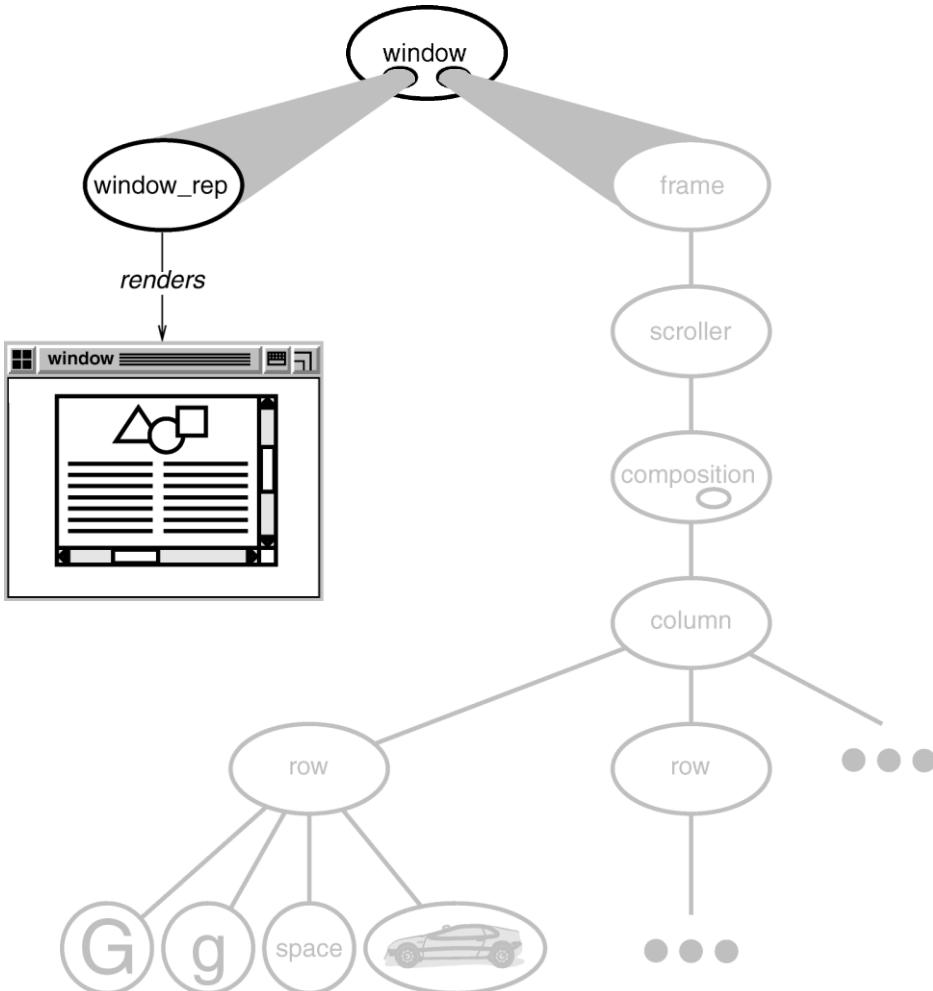
void Window::drawText (...) {
    rep->deviceText(...);
}

void XWindowRep::deviceText (...) {
    XText(...);
```





## Multiple Window Systems (cont'd)



**Note the decoupling between the logical structure of the contents in a window from the physical rendering of the contents in the window**





# Multiple Window Systems (cont'd)

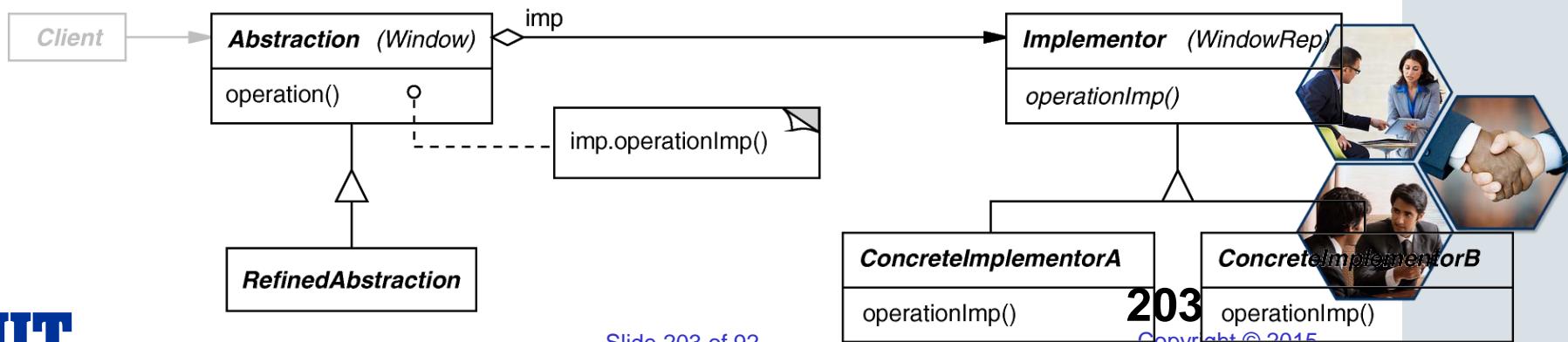
## Intent

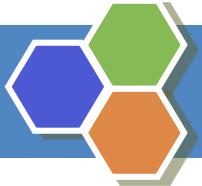
separate a (logical) abstraction interface from its (physical) implementation(s)

## Applicability

- when interface & implementation should vary independently
- require a uniform interface to interchangeable class hierarchies

## Structure





# Multiple Window Systems (cont'd)

## Consequences

- + abstraction interface & implementation are independent
- + implementations can vary dynamically
- one-size-fits-all Abstraction & Implementor interfaces

## Implementation

- sharing Implementors & reference counting
- creating the right Implementor (often use factories)

## Known Uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- AWT Component/ComponentPeer





# User Operations

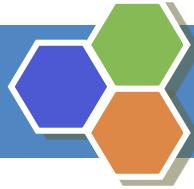
## Goals:

- support execution of user operations
- support unlimited-level undo/redo

## Constraints/forces:

- scattered operation implementations
- must store undo state
- not all operations are undoable





# User Operations (cont'd)

A **Command** encapsulates

- an operation (`execute()`)
- an inverse operation (`unexecute()`)
- a operation for testing reversibility  
(`boolean reversible()`)
- state for (un)doing the operation

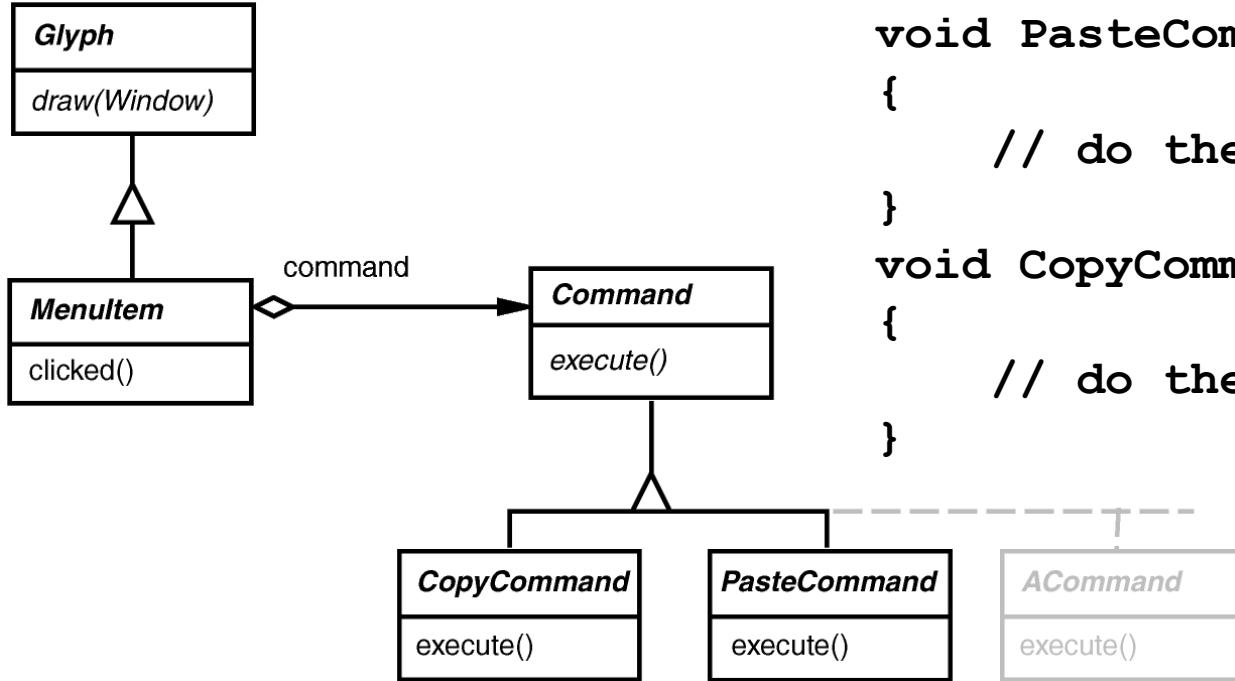
Command may

- implement the operations itself, or
- delegate them to other object(s)





## User Operations (cont'd)



```
void MenuItem::clicked ()  
{  
    command->execute ();  
}
```

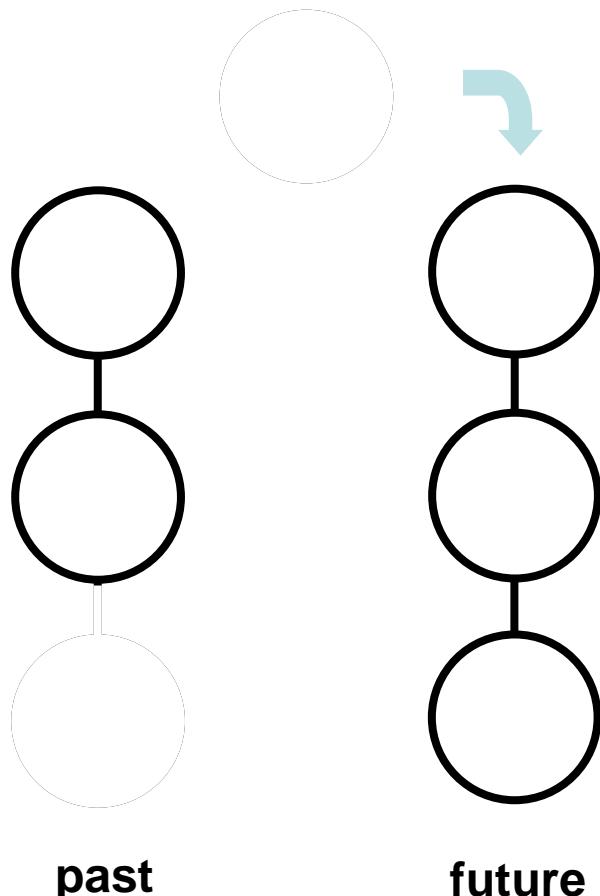
```
void PasteCommand::execute ()  
{  
    // do the paste  
}  
void CopyCommand::execute ()  
{  
    // do the copy  
}
```



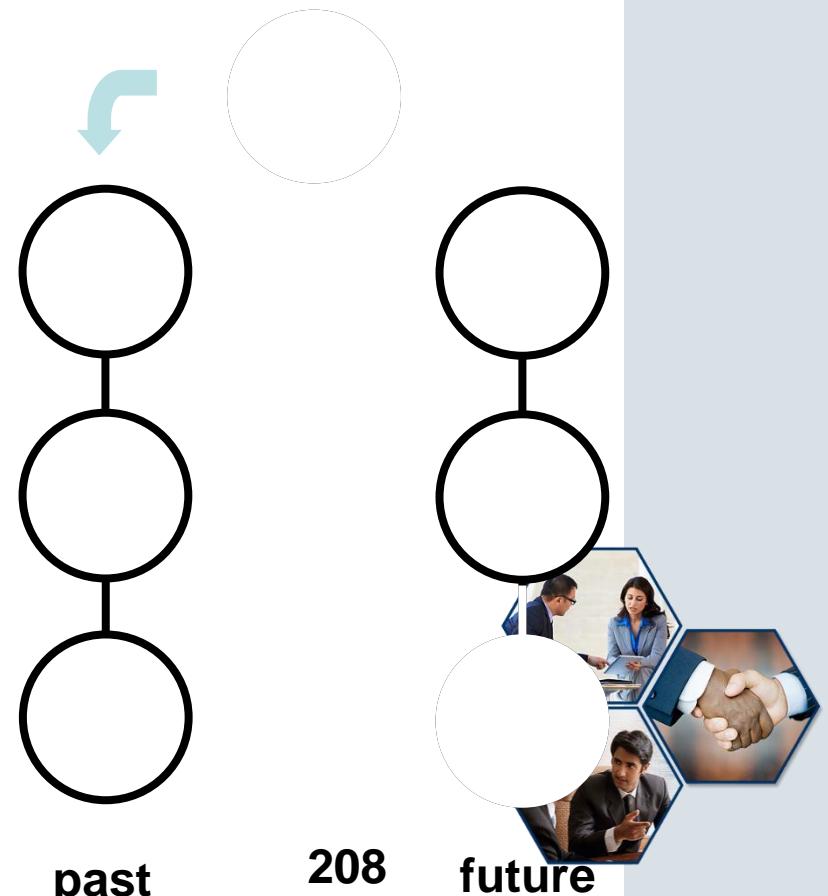


## User Operations (cont'd)

**Undo:**



**Redo:**





# User Operations (cont'd)

## Intent

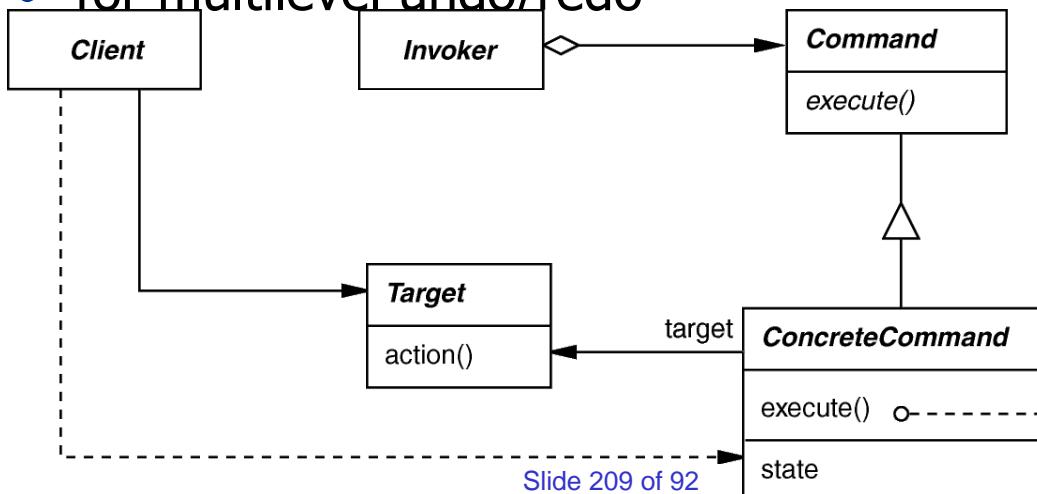
encapsulate the request for a service

## Applicability

- to parameterize objects with an action to perform
- to specify, queue, & execute requests at different times

## Structure

- for multilevel undo/redo

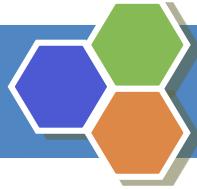


20

target.action()

Copyright © 2015





# User Operations (cont'd)

## Consequences

- + abstracts executor of a service
- + supports arbitrary-level undo-redo
- + composition yields macro-commands
- might result in lots of trivial command subclasses

## Implementation

- copying a command before putting it on a history list
- handling hysteresis
- supporting transactions

## Known Uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- Emacs





# Spelling Checking & Hyphenation

## Goals:

- analyze text for spelling errors
- introduce potential hyphenation sites

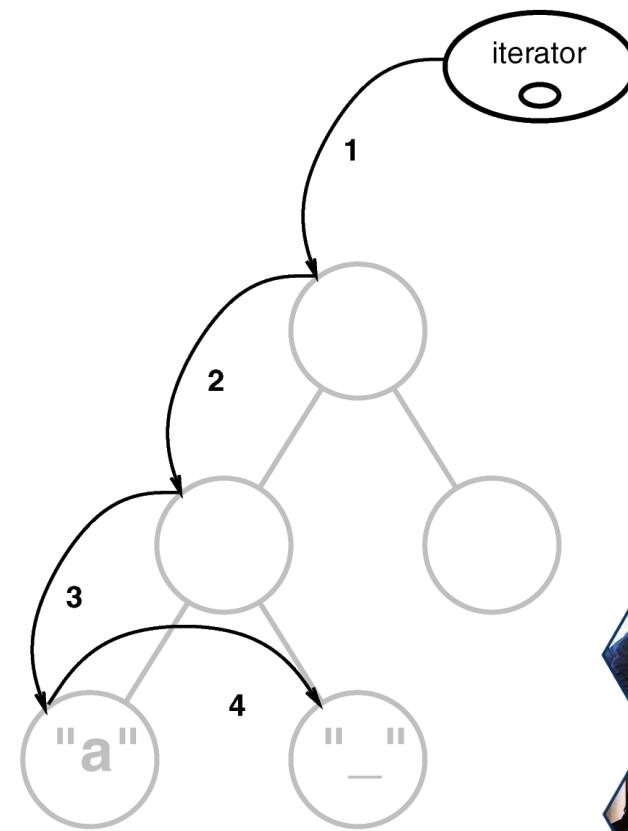
## Constraints/forces:

- support multiple algorithms
- don't tightly couple algorithms with document structure



### Iterator

- encapsulates a traversal algorithm without exposing representation details to callers
- uses Glyph's child enumeration operation
- This is an example of a “preorder iterator”





# Spelling Checking & Hyphenation (cont'd)

## Intent

access elements of a container without exposing its representation

## Applicability

- require multiple traversal algorithms over a container
- require a uniform traversal interface over different containers
- when container classes & traversal algorithm must vary independently

## Structure



Iterators are used heavily in the C++ Standard Template Library (STL)

```
int main (int argc, char *argv[]) {  
    vector<string> args;  
    for (int i = 0; i < argc; i++)  
        args.push_back (string (argv[i]));  
    for (vector<string>::iterator i (args.begin ()) ;  
         i != args.end () ;  
         i++)  
        cout << *i;  
    cout << endl;  
    return 0;  
}  
  
for (Glyph::iterator i = glyphs.begin () ;  
     i != glyphs.end () ;  
     i++)  
    ...
```

The same iterator pattern can be applied  
to any STL container!





## Spelling Checking & Hyphenation (cont'd) ITERATOR (cont'd) object behavioral

### Consequences

- + flexibility: aggregate & traversal are independent
- + multiple iterators & multiple traversal algorithms
- additional communication overhead between iterator & aggregate

### Implementation

- internal versus external iterators
- violating the object structure's encapsulation
- robust iterators
- synchronization overhead in multi-threaded programs
- batching in distributed & concurrent programs

### Known Uses

- C++ STL iterators
- JDK Enumeration, Iterator





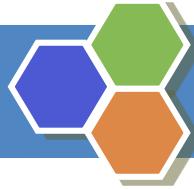
# Spelling Checking & Hyphenation (cont'd)

- defines action(s) at each step of traversal
- avoids wiring action(s) into Glyphs
- iterator calls glyph's **accept(visitor)** at each node
- **accept()** calls back on visitor (a form of "static polymorphism" based on method overloading by type)

```
void Character::accept (Visitor &v) { v.visit (*this); }
```

```
class Visitor {  
public:  
    virtual void visit (Character &);  
    virtual void visit (Rectangle &);  
    virtual void visit (Row &);  
    // etc. for all relevant Glyph subclasses
```



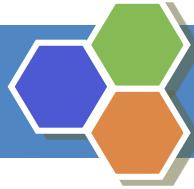


# Spelling Checking & Hyphenation (contd)

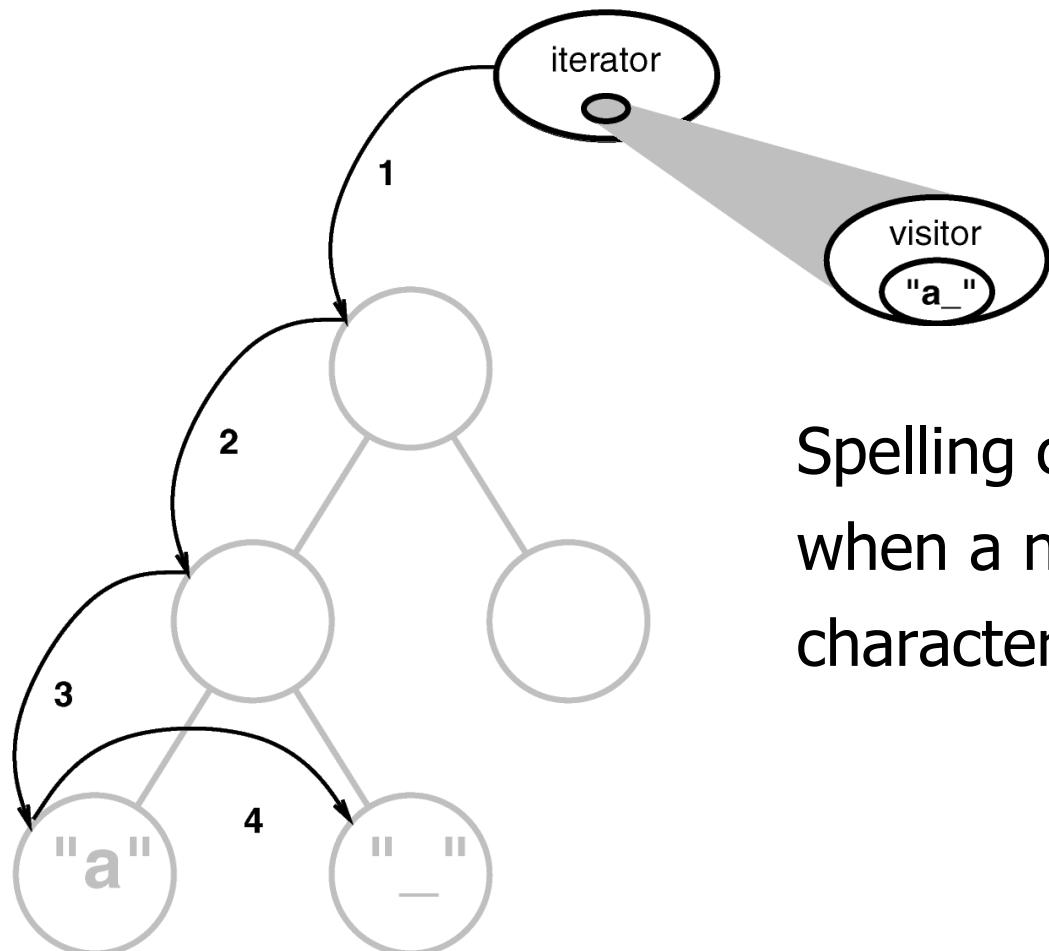
- gets character code from each character glyph  
Can define `getCharCode()` operation just on `Character()` class
- checks words accumulated from character glyphs
- combine with **PreorderIterator**

```
class SpellCheckerVisitor : public Visitor {  
public:  
    virtual void visit (Character &);  
    virtual void visit (Rectangle &);  
    virtual void visit (Row &);  
    // etc. for all relevant Glyph subclasses  
private:  
    std::string accumulator_;  
};
```





# Spelling Checking & Hyphenation (cont'd)



Spelling check performed  
when a nonalphabetic  
character it reached

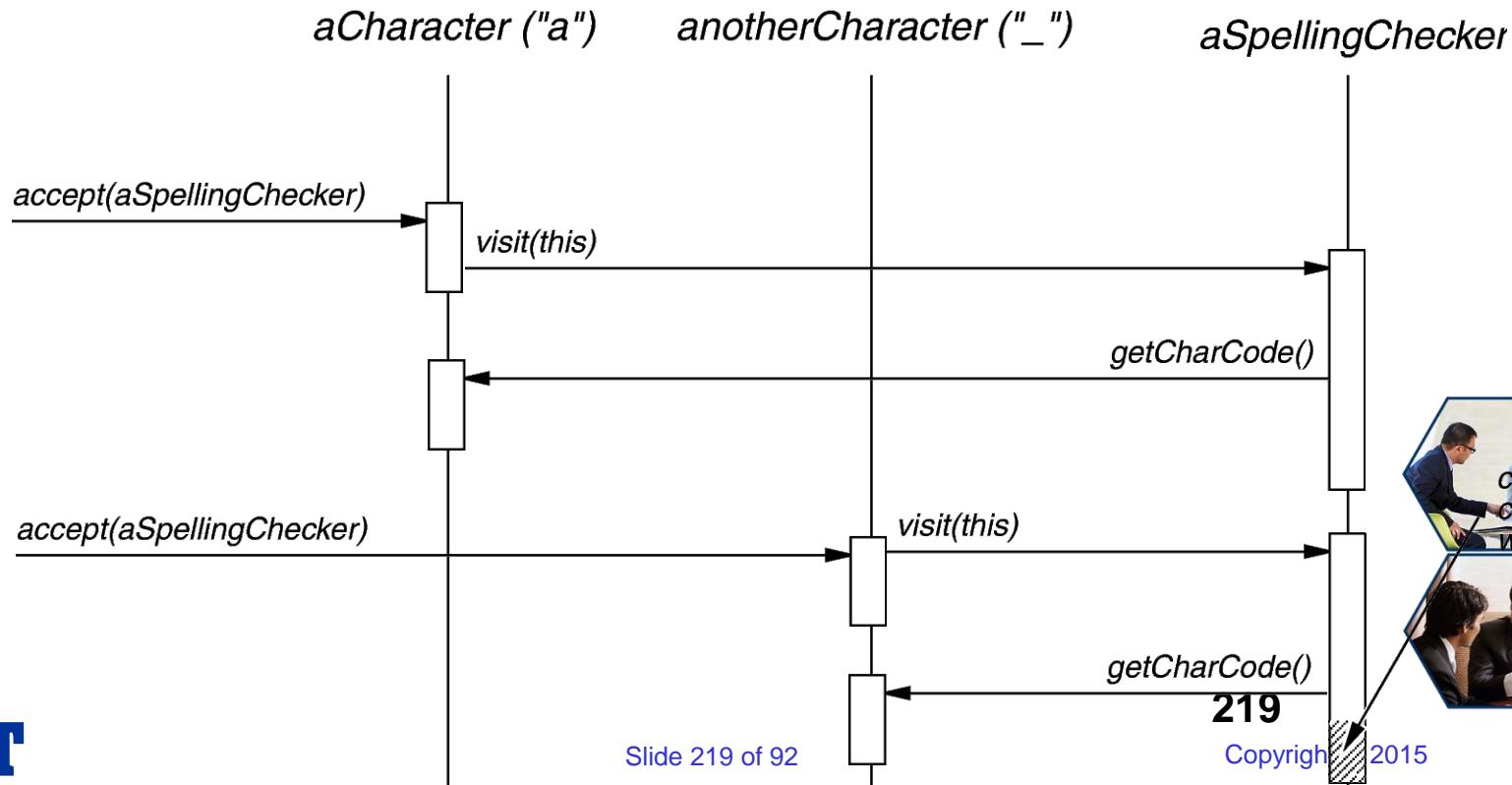




## Spelling Checking & Hyphenation (cont'd)

### Interaction Diagram

- The iterator controls the order in which accept() is called on each glyph in the composition
- accept() then “visits” the glyph to perform the desired action
- The Visitor can be subclassed to implement various desired actions





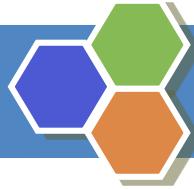
## Spelling Checking & Hyphenation (cont'd)

### HyphenationVisitor

- gets character code from each character glyph
- examines words accumulated from character glyphs
- at potential hyphenation point, inserts a...

```
class HyphenationVisitor : public Visitor {  
public:  
    void visit (Character &);  
    void visit (Rectangle &);  
    void visit (Row &);  
    // etc. for all relevant Glyph subclasses  
};
```





# Spelling Checking & Hyphenation (cont'd)

- looks like a hyphen when at end of a line
- has no appearance otherwise
- Compositor considers its presence when determining linebreaks

... "a" "l" **discretionary** "l" "o" "y" ...

aluminum alloy

or

aluminum al-

loy

221

Copyright © 2015





## Spelling Checking & Hyphenation (cont'd) VISITOR object behavioral

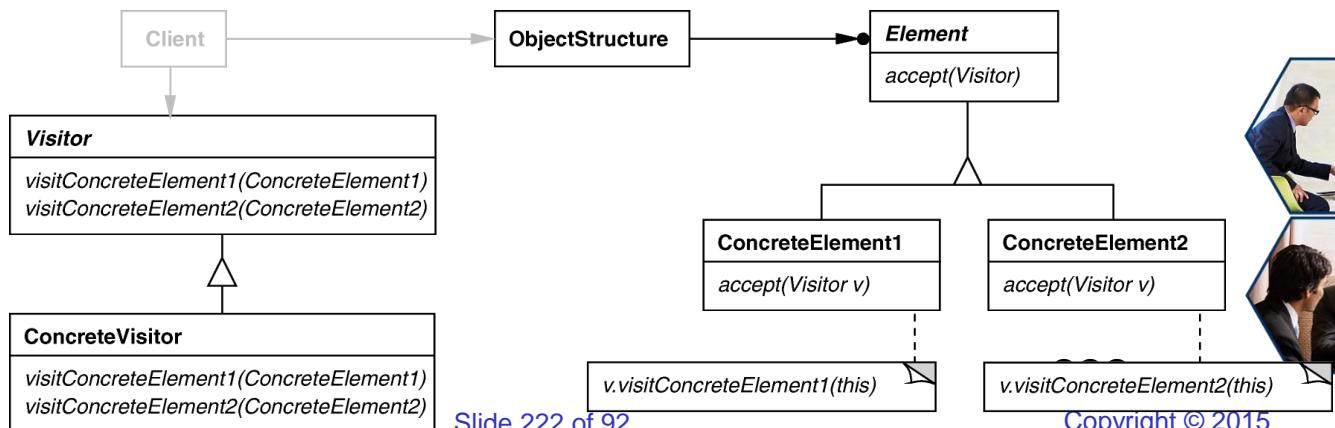
Intent

centralize operations on an object structure so that they can vary independently but still behave polymorphically

Applicability

- when classes define many unrelated operations
- class relationships of objects in the structure rarely change, but the operations on them change often
- algorithms keep state that's updated during traversal

Structure



```
SpellCheckerVisitor spell_check_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
    (*i)->accept (spell_check_visitor);
}

HyphenationVisitor hyphenation_visitor;

for (Glyph::iterator i = glyphs.begin ();
     i != glyphs.end ();
     i++) {
    (*i)->accept (hyphenation_visitor);
}
```





## Spelling Checking & Hyphenation (cont'd) VISITOR (cont'd) object behavioral

### Consequences

- + flexibility: visitor & object structure are independent
- + localized functionality
- circular dependency between Visitor & Element interfaces
- Visitor brittle to new ConcreteElement classes

### Implementation

- double dispatch
- general interface to elements of object structure

### Known Uses

- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor scene rendering





## Part III: Wrap-Up

### Observations

Patterns are applicable in all stages of the OO lifecycle

- analysis, design, & reviews
- realization & documentation
- reuse & refactoring

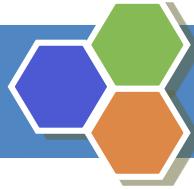
Patterns permit design at a more abstract level

- treat many class/object interactions as a unit
- often beneficial *after* initial design
- targets for class refactorings

Variation-oriented design

- consider what design aspects are variable
- identify applicable pattern(s)
- vary patterns to evaluate tradeoffs





## Part III: Wrap-Up (cont'd)

**Don't apply them blindly**

**Added indirection can yield increased complexity, cost**

**Resist branding everything a pattern**

**Articulate specific benefits**

**Demonstrate wide applicability**

**Find at least *three* existing examples from code other than your own!**

**Pattern design even harder than OO design!**

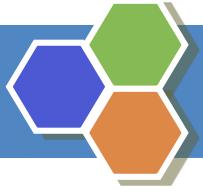




## Part III: Wrap-Up (cont'd)

- *design reuse*
- uniform design vocabulary
- understanding, restructuring, & team communication
- provides the basis for automation
- a “new” way to think about design





# Java Servlet





# What are Servlets?

- Units of Java code that run server-side.
- Run in *containers* (provide context)
- Helps with client-server communications
  - Not necessarily over HTTP
  - But usually over HTTP (we'll focus here)





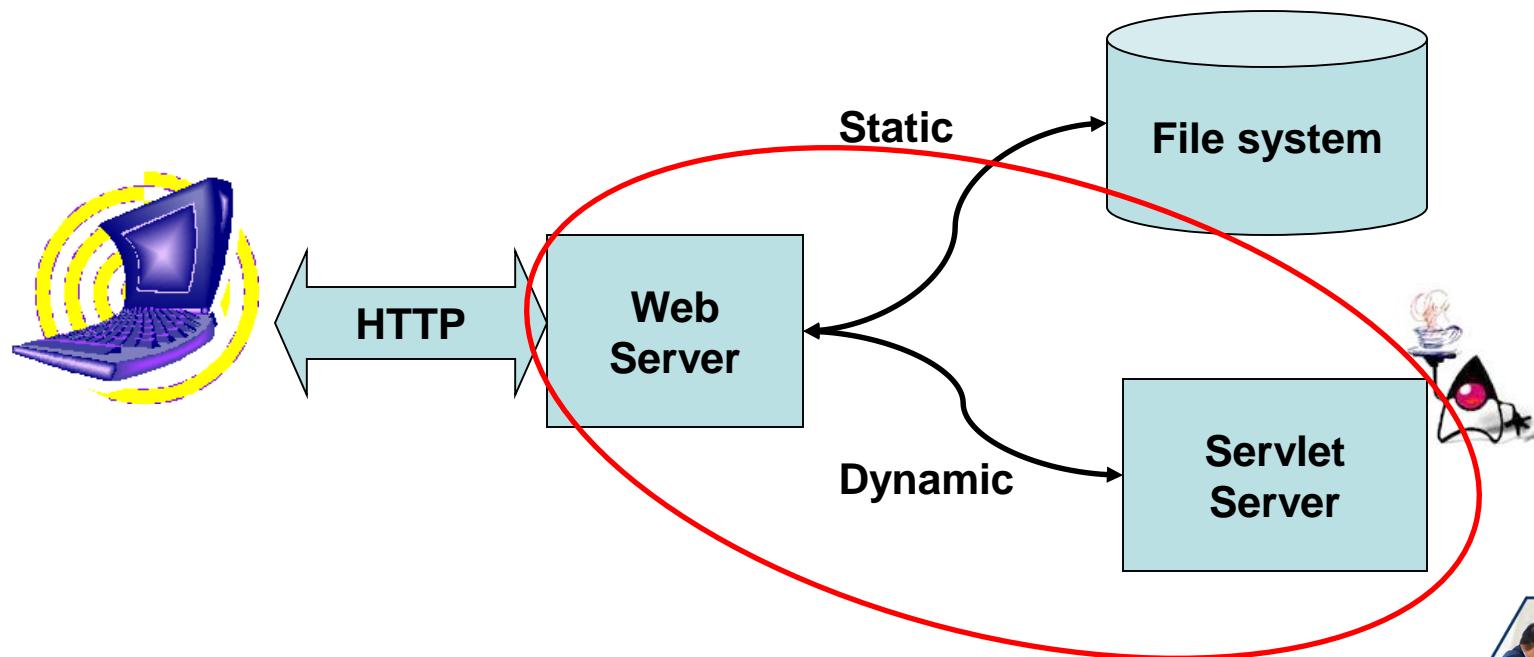
# Why are Servlets?

- Web pages with dynamic content
- Easy coordination between Servlets to make Web applications
- Containers support many features
  - Sessions, persistence, resource management (e.g., database connections), security, etc.



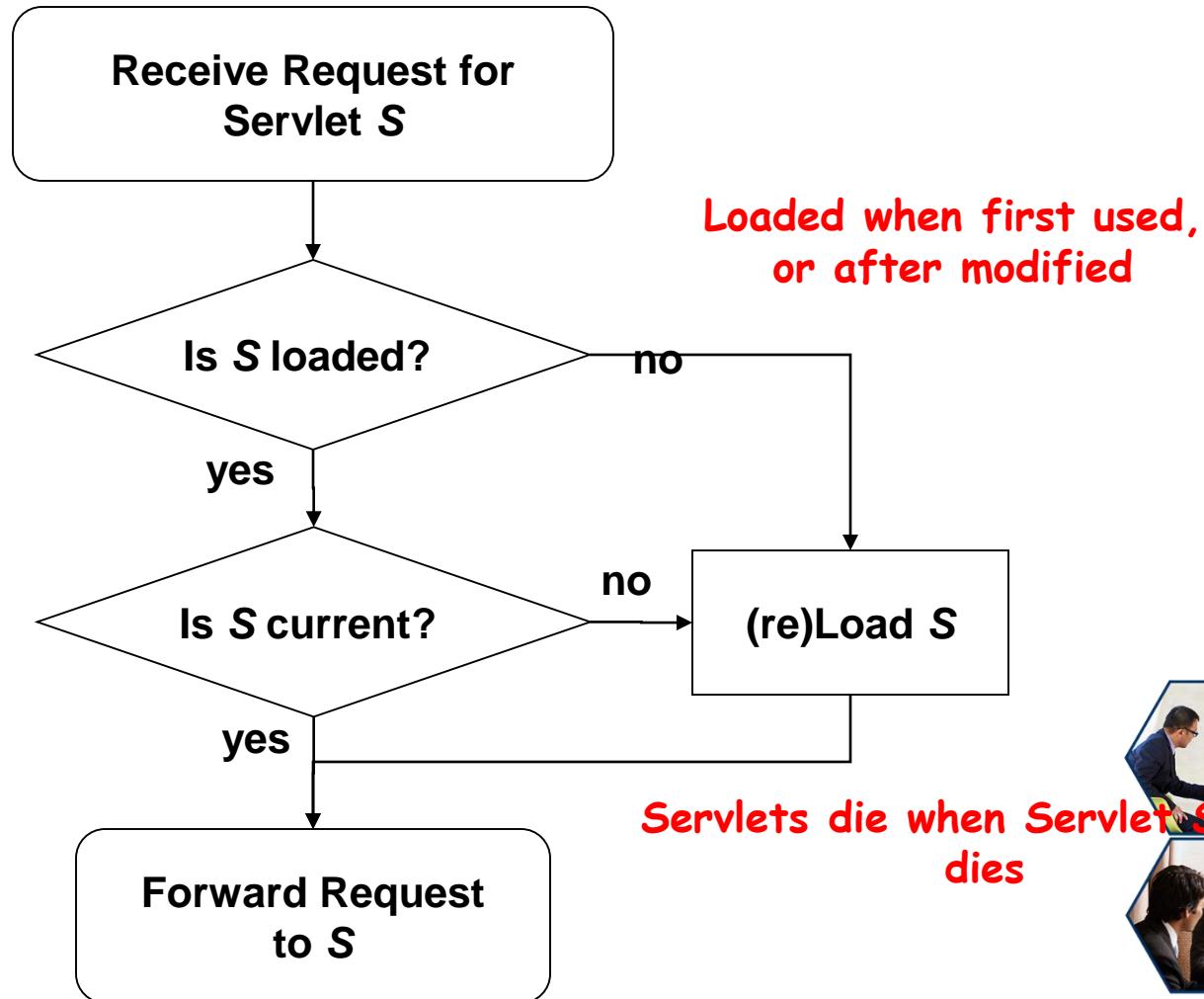


## Where are Servlets?



**Tomcat = Web Server + Servlet Server**

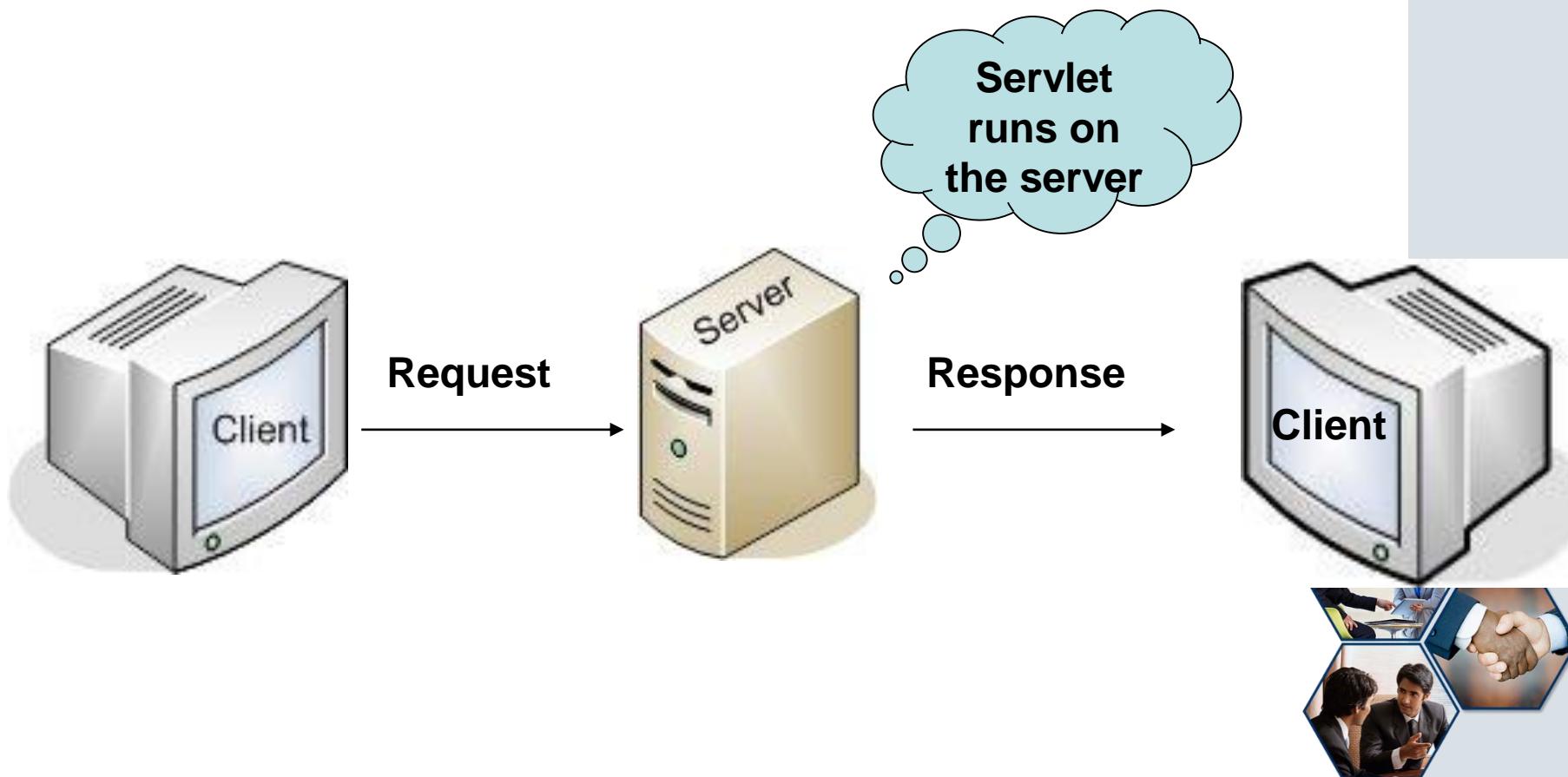


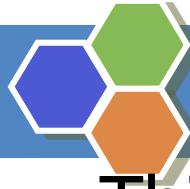




## Introduction

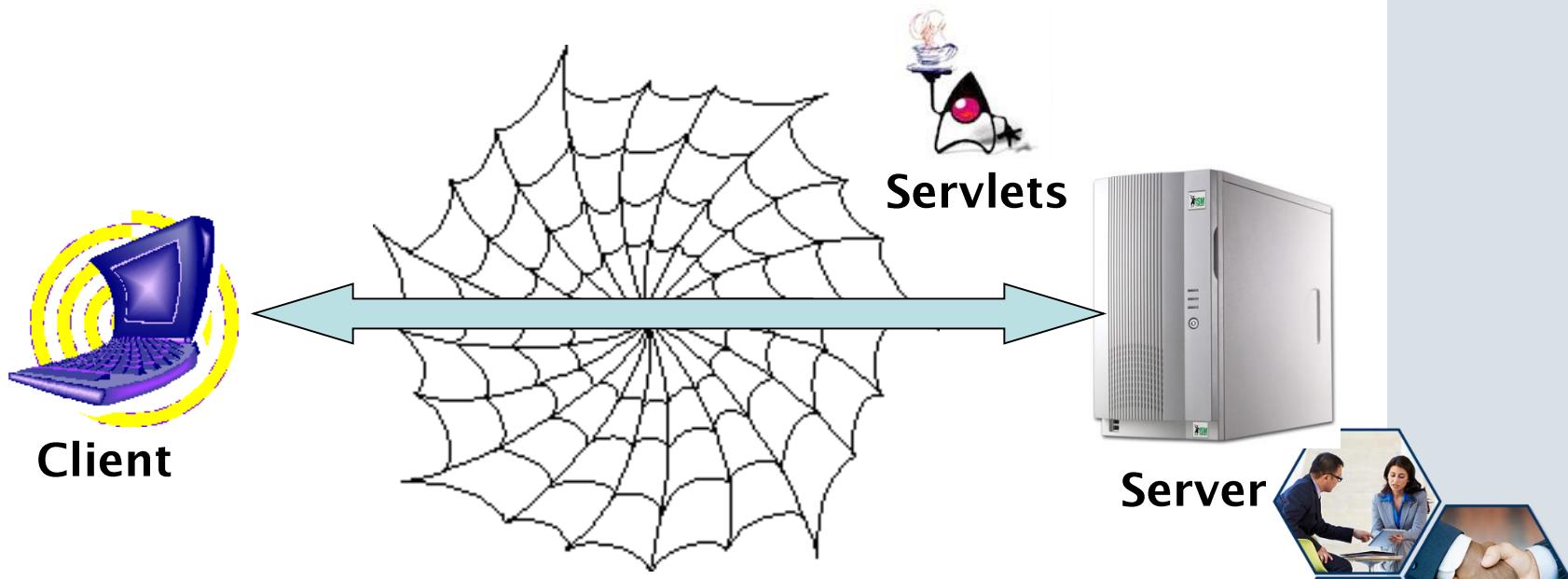
- ❑ A Servlet is a program written in Java, which runs on the server to process client requests.





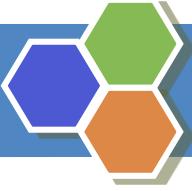
# Java on the Web: J2EE

- Thin clients (minimize download)
- Java all “server side”



**THIS IS WHAT YOU'LL BE DOING!!**





## Basics of HTTP Servlet

- HttpServlet class provides an abstract class to create an HTTP servlet.

```
protected void doGet(HttpServletRequest req,  
HttpServletResponse res);
```

**doGet() method handles the GET request made by the client**

```
protected void doPost(HttpServletRequest req,  
HttpServletResponse res);
```

**doPost() method handles the POST request made by the client**

The other methods of HttpServlet class are:

```
protected void doDelete(HttpServletRequest req,  
HttpServletResponse res);
```

**doDelete()** method is used to delete a resource from the server

```
protected void doPut(HttpServletRequest req,  
HttpServletResponse res);
```

**doPut()** method is used to place a resource on the server

```
protected void doHead(HttpServletRequest req,  
HttpServletResponse res);
```

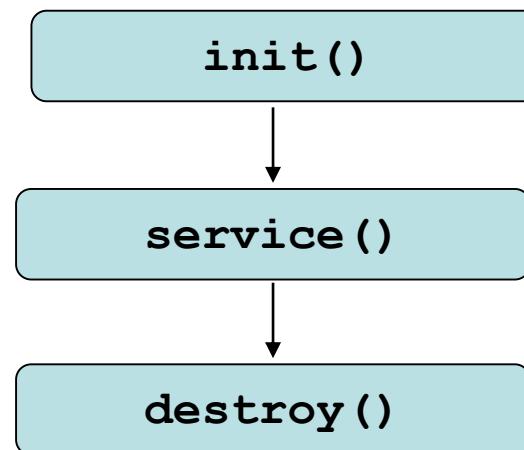
**doHead()** method handles the HEAD request made by the client





# Life Cycle of a Servlet

- An instance of a Servlet is created by the Servlet container.
- The life cycle of Servlet explains when this instance was created, for how long it lived and when it expired.
- The 3 methods of Servlet life cycle are:





- The different stages of life cycle are:

**Instantiation**

The Servlet container creates an instance of Servlet.

**Initialization**

The init() method is called by the container.

**Service**

The service() method is called by the container if it has a request for a Servlet.

**Destroy**

The destroy() method is called before destroying the instance.

**Unavailable**

The instance is destroyed and marked for garbage collection.





## Servlet Hierarchy

### javax.servlet package

- The interfaces of javax.servlet package are:

**Interface ServletConfig**



Used by Servlet container during initialization

**Interface ServletContext**



Defines methods used by the servlets to get information from its container

**Interface ServletRequest**



Requests information from the server

**Interface ServletResponse**



Responds to client request

### javax.servlet package

- The classes of javax.servlet package are:

**Class ServletInputStream**



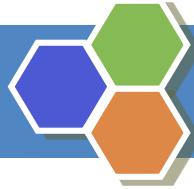
Used to read binary data from client

**Class ServletOutputStream**



Used to send binary data to client



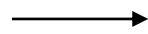


## Servlet Hierarchy

### javax.servlet.http package

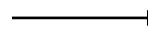
- The interfaces of javax.servlet.http package are:

Interface HttpServletRequest



Provides HTTP request information

Interface HttpServletResponse



Provides HTTP response





## Packages need to be Imported

## Method used to process the request

## Code to set text format and PrintWriter object is created to send data to the client

## Methods used to process the GET and POST request

```
package example1;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

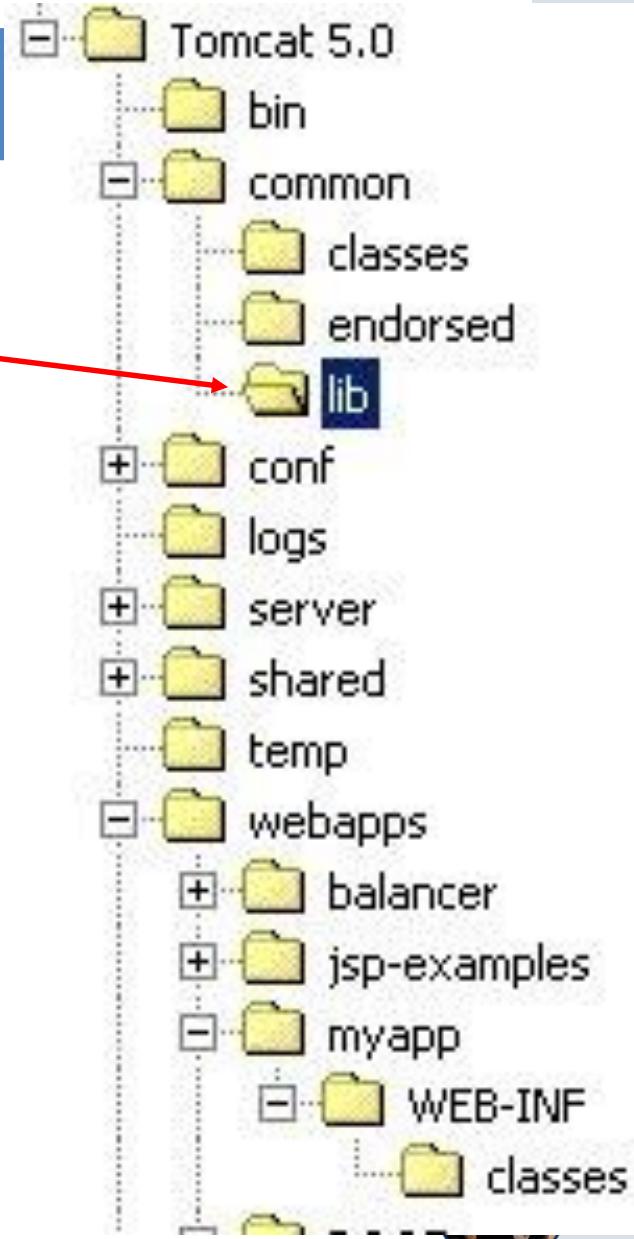
public class HelloWorld extends HttpServlet {
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Servlet HelloWorld</title>");
    out.println("</head>");
    out.println("<body>");
    out.println("<h1>Servlet HelloWorld at" +request.getContextPath()+"</h1>");
    out.println("</body>");
    out.println("</html>");
    out.close();
}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    processRequest(request, response);
}
}
```



# javac -classpath

\$LIB/servlet-api.jar

Hellox.java



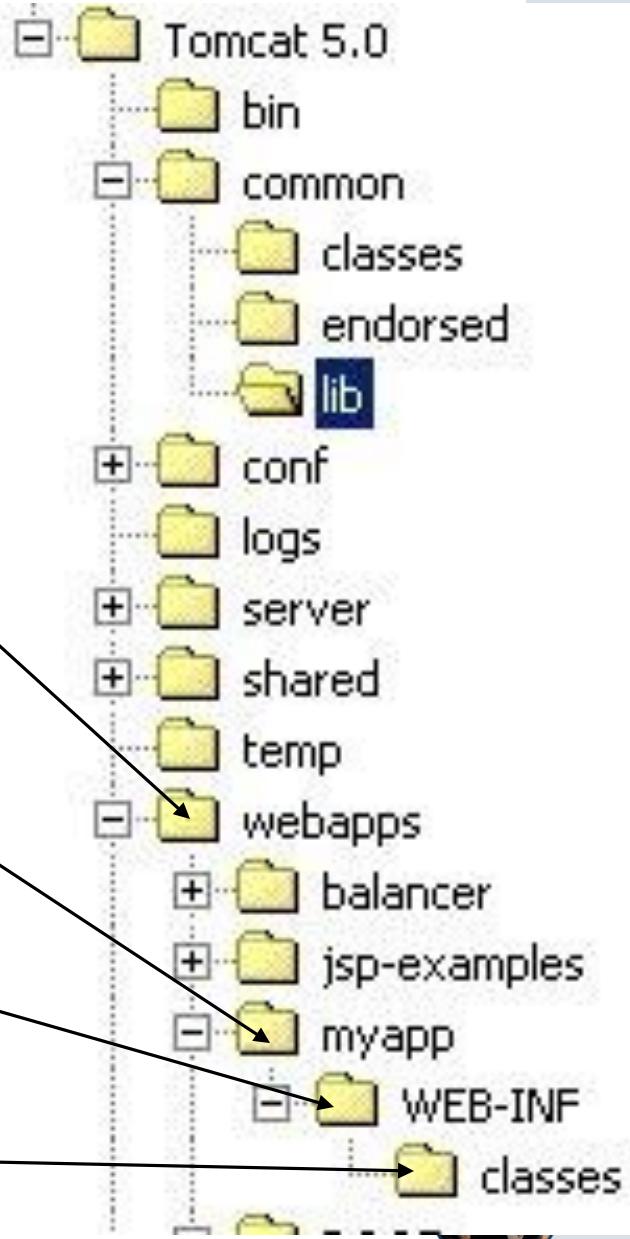


Create your  
web applications  
here

Create a directory  
*D* for your  
web application

Create "WEB-INF"  
under *D*

Create "classes"  
under "WEB-INF"





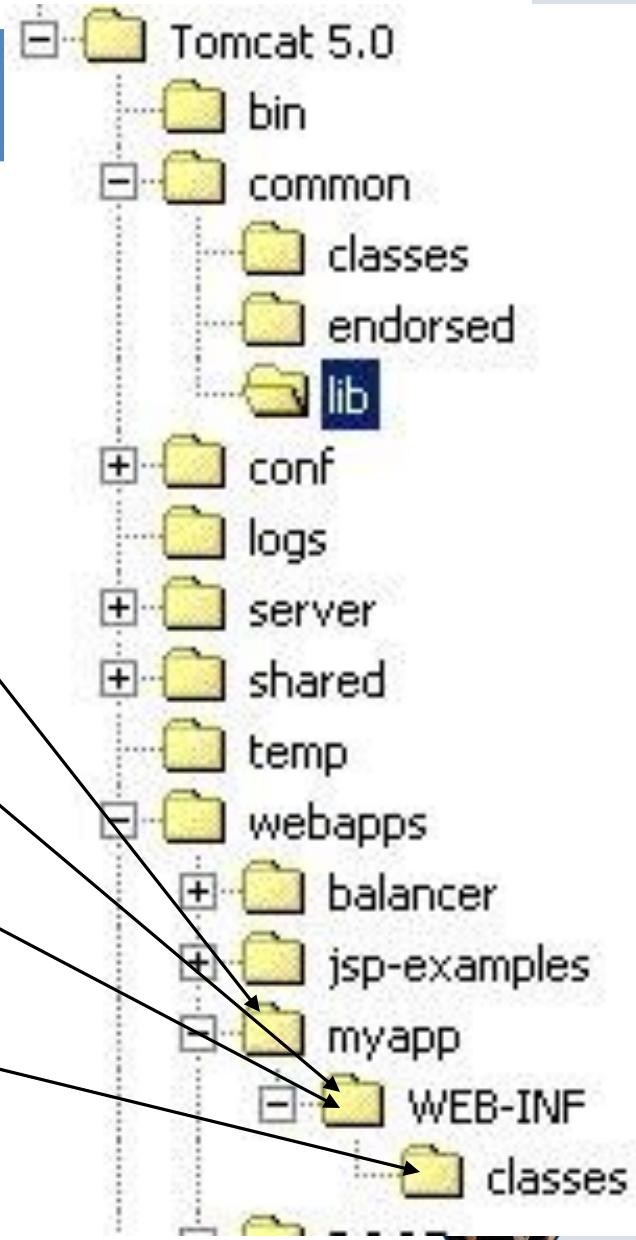
(cont.)

Static content in D

Dynamic content  
in WEB-INF

web.xml in WEB-INF

Servlets in classes



```
<?xml version="1.0" encoding="ISO-8859-1"?>
```



```
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
         xsi:schemaLocation=  
         "http://java.sun.com/xml/ns/j2ee  
          http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"  
         version="2.4">
```

```
    <description>Examples</description>
```

```
    <display-name>Examples</display-name>
```

Declares servlet

```
    <servlet>
```

abbreviation

```
        <servlet-name>Hellox</servlet-name>
```

```
        <servlet-class>pack.Hellox</servlet-class>
```

```
    </servlet>
```

Maps servlet to URL (rooted at D)

```
    <servlet-mapping>
```

```
        <servlet-name>Hellox</servlet-name>
```

```
        <url-pattern>/Hellox</url-pattern>
```

```
    </servlet-mapping> </web-app>
```





```
Package pack;  
public class Helloy extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException, ServletException  
{  
    response.setContentType("text/html");  
    PrintWriter out = response.getWriter();  
    out.println("<html>");  
    out.println("<body>");  
    out.println("<head>");  
    out.println("<title>Hello, Tell me your name!</title>");  
    out.println("</head>");  
    out.println("<body>");  
    out.println("<h1>Hello, Tell me your name!</h1> <br>");  
    out.print("<form action=\"");  
    out.println("NamedHello\" method=POST>");  
    out.println("<input type=text length=20 name=yourname><br>");  
    out.println("<input type=submit></form>");  
  
    out.println("</body>");  
    out.println("</html>");  
}}
```



```

Package pack;

public class NamedHello extends HttpServlet {
    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String name = request.getParameter("yourname");
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Hello, Tell me your name again!</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h2>Hello, " + name + "</h2> <br>");
        out.print("<form action=\"\">");
        out.println("NamedHello\" method=POST>");
        out.println("<input type=text length=20 name=yourname><br>");
        out.println("<input type=submit></form>");
        out.println("</body>");
        out.println("</html>");
    }
}

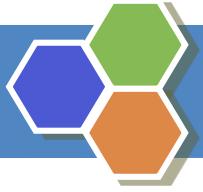
```





```
public class NamedSessionHello1 extends HttpServlet {  
    public void doGet(HttpServletRequest request,  
                      HttpServletResponse response)  
        throws IOException, ServletException {  
  
        response.setContentType("text/html"); PrintWriter out = response.getWriter();  
        HttpSession hs = request.getSession(true);  
        String sn = (String) hs.getAttribute("yourname");  
        out.println("<html>"); out.println("<body>"); out.println("<head>");  
        out.println("<title>Hello, Tell me your name again!</title>");  
        out.println("</head>"); out.println("<body>");  
        if(sn != null && ! sn.equals("")) {  
            out.println("<h1><blink> OH, NamedSessionHello1" +  
                       "already know your name: " + sn + "</blink></h1>");  
        } else {  
            String sn2 = request.getParameter("yourname");  
            if (sn2 == null || sn2.equals("")) {  
                out.println("<h2>Hello,noname " + "</h2> <br>");  
            } else {  
                out.println("<h2>Hello, " + sn2 + "</h2> <br>");  
                hs.setAttribute("yourname", sn2);  
            }  
            out.print("<form action=\"\""); out.println("NamedSessionHello2\" method=GET>");  
        }  
    }  
}
```





# Session Tracking

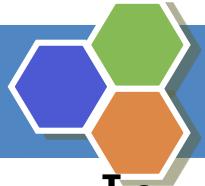




# Session Tracking

- Session tracking is the capability of the server to maintain the current state of a single client's sequential requests
- HTTP is stateless -> how do you keep track of client request sequences?
- Following are different ways to manage session tracking:
  - URL Rewriting
  - Hidden Form Fields
  - Persistent Cookies
  - Session Tracking API

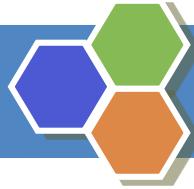




# URL Rewriting

- In <A HREF=""> rewriting URL and passing var=value as query string
- Typically, only a unique session ID is passed
- By this way, we can pass only strings.
- Can send only limited data
- `http://myweb.com/otherpage.jsp?name=harish`
- In other page, `request.getParameter("name")` is used to access data

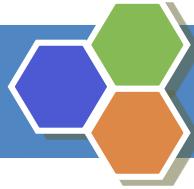




# URL Rewriting

- Advantage
  - Easy to Implement
  - Supported by all popular browsers
  - User need not log in (server can generate unique IDs)
  - Works for all dynamically created documents
- Disadvantage
  - Only small amount of information can be send
  - URL rewriting does not work for static documents
  - Browser shutdown breaks the process

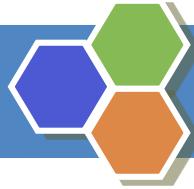




# Hidden Form Field

- One of the simplest methods for session tracking
- Use standard form fields, which are not displayed to the client
- Parameters passed back to server, when form is submitted
- `<INPUT TYPE="hidden NAME="user" VALUE="Harish">`
- No difference between a hidden and visible field (at server end)

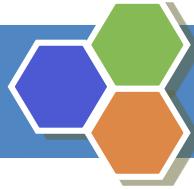




# Hidden Form Field

- Advantage
  - Easy to Implement
  - Supported by all popular browsers
  - User need not log in (server can generate unique IDs)
  - Works for all dynamically created documents
- Disadvantage
  - Only small amount of information can be send
  - URL rewriting does not work for static documents
  - Browser shutdown breaks the process

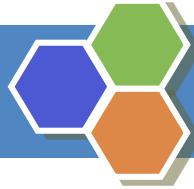




# Cookies

- Cookie is a simple piece of information stored on the client, on behalf of the server
- When a browser receives a cookie, it saves the cookie on the local hard drive.
- Expiry of the cookie can be set. Otherwise it will be deleted at browser shutdown
- Each subsequent visit to the same site, browser sends cookie back to server with each request
- Cookies not part of original HTTP specification -> introduced in Netscape and have become a 'de-facto'





# Cookies

- Cookies type
  - Non-persistence
  - Persistence
- Both types are stored at client location.
- Cookies stores information in key/value pair format.
- Browser limitation for cookies
  - 20 cookies for each Web server
  - 300 cookies total
  - Cookie size upto 4 KB each





# Cookies

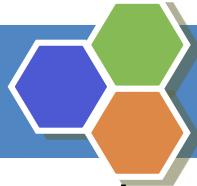
- Servlet API provides javax.servlet.http.Cookie for using cookies
  - public Cookie (String name, String value)
  - public void HttpServletResponse.addCookie(Cookie cookie)
- Example:

```
Cookie cookie = new Cookie("userid", "HARISH");
cookie.setMaxAge(60*60*24); // 1 day expiry
response.addCookie(cookie);
```

- Servlet retrieves cookies by calling the getCookies() method

```
Cookie[] cookies = request.getCookies();
if (cookies != null) {
    for (int i=0; i< cookies.length; i++) {
        String name = cookies[i].getName();
        String value = cookies[i].getValue();
    }
}
```





# Session Tracking API

- The Servlet API has built-in support for session tracking
- Session tracking should be supported by any web server that supports servlets
- To create a new session
  - `public HttpSession HttpServletRequest.getSession(boolean create);`

```
HttpSession session = req.getSession(true);
```

- Method returns the current session associated with the user making the request. If the user has no current valid session, this method creates one if 'create' is set as true, or returns null if 'create' is false





# Session Tracking API

- HttpSession object has following methods we are concerned with:

- public void setAttribute(String name, Object value)

Binds a name/value pair to store in the current session      If the name already exists, then it is replaced

- public Object getAttribute(String name)

Used to get an object stored in the session

- public void removeAttribute(String name)

Allows the removal of an object from the session

- public void invalidate()

Can manually invalidate their session and all information





# Session Tracking API

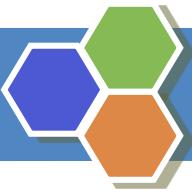
- Advantage

- Very efficient and simple way to implement session tracking
- Session information can persist (server specific) even with server shutdown/restart
- Users need not log on (server generates unique IDs)
- Works for all documents/pages, including static HTML
- Standardized and included in J2EE
- Sessions can be set to “time-out” at user configurable times (`setMaxInactiveInterval(second)`)
- Objects stored rather than just strings

- Disadvantage

- Restricted to one browser and tracking lost if browser shuts down or crashes
- Slightly more difficult if clients have cookies disabled





# Java Server Faces





# Agenda

- JavaServer Faces – Objectives, Overview and Intended Users
- Components Of JSF
- Features
- JSF Life Cycle
- Tools supporting JSF
- Sample Application
- JavaServer Faces or Struts?
- Comparison Summary
- JSF in Multitiered architecture
- Integrating JavaServer Faces With Spring
- Useful Links





# Introduction

JavaServer Faces is a framework that simplifies development of sophisticated web application user interfaces, primarily by defining a user interface component model tied to a well defined request processing life cycle.





# Objectives

- To provide event-driven component based technology for developing web applications.
- To separate the user interface from the model objects that encapsulate the data and application logic.
- To provide the server side validation and data conversion.
- To retain the state of the components.





# Overview

- JSF is a framework for building user interfaces for web applications.
- It brings component-based model to web application development.
- It focuses only on user interface details and is not concerned about how the rest of the application is implemented.
- It includes:
  - A set of APIs for representing UI components and managing their state, handling events and input validation, page navigation, and supporting internationalization and accessibility.
  - A Java Server Pages custom tag library for expressing a JavaServer Faces interface within a JSP page.



- It supports different renderers and not restricted to HTML only like other frameworks.



# Intended Users of JSF

- Because of the extensibility and ease-of-use that JSF technology provides, a wide range of developers and web-page designers can take advantage of the power of JSF technology. These users include:
- Page Authors: who creates the user interfaces.
- Application developers: who write the application code, including the data-access, event-handling, and business logic.
- Component writers: who construct reusable UI components, and custom components.
- Tools vendors: who build tools leveraging JavaServer Faces technology to make building a user interface with JavaServer Faces technology even easier.





# Components of JSF

- What is a component?

In JSF, a component is a group of classes that together provide a reusable piece of web-based user interface code.

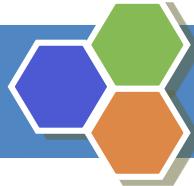
- A component is made up of three classes that work together. They are:

Render class.

UIComponent subclass.

JSP custom action class.





# UI Component Model

JSF UI components are configurable, reusable elements that compose the user interfaces of JSF applications. A component can be simple, such as a button, or compound, such as a table, which can be composed of multiple components.

JSF technology provides a rich, flexible component architecture that includes the following:

- 1. A set of `UIComponent` classes for specifying the state and behaviour of UI components.**
- 2. A rendering model that defines how to render the components in various ways.**
- 3. An event and listener model that defines how to handle component events.**
- 4. A conversion model that defines how to register data converters on to a component.**
- 5. A validation model that defines how to register validators on to a component.**





# UI Component Classes

- JSF technology provides set of UI component classes and associated behavioral interfaces that specify all the UI component functionality, such as holding component state, maintaining reference to objects, and driving event handling and rendering for a set of standard components.
- The component classes are completely extensible, allowing component writers to create their own custom components.
- All JSF UI Component classes extend UIComponentBase, which defines the default state and behavior of a UI component.
- In addition to extending the UIComponentBase, the component class also implement one or more behavioral interfaces, each of which defines certain behavior for a set of components whose classes implement the interface.





# Few Behavioral Interfaces

- ActionSource: Indicates that the component can fire an action event.
- EditableValueHolder: Extends ValueHolder and specifies additional features for editable components, such as validation and emitting value-change events.
- NamingContainer: Mandates that each component rooted at this component have a unique ID.
- StateHolder: Denotes that a component has state that must be saved between requests.
- ValueHolder: Indicates that the component maintains a local value as well as the option of accessing data in the model tier





# Render Classes

- For every UI Component that a render kit supports, the render kit defines a set of renderer classes.
- Each renderer class defines a different way to render the particular component to the output defined by the render kit.
- For example a UISelectOne component has three different renderers. One of them renders the component as a set of radio buttons. Another renders the component as a combo box. The third one renders the component as a list box.
- Each JSP custom tag defined in the standard HTML render kit is composed of the component functionality(defined in UIComponent class) and the rendering attributes(defined by the renderer class)





# Example – How rendering is done?

- The two tags `commandButton` and `commandLink` that represent a `UICommand` component rendered in two different ways as shown below.

Tag	Rendered As
<code>commandButton</code>	<input type="button" value="Login"/>
<code>commandLink</code>	<a href="#">hyperlink</a>

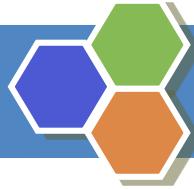




# Few Important UI Components

- Few important UI components are:
- **UIForm:** Encapsulates a group of controls that submit data to the application. This component is analogous to the form tag in HTML.
- **UIInput:** Takes data input from a user. This class is a subclass of **UIOutput**.
- **UICommand:** Represents a control that fires actions when activated.
- **UIOutput:** Displays data output on a page.
- **UIMessage:** Displays a localized message.





# JavaServer Faces – Features

- Page navigation specification
- Standard user interface components like input fields, buttons, and links
- Type conversion
- User input validation
- Easy error handling
- Java bean management
- Event handling
- Internationalization support



- **All multi-page web applications need some mechanism to manage page navigation for their users.**
  
- **JSF offers page navigation through page navigation rules in the Application Configuration file(faces-config.xml)**
  
- **Page Navigation can be**
  - Simple Page Navigation
  - Conditional Page Navigation

Simple page navigation

```
<navigation-rule>
    <from-tree-
        id>/page1.jsp</from-tree-id>
    <navigation-case>
        <to-tree-id>/page2.jsp</to-
            tree-id>
    </navigation-case>
</navigation-rule>
```

Conditional Page Navigation

```
<navigation-rule>
    <from-tree-
        id>/login.jsp</from-tree-id>
    <navigation-case>
        <from-
            outcome>success</from-outcome>
        <to-tree-id>/welcome.jsp</to-
            tree-id> </navigation-case>
    </navigation-case >
</navigation-rule>
```





# How Navigation is done

- When a button or hyperlink is clicked the component associated with it generates an action event.
- This event is handled by the default ActionListener instance, which calls the action method referenced by the component that triggered the event.
- This action method is located in backing bean and is provided by application developer.
- This action method returns a logical outcome String which describes the result of the processing.
- The listener passes the outcome and a reference to the action method that produced the outcome to the default NavigationHandler.
- The NavigationHandler selects the next page to be displayed by matching the outcome or the action method reference against the navigation rules in the application configuration resource file.





# Standard UI components

- To use the HTML and Core custom tag libraries in a JSP page, you must include the taglib directives in the page.
- The components are reusable
- Taglib directives

```
<%@ taglib uri="http://java.sun.com/jsf/html/"  
prefix="h" %>  
  
<%@ taglib uri="http://java.sun.com/jsf/core/"  
prefix="f" %>
```
- Components
  - ```
<h:commandButton id="submit" action="next"  
value="Submit" />
```
  - ```
<h:inputText id="userName"  
value="#{GetNameBean.userName}"  
required="true" />
```
  - ```
<h:outputText  
value="#{Message.greeting_text}" />
```





# User input validation

- If validation or type conversion is unsuccessful, a component specific FacesMessage instance is added to FacesContext. The message contains summary, detail and severity information
- Validation can also be delegated to a managed bean by adding a method binding in the validator attribute of an input tag.
- This mechanism is particularly useful for accomplishing form validation, where combinations of inputted values need to be evaluated to determine whether validation should succeed.

- Standard/Built-in validation components

```
<h:inputText id="age" value="#{UserRegistration.user.age}">  
    <f:validateLongRange maximum="150" minimum="0"/>  
</h:inputText>
```

- Custom Component

```
public class CodeValidator implements Validator{  
    public void validate(FacesContext context, UIComponent  
        component, Object value) throws  
            ValidatorException
```

```
{ } }
```

```
<validator>  
    <validator-id>jcoe.codeValidator</validator-id>  
    <validator-  
        class>com.jcoe.validation.CodeValidator</validator-class>  
</validator>
```

```
<h:inputText id="zipCode"  
    value="#{UserRegistration.user.zipCode}">  
    <f:validator validatorId="jcoe.codeValidator"/>  
</h:inputText>
```





# Type Conversion

- A JavaServer Faces application can optionally associate a component with server-side object data. This object is a JavaBeans component, such as a backing bean. An application gets and sets the object data for a component by calling the appropriate object properties for that component.
- When a component is bound to an object, the application has two views of the component's data:
  - The model view, in which data is represented as data types, such as int or long.
  - The presentation view, in which data is represented in a manner that can be read or modified by the user. For example, a `java.util.Date` might be represented as a text string in the format `mm/dd/yyyy`, or as a set of three text strings.





# How Conversion is done?

- The JSF technology automatically converts the component data between the model view and the presentation view.
- You can create your own custom converter.
- To create a custom converter converter in your application, three things must be done:
  1. The application developer must implement the Converter class.
  2. The application architect must register the Converter with the application.
  3. The page author must refer to the Converter from the tag of the component whose data must be converted

- The converter attribute on the component tag

```
<h:inputText value="#{LoginBean.Age}"  
converter="javax.faces.convert.IntegerConverter"  
/>>
```

- The method to convert the model value of the component

```
Integer age = 0;  
public Integer getAge()  
{  
    return age;  
}  
public void setAge(Integer age)  
{  
    this.age = age;  
}
```

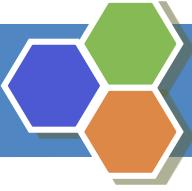




# Error handling

- The JSF core component set provide an `HtmlMessages` component, which simply outputs the summary message from all the `FacesMessage` instances added to the `FacesContext` during validation
- Depending on the severity and type of error, the response to it may vary, but at least a sensible error message usually should be shown to the end user.
- The JSF framework has several points within its page request processing lifecycle that can raise errors and display consistent error messages.





# Java bean management

- The managed-bean element in the faces-config.xml application configuration file manages the java beans.
- Each managed-bean element registers a JavaBean that JSF will instantiate and store in the specified scope.

## Faces-config.xml

```
<!ELEMENT managed-bean  
        (description*,  
         display-name*,  
         icon*,  
         managed-bean name,  
         managed-bean-class,  
         managed-bean-scope,  
         (managed-property* | map-  
             entries |  
             list-entries  
)>
```

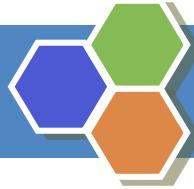




# Event handling

- JSF applications are event-driven. Handling events in JSF is surprisingly easy. Here are the steps:
  - Write an event listener.
  - Deploy the event listener in the WEB-INF/classes or WEB-INF/lib directory under the application directory.
  - In the tag representing the component whose event is to be captured, use an action\_listener or a valuechange\_listener tag defined in the Core custom tag library.
- ## • Event objects
- Must extend javax.faces.event.FacesEvent
  - FacesEvent is a subclass of the java.util.EventObject class
  - It adds the getComponent method, which returns the UIComponent component that fired the event.
  - The FacesEvent class has two subclasses: ActionEvent and ValueChangeEvent.
  - The ActionEvent class represents the activation of the UI component, such as a UICommand component.
  - The ValueChangeEvent class represents a notification that the local value of a UIInput component has been changed.



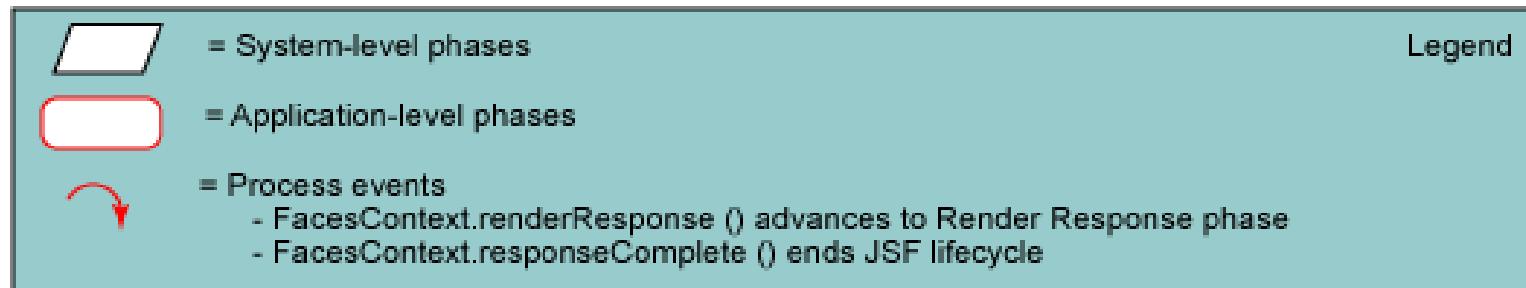
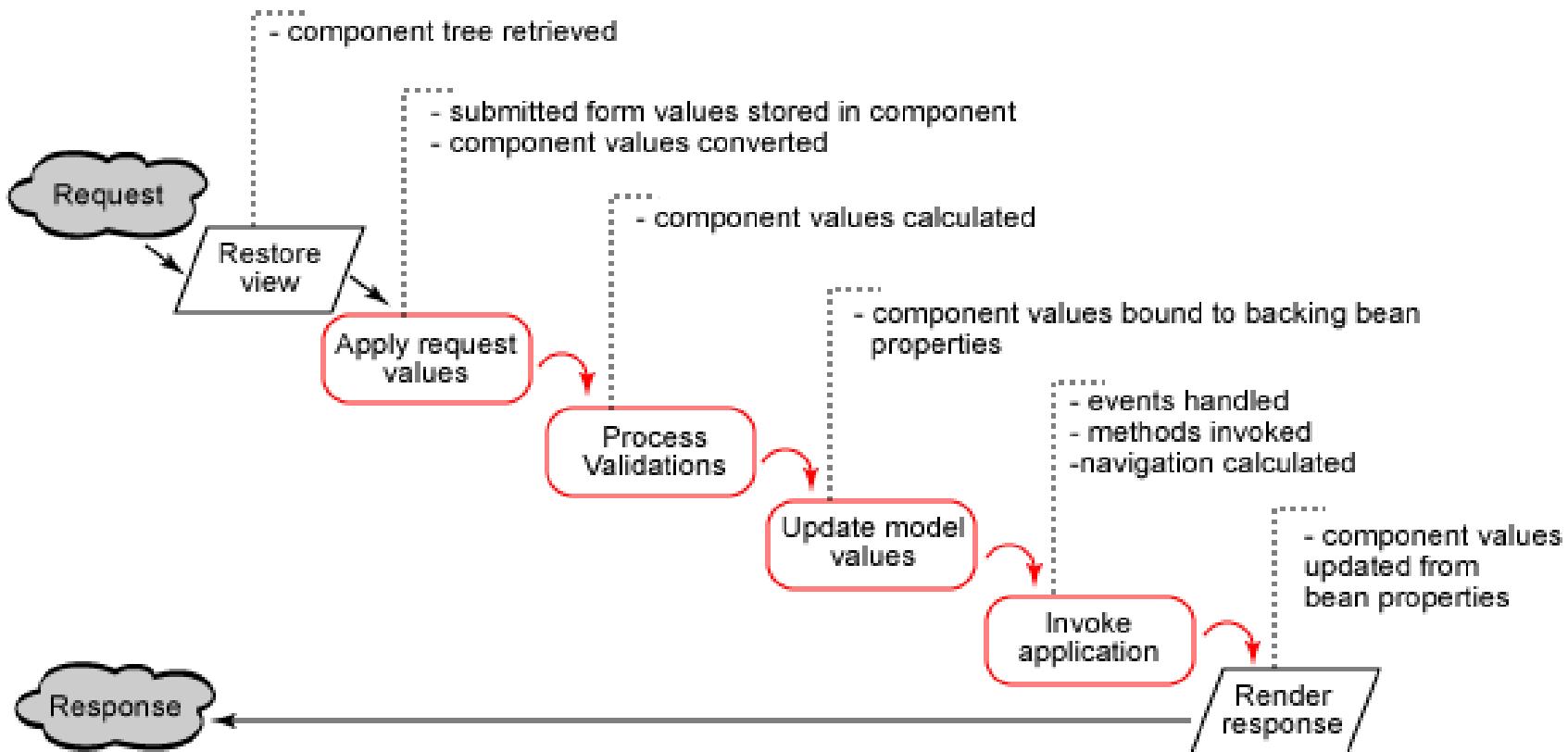


## Event handling (Cont.)

- **Event listeners**
- **javax.faces.event.FacesListener interface**
- **This interface extends the java.util.EventListener interface**
- **The FacesListener interface has two subinterfaces: ActionListener and ValueChangeListener**



# JSF Life Cycle





# MVC Architecture in JSF

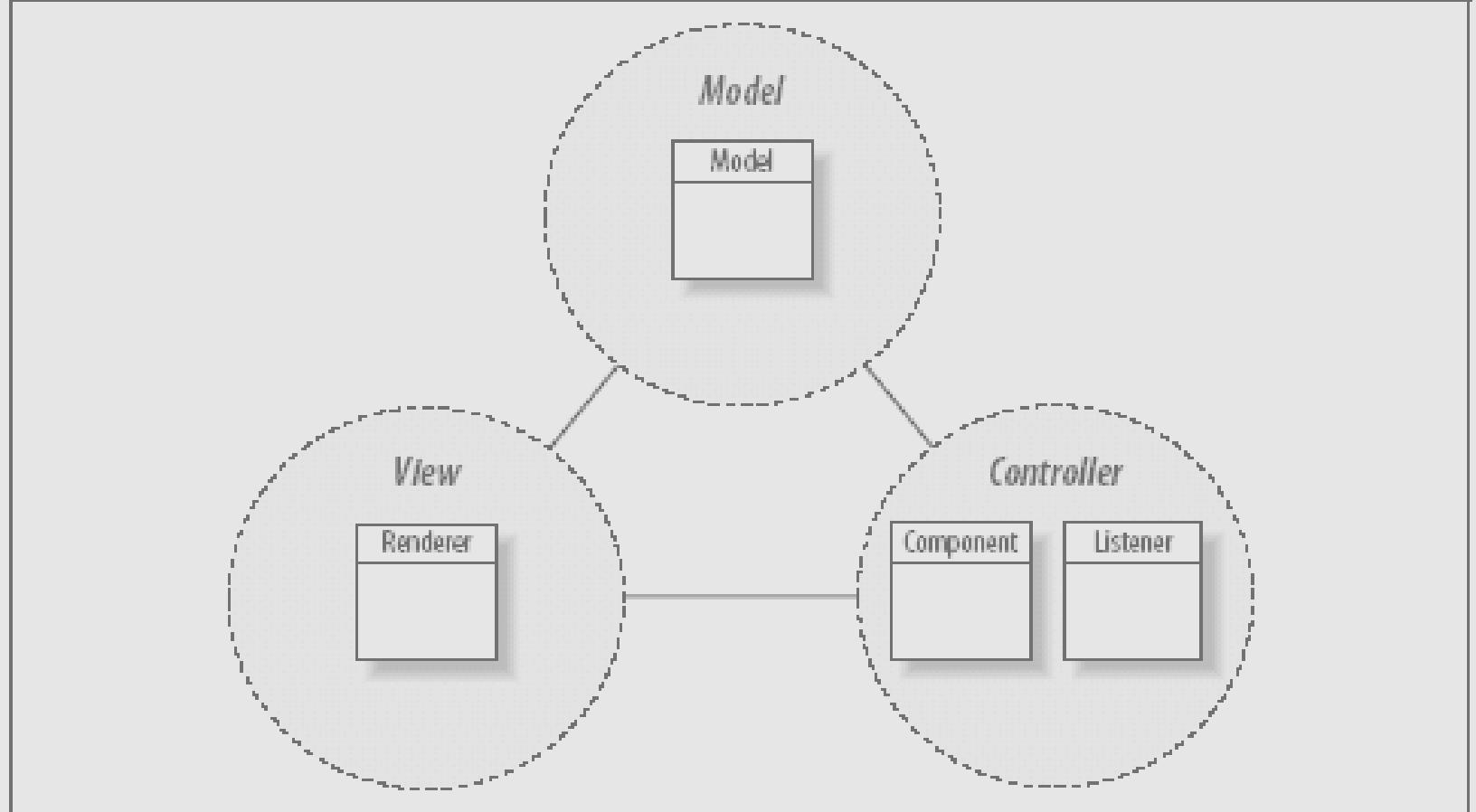


Figure 1-2. JThe JSF MVC design



# Tools supporting JSF

- Many java IDEs support Java Server Faces in their tools with some common and unique features.
  - **Borland JBuilder Enterprise Edition 2005**
  - **Eclipse with Exadel Studio Pro IDE**
  - **Eclipse with Nitrox JSF IDE**
  - **IBM Rational with JSF Support**
  - **Macromedia Dreamweaver with JSTL & JSF extensions**
  - **MyEclipse with JSF Designer**
  - **Oracle's JDeveloper IDE**
  - **Sun's Java Studio Creator**





# Sample Application

- Requirements:
- Index page - > inputname.jsp
- Forward - > greetings.jsp
- Name - > The number of characters should be more than 2 and less than 11 and the name should not be null.
- Design:
- Inputname.jsp
- Greetings.jsp
- Messages.properties





Navigator Package Explorer X

jsfKS

- JavaSource
  - demo
    - GetNameBean.java
      - GetNameBean
      - user Name
      - getUserName()
      - setUserName(String)
  - demo.bundle
    - Messages.properties
- JRE System Library [DEV\_ENV\_JDK]
- commons-beanutils.jar
- commons-collections.jar
- commons-digester.jar
- commons-logging.jar
- jsf-api.jar
- jsf-impl.jar
- jstl.jar
- standard.jar
- WebContent
  - WEB-INF
    - lib
      - faces-config.xml
      - web.xml
  - pages
    - greeting.jsp
    - inputname.jsp
    - index.jsp
- ant
  - build.xml
- build
  - jsfKS.war
- deploy
  - jsfKS.war
- LICENSE-APACHE.txt

build.xml inputname.jsp faces-con... greeting.jsp web.xml GetNameBe... Messages.p...

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
  "-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
  "http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>

  <navigation-rule>
    <from-view-id>/pages/inputname.jsp</from-view-id>
    <navigation-case>
      <to-view-id>/pages/greeting.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <description>
      Input Value Holder
    </description>
    <managed-bean-name>GetNameBean</managed-bean-name>
    <managed-bean-class>demo.GetNameBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>userName</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
  </managed-bean>
</faces-config>
```

Problems Javadoc Declaration Search Console Debug

C:\SDE\jdk141\_05\bin\javaw.exe (Dec 1, 2005 3:46:11 PM)



# Inputname.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="demo.bundle.Messages"
  var="Message"/>

<HTML>
  <HEAD> <title>Input Name Page</title> </HEAD>
  <body bgcolor="white">
<f:view>
<h1><h:outputText
  value="#{Message.inputname_header}"/></h1>
<h:messages style="color: red"/>
<h:form id="helloForm" >
  <h:outputText value="#{Message.prompt}"/>
  <h:inputText id="userName"
    value="#{GetNameBean.userName}" required="true">
    <f:validateLength minimum="3" maximum="10"/>
  </h:inputText>
  <h:commandButton id="submit" action="sayhello"
    value="Submit" />
</h:form>
</f:view>
</HTML>
```





# Greeting.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<f:loadBundle basename="demo.bundle.Messages" var="Message"/>
<HTML>
    <HEAD> <title>Greeting Page</title> </HEAD>
    <body bgcolor="white">
        <f:view>
<h3>
    <h:outputText value="#{Message.greeting_text}" />,
    <h:outputText value="#{GetNameBean.userName}" />!
</h3>
        </f:view>
    </body>
</HTML>
```

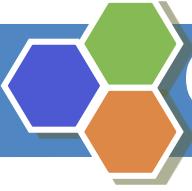
## Messages.properties

**inputname\_header=Hello Application**

**prompt=Name:**

**greeting text>Hello**





# GetNameBean.java

```
package demo;  
public class GetNameBean {  
    String userName;  
    /**  
     * @return User Name  
     */  
    public String getUserName() {  
        return userName;  
    }  
    /**  
     * @param User Name  
     */  
    public void setUserName(String name) {  
        userName = name;  
    }  
}
```



# Snapshots

Import Name Page Microsoft Internet Explorer provided by Sagittarius Technology Solutions

File Edit View Favorites Tools Help

Back Search

Address http://localhost:8080/jsfks/faces/pages/inputname.jsp

## Hello Application

Name:

Done Local intranet

start 5 Microsoft ... Java - greetin... 3 Internet E... 5 Windows E... untitled - Paint 2 Microsoft ... 4:05 PM

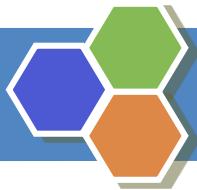


## Hello Application

Validation Error: Value is required.

Name:





**Input Name Page - Microsoft Internet Explorer provided by Cognitiv**

File Edit View Favorites Tools Help

Back → × Search Favorites

Address http://localhost:8080/jsfks/faces/pages/inputname.jsp

# Hello Application

Name:

**Greeting Page - Microsoft Internet Explorer provided by Cognitiv**

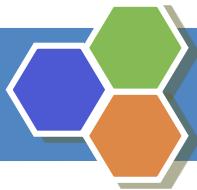
File Edit View Favorites Tools Help

Back → × Search Favorites

Address http://localhost:8080/jsfks/faces/pages/inputname.jsp

# Hello, JSF!





Input Name Page - Microsoft Internet Explorer provided by Co

File Edit View Favorites Tools Help

Back → Stop Refresh Home Search Favorites

Address http://localhost:8080/jsfks/faces/pages/inputname.jsp

# Hello Application

Name: JS  Submit

Input Name Page - Microsoft Internet Explorer provided by Co

File Edit View Favorites Tools Help

Back → Stop Refresh Home Search Favorites

Address http://localhost:8080/jsfks/faces/pages/inputname.jsp

# Hello Application

Validation Error: Value is less than allowable minimum of '3'.

Name: JS  Submit

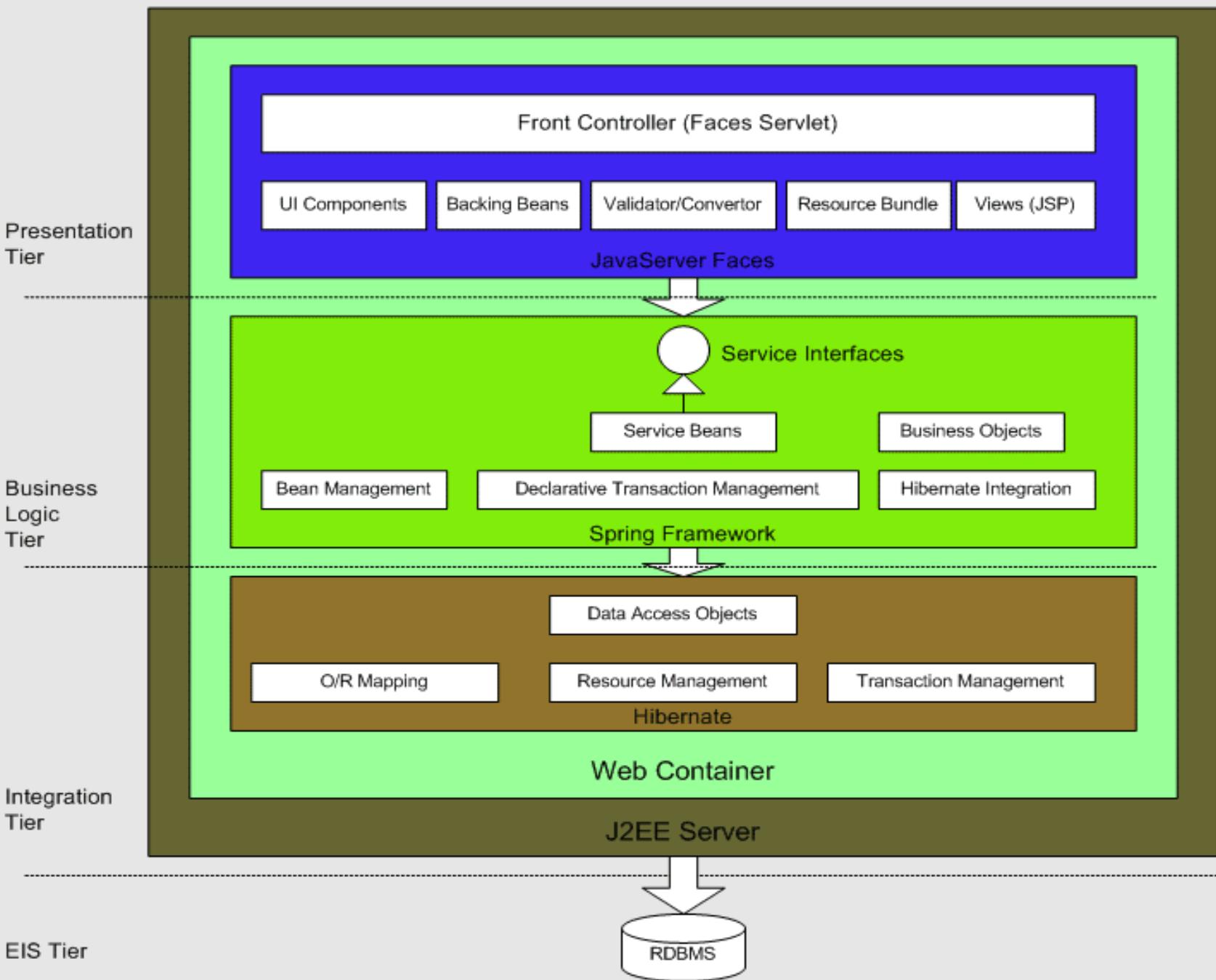




# JSF in Multitiered architecture

- JavaServer Faces (JSF) technology is a new user interface framework for J2EE applications.
- It is particularly suited, by design, for use with applications based on the MVC (Model-View-Controller) architecture.
- JSF can be integrated with other Java frameworks—like Spring, hibernate to build a real-world Web application.







# Presentation tier

- JavaServer Faces
  - **Collect user input**
  - **Present data**
  - **Control page navigation**
  - **Delegate user input to the business-logic tier**
  - **The presentation tier can also validate user input and maintain the application's session state.**





# Bussiness Logic Tier

- EJB (Enterprise JavaBeans) or POJO (plain old Java objects)
- EJB
  - **Provides Remote interface**
  - **Useful if the application is distributed.**
- POJO
- Simple but Doesn't Provide Remote interface
- For a typical Web application with no remote access required, POJO, with the help of the Spring Framework, can be used to implement the business-logic tier.

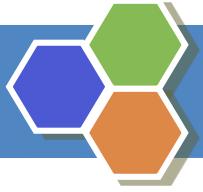




# Integration tier

- The integration tier handles the data persistence with the relational database.
- Different approaches:
  - **Pure JDBC (Java Database Connectivity):**
    - The most flexible approach; however, low-level JDBC is cumbersome to work with, and bad JDBC code does not perform well.
  - **Entity beans:**
    - An expensive way to isolate data-access code and handle O/R (object-relational) mapping data persistence.
    - An application-server-centric approach.
    - Does not tie the application to a particular type of database, but does tie the application to the EJB container.
  - **O/R mapping framework:**
    - Takes an object-centric approach to implementing data persistence.
    - Easy to develop and highly portable.
    - Example :JDO (Java Data Objects), Hibernate.





# Spring 3

## Spring

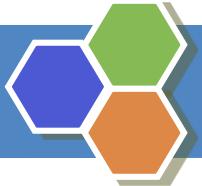




# What is Spring?

- Spring is a **lightweight dependency injection** and **aspect-oriented container** and **framework**

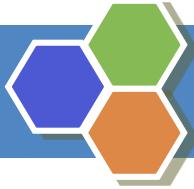




- ***Lightweight***

- Spring is lightweight in terms of both size and overhead.
- The bulk of the Spring framework can be distributed in a single JAR file that weighs in at just over 2.5 MB.
- Processing overhead required by Spring is negligible.
- Spring is nonintrusive: objects in a Spring-enabled application often have no dependencies on Spring-specific classes.

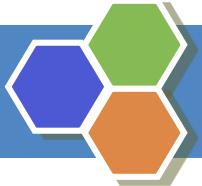




- ***Dependency Injection***

- Spring promotes loose coupling through a technique known as dependency injection (DI).
- When DI is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves.
- You can think of DI as JNDI in reverse—instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

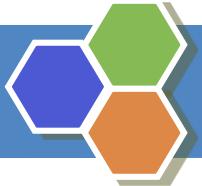




- ***Aspect-oriented Programming***

- Spring support aspect-oriented programming (AOP)
- AOP enables cohesive development by separating application business logic from system services (such as auditing and transaction management).
- Application objects do what they're supposed to do—perform business logic—and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.

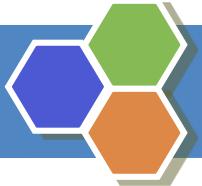




- ***Container***

- Spring is a container in the sense that it contains and manages the lifecycle and configuration of application objects.
- In Spring, you can declare how each of your application objects should be created, how they should be configured, and how they should be associated with each other.

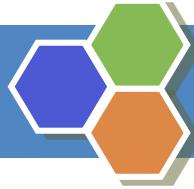




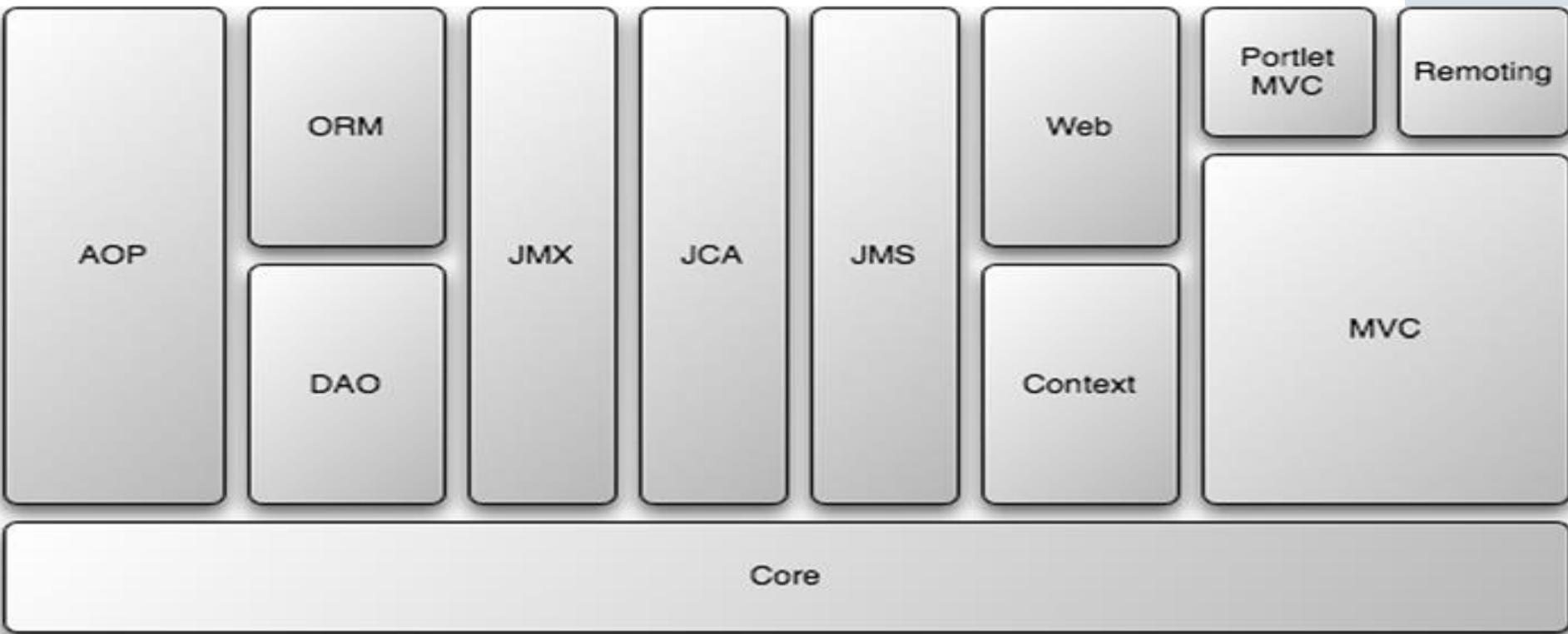
- ***Framework***

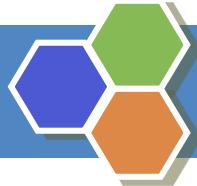
- Spring makes it possible to configure and compose complex applications from simpler components.
- In Spring, application objects are composed declaratively, typically in an XML file.
- Spring also provides much infrastructure functionality (transaction management, persistence framework integration, etc.), leaving the development of application logic to you.





# Spring Modules



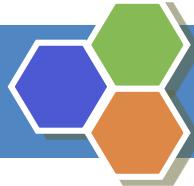


## Core Container

- Spring's core container provides the fundamental functionality of the Spring Framework. This module contains the BeanFactory, which is the fundamental Spring container and the basis on which Spring's DI is based.

- 





# ApplicationContext Module

- The core module's BeanFactory makes Spring a container, but the context module is what makes it a framework.
- This module extends the concept of BeanFactory, adding support for I18N, messages, application lifecycle events, and validation.
- This module supplies many enterprise services such as email, JNDI access, EJB integration, remoting, and scheduling.





# Spring AOP Module

- Like DI, AOP supports loose coupling of application objects.
- With AOP, however, applicationwide concerns (such as transactions and security) are decoupled from the objects to which they are applied.
- Spring's AOP module offers several approaches to building aspects, including building aspects based on AOP Alliance interfaces and support for AspectJ.





# JDBC abstraction and the DAO

- This module abstracts away the boilerplate code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources.
- This module also builds a layer of meaningful exceptions on top of the error messages given by several database servers.
- This module uses Spring's AOP module to provide transaction management services for objects in a Spring application.





# ORM integration module

- Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions.
- Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps.
- Spring's transaction management supports each of these ORM frameworks as well as JDBC.





# Java Management Extensions (JMX)

- Exposing the inner workings of a Java application for management is a critical part of making an application production ready.
- Spring's JMX module makes it easy to expose your application's beans as JMX MBeans.
- This makes it possible to monitor and reconfigure a running application.
- 

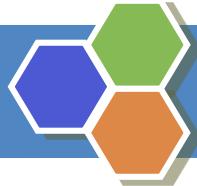




# Java EE Connector API (JCA)

- JCA provides a standard way of integrating Java applications with a variety of enterprise information systems, including mainframes and databases.
- In many ways, JCA is much like JDBC, except where JDBC is focused on database access, JCA is a more general-purpose API connecting to legacy systems.
- Spring's support for JCA is similar to its JDBC support, abstracting away JCA's boilerplate code into templates.





# The Spring MVC framework

- Spring integrates with several popular MVC frameworks, it also comes with its own very capable MVC framework that promotes Spring's loosely coupled techniques in the web layer of an application.





# Spring Portlet MVC

- Portlet-based applications aggregate several bits of functionality on a single web page.
- This provides a view into several applications at once.
- Spring Portlet MVC builds on Spring MVC to provide a set of controllers that support Java's portlet API.

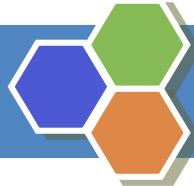




# Spring's web module

- Spring MVC and Spring Portlet MVC require special consideration when loading the Spring application context.
- Spring's web module provides special support classes for Spring MVC and Spring Portlet MVC.
- The web module also contains support for several web-oriented tasks, such as multipart file uploads and programmatic binding of request parameters to your business objects.

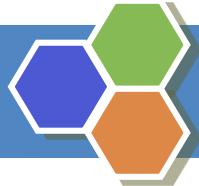




# Remoting

- Spring's remoting support enables you to expose the functionality of your Java objects as remote objects.
- Or if you need to access objects remotely, the remoting module also makes simple work of wiring remote objects into your application as if they were local POJOs.
- Several remoting options are available, including Remote Method Invocation (RMI), Hessian, Burlap, JAX-RPC, and Spring's own HTTP Invoker.

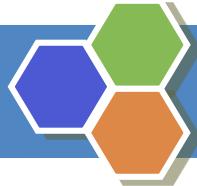




# Java Message Service (JMS)

- Spring's Java Message Service (JMS) module helps you send messages to JMS message queues and topics.
- This module also helps you create message-driven POJOs that are capable of consuming asynchronous messages.





# Bean Wiring

- In Spring, objects are not responsible for finding or creating the other objects that they need to do their job. Instead, they are given references to the objects that they collaborate with by the container.
- The act of creating associations between application objects is the essence of dependency injection (DI) and is commonly referred to as *wiring*.





- In a Spring-based application, your application objects will live within the Spring container. The container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave.





# Container Implementation

- **Bean factories** - defined by `org.springframework.beans.factory.BeanFactory` interface are the simplest of containers, providing basic support for DI.
- **Application contexts** - defined by `org.springframework.context.ApplicationContext` interface build on the notion of a bean factory by providing application framework services, such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners

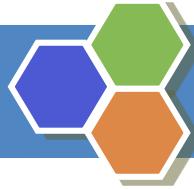




# BeanFactory

- A bean factory is a general-purpose factory, creating and dispensing many types of beans.
- A bean factory also takes part in the lifecycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.





# Resource

- **ByteArrayResource** -- Defines a resource whose content is given by an array of bytes
- **ClassPathResource** -- Defines a resource that is to be retrieved from the classpath
- **DescriptiveResource** -- Defines a resource that holds a resource description but no actual readable resource
- **FileSystemResource** -- Defines a resource that is to be retrieved from the file system
- **InputStream – Resource** Defines a resource that is to be retrieved from an input stream





# ApplicationContext

- ApplicationContext offers:
  - Application contexts provide a means for resolving text messages, including support for internationalization (I18N) of those messages.
  - Application contexts provide a generic way to load file resources, such as images.
  - Application contexts can publish events to beans that are registered as listeners.





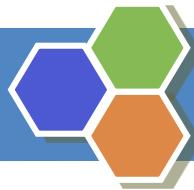
- `ClassPathXmlApplicationContext` — Loads a context definition from an XML file located in the classpath, treating context definition files as classpath resources.
- `FileSystemXmlApplicationContext` — Loads a context definition from an XML file in the file system.
- `XmlWebApplicationContext` — Loads context definitions from an XML file contained within a web application.





- `ApplicationContext context = new  
FileSystemXmlApplicationContext("c:/foo.xml");`
- `ApplicationContext context = new  
ClassPathXmlApplicationContext("foo.xml");`

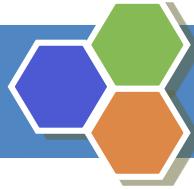




# Bean Scope

Scope	Description
Singleton	Scopes the bean definition to a single instance per Spring container (default).
Prototype	Allows a bean to be instantiated any number of times (once per use).
Request	Scopes a bean definition to an HTTP request. Only valid when used with a web-capable Spring context (such as with Spring MVC).
Session	Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).
Global-session	Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

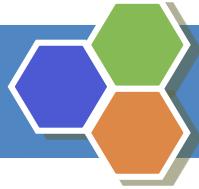




# Wiring Collection

Collection	Description
<list>	Wiring a list of values, allowing duplicates
<set>	Wiring a set of values, ensuring no duplicates
<map>	Wiring a collection of name-value pairs where name and value can be of any type
<prop>	Wiring a collection of name-value pairs where the name and value are both Strings

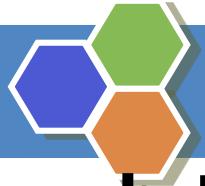




# AutoWiring

- Autowiring helps reduce or even eliminate the need for <property> and <constructor-arg> elements.
- Spring automatically figure out how to wire bean dependencies.





## AutoWiring types

- **byName** — Match all properties of the autowired bean with beans that have the same name (or ID) as the properties.
- **byType** — Match all properties of the autowired bean with beans whose types are assignable to the properties.
- **constructor** — Match a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.
- **autodetect** — First try constructor autowiring, if fails then byType autowiring





## byName

- byName autowiring establishes a convention where a property will automatically be wired with a bean of the same name.
- The downside of using byName autowiring is that it assumes that you'll have a bean whose name is the same as the name of the property of another bean.





- When attempting to autowire a property by type, Spring will look for beans whose type is assignable to the property's type.
- Limitation
  - In case of Spring finds more than one bean whose type is assignable to the autowired property, then Spring will throw an exception.
  - You're allowed to have only one bean configured that matches the autowired property. Whereas, there may be several beans whose types are subclasses of **Instrument**.





- To overcome ambiguities with autowiring by type, Spring offers two options:
  - You can either identify a primary candidate for autowiring
  - You can eliminate beans from autowiring candidacy.

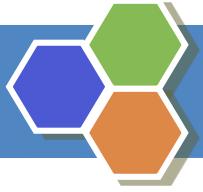




# defaultAutowire

- <beans ..... default-autowire="byType">
- By default default-autowire = none





# Any Queries...

