INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad – 500 043

Object Oriented Programming through Java

Course Code: ACS003

IV Semester (IARE – R16)

Prepared by:

Mr. G Chandra Sekhar

Assistant Professor

Mr. E Sunil Reddy

Assistant Professor

Introduction

- **OOP concepts-** Data abstraction- encapsulation- inheritance- benefits of inheritance- polymorphism-classes and objects- procedural and object oriented programming paradigm.
- **Java programming –** History of java- comments data types-variables-constants-scope and life time of variables-operators-operator hierarchy-expressions-type conversion and casting- enumerated types- control flow – block scope- conditional statements-loops-break and continue statements-simple java stand alone programs-arrays-console input and output-formatting output-constructors-methods-parameter passing- static fields and methods- access control- this reference- overloading methods and constructors-recursion-garbage collection- building strings- exploring string class

Need for OOP Paradigm

- OOP is an approach to program organization and development, which attempts to eliminate some of the drawbacks of conventional programming methods by incorporating the best of structured programming features with several new concepts.
- OOP allows us to decompose a problem into number of entities called objects and then build data and methods (functions) around these entities.
- The data of an object can be accessed only by the methods associated with the object.

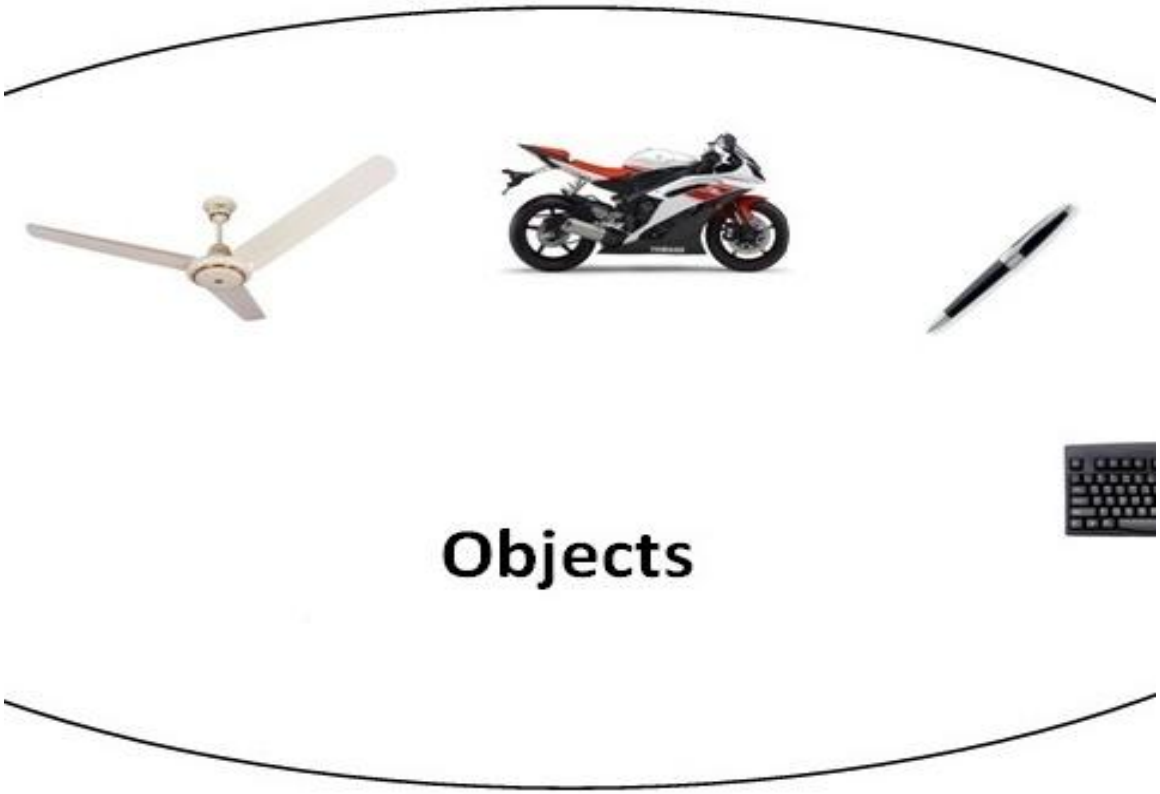
Object-oriented programming (OOP) is a programming paradigm that uses "Objects "and their interactions to design applications.

It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Data Abstraction & Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

Object

- Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.



Class

- The entire set of data and code of an object can be made of a user defined data type with the help of a class.
- In fact, Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class.
- **Classes** are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects.

- Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.
- A class is thus a collection of objects of similar type . for example: mango, apple, and orange are members of the class fruit . ex: fruit mango; will create an object mango belonging to the class fruit.

Example for class

```
class Human

{

 private:  EyeColor IColor;

 NAME personname;

public:

void SetName(NAME anyName);

 void SetIColor(EyeColor eyecolor);

};
```

Data abstraction

- Abstraction refers to the act of representing essential features without including the background details or explanations. since the classes use the concept of data abstraction ,they are known as *abstraction data type(ADT).*
- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only know how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

An example for abstraction

- Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic).
- What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car has different implementation of car working but its basic interface has not changed (i.e. you still use steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

Some of the Object-Oriented Paradigm are:

- Emphasis is on data rather than procedure.
- Programs are divided into objects.
- Data Structures are designed such that they Characterize the objects.
- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and can not be accessed by external functions.
- Objects may communicate with each other through methods.

Encapsulation

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java −

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.

Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the IS-A relationship which is also known as a parent-child relations

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Co

Terms used in Inheritance

- Class: A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- Reusability: As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.de Reusability.hip.

Benefit of using inheritance:

- A code can be used again and again
- Inheritance in Java enhances the properties of the class, which means that property of the parent class will automatically be inherited by the base class
- It can define more specialized classes by adding new details.

Polymorphism

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
- If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.
- Runtime Polymorphism in Java.

Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- Let's first understand the upcasting before Runtime Polymorphism.

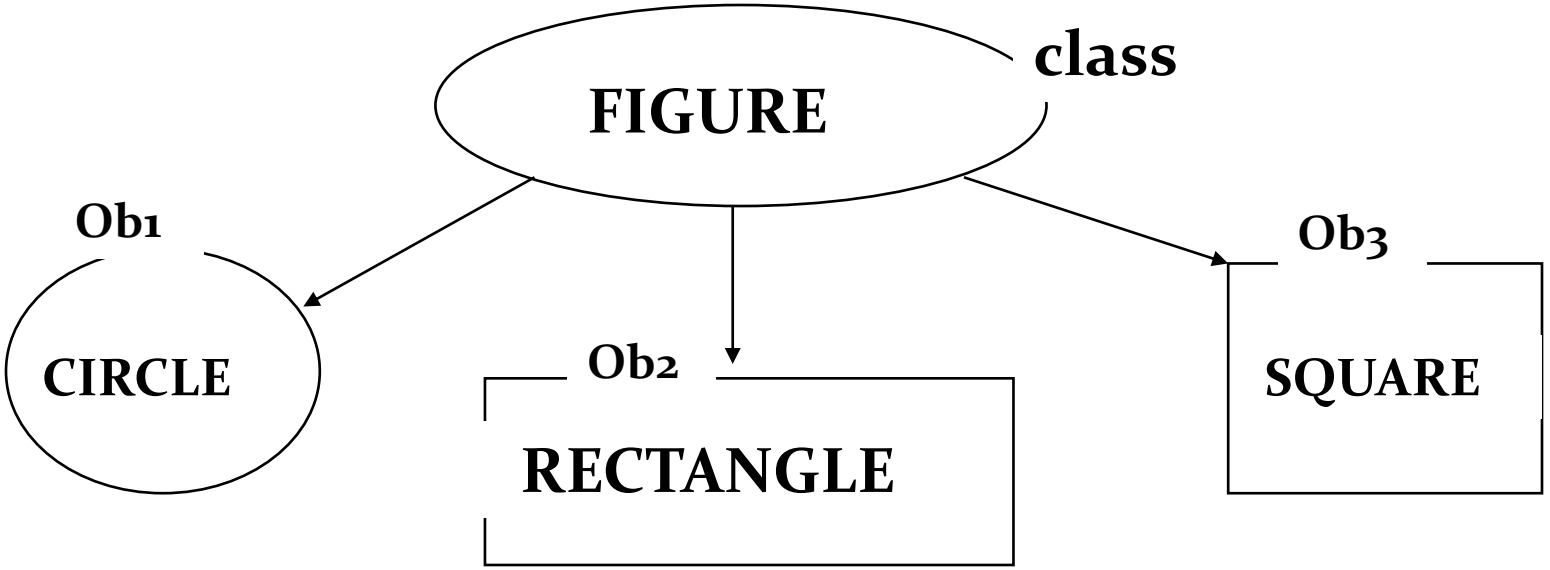
Need for OO Paradigm

Differences between Procedural and OO Programing

Procedural	OO programming
Code is placed into totally distinct functions or procedures	Everything treated as an Object
Data placed in separate structures and is manipulated by these functions or procedures	Every object consist of attributes(data) and behaviors (methods)
Code maintenance and reuse is difficult	Code maintenance and reuse is easy
Data is uncontrolled and unpredictable (i.e. multiple functions may have access to the global data)	The data of an object can be accessed only by the methods associated with the object
You have no control over who has access to the data	Good control over data access
Testing and debugging are much more difficult	Testing and debugging are much easy
Not easy to upgrade	Easy to upgrade
Not easy to partition the work in a project	Easy to partition the work in a project

CLASSES

- Class is blue print or an idea of an Object
- From One class any number of Instances can be created
- It is an encapsulation of attributes and methods



Syntax of CLASS

```
class <ClassName>
{
        attributes/variables;
        Constructors();
        methods();
}
```

Instance

- Instance is an Object of a class which is an entity with its own attribute values and methods.

Creating an Instance

```
ClassName  refVariable;
refVariable = new Constructor();
                or
ClassName  refVariable = new  Constructor();
```

Java Class Hierarchy

In Java, class “Object” is the base class to all other classes

- If we do not explicitly say extends in a new class definition, it implicitly extends Object
- The tree of classes that extend from Object and all of its subclasses are is called the class hierarchy
- All classes eventually lead back up to Object
- This will enable consistent access of objects of different classes.

Method Binding

- Objects are used to call methods.
- **MethodBinding** is an object that can be used to call an arbitrary public method, on an instance that is acquired by evaluating the leading portion of a method binding expression via a value binding.
- It is legal for a class to have two or more methods with the same name.
- Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:

  1) by the number of arguments, or

  2) by the types of arguments
- Overloading and inheritance are two ways to implement polymorphism.

History of Java

Computer language innovation and development occurs for two fundamental reasons:

1) to adapt to changing environments and uses

2) to implement improvements in the art of programming

- The development of Java was driven by both in equal measures.
- Many Java features are inherited from the earlier languages:

B → C → C++ → Java

Before Java: C

- Designed by Dennis Ritchie in 1970s.
- Before C: BASIC, COBOL, FORTRAN, PASCAL
- C- structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Designed, implemented and tested by programmers.

Before Java: C++

- Designed by Bjarne Stroustrup in 1979.
- Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:
  - 1) assembler languages
  - 2) high-level languages
  - 3) structured programming
  - 4) object-oriented programming (OOP)
- OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.
- C++ extends C by adding object-oriented features.

Java: History

- In 1990, Sun Microsystems started a project called Green.
- Objective: to develop software for consumer electronics.
- Project was assigned to James Gosling, a veteran of classic network software design. Others included Patrick Naughton, ChrisWarth, Ed Frank, and Mike Sheridan.
- The team started writing programs in C++ for embedding into
  - toasters
  - washing machines
  - VCR's
- Aim was to make these appliances more "intelligent".

Java: History (contd.)

- C++ is powerful, but also dangerous. The power and popularity of C derived from the extensive use of pointers. However, any incorrect use of pointers can cause memory leaks, leading the program to crash.
- In a complex program, such memory leaks are often hard to detect.
- Robustness is essential. Users have come to expect that Windows may crash or that a program running under Windows may crash. (“This program has performed an illegal operation and will be shut down”)
- However, users do not expect toasters to crash, or washing machines to crash.
- A design for consumer electronics has to be *robust*.
- Replacing pointers by references, and automating memory management was the proposed solution.

Java: History (contd.)

- Hence, the team built a new programming language called Oak, which avoided potentially dangerous constructs in C++, such as pointers, pointer arithmetic, operator overloading etc.
- Introduced automatic memory management, freeing the programmer to concentrate on other things.
- Architecture neutrality (Platform independence)
- Many different CPU's are used as controllers. Hardware chips are evolving rapidly. As better chips become available, older chips become obsolete and their production is stopped. Manufacturers of toasters and washing machines would like to use the chips available off the shelf, and would not like to reinvest in compiler development every two-three years.
- So, the software and programming language had to be *architecture neutral*.

Java: History (contd)

- It was soon realized that these design goals of consumer electronics perfectly suited an ideal programming language for the Internet and WWW, which should be:
  - ❖ object-oriented (& support GUI)
  - ❖ – robust
  - ❖ – architecture neutral
- Internet programming presented a BIG business opportunity. Much bigger than programming for consumer electronics.
- Java was “re-targeted” for the Internet
- The team was expanded to include Bill Joy (developer of Unix), Arthur van Hoff, Jonathan Payne, Frank Yellin, Tim Lindholm etc.
- In 1994, an early web browser called WebRunner was written in Oak. WebRunner was later renamed HotJava.
- In 1995, Oak was renamed Java.
- A common story is that the name Java relates to the place from where the development team got its coffee. The name Java survived the trade mark search.

Java History

- Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.
- With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.
- Java as an “Internet version of C++”? No.
- Java was not designed to replace C++, but to solve a different set of problems.

The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:
  - ❖ Simple
  - ❖ Secure
  - ❖ Portable
  - ❖ Object-oriented
  - ❖ Robust
  - ❖ Multithreaded
  - ❖ Architecture-neutral
  - ❖ Interpreted
  - ❖ High performance
  - ❖ Distributed
  - ❖ Dynamic

- **simple** – Java is designed to be easy for the professional programmer to learn and use.
- **object-oriented:** a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- **Robust:** restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.
- **Multithreaded:** supports multi-threaded programming for writing program that perform concurrent computations.
- **Architecture-neutral:** Java Virtual Machine provides a platform independent environment for the execution of Java byte code
- **Interpreted and high-performance:** Java programs are compiled into an intermediate representation – byte code:

  a) can be later interpreted by any JVM

  b) can be also translated into the native machine code for efficiency.

- **Distributed:** Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- **Dynamic:** substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- **Secure:** programs are confined to the Java execution environment and cannot access other parts of the computer.
- **Portability:** Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet.
- For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. The same mechanism that helps ensure security also helps create portability.
- Indeed, Java's solution to these two problems is both elegant and efficient.

Comments

- The java comments are statements that are not executed by the compiler and interpreter. The comments can be used to provide information or explanation about the variable, method, class or any statement. It can also be used to hide program code for specific time.

- Types of Java Comments
- There are 3 types of comments in java.

- Single Line Comment
- Multi Line Comment
- Documentation Comment

Comments

1) Java Single Line Comment

The single line comment is used to comment only one line.

- **Syntax:**
- //This is single line comment

Example:

```
public class CommentExample1 {
public static void main(String[] args) {
 int i=10;//Here, i is a variable
System.out.println(i);
}
}
```

Output:

10

Comments

2) Java Multi Line Comment

The multi line comment is used to comment multiple lines of code.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

```
public class CommentExample2 {
public static void main(String[] args) {
/* Let's declare and
 print variable in java. */
   int i=10;
   System.out.println(i);
} } Output: 10
```

Comments

3) Java Documentation Comment

The documentation comment is used to create documentation API. To create documentation API, you need to use **javadoc tool**.

Syntax:

```
/**
This
is
documentation
comment
*/
```

Data Types

- Java defines eight simple types:
  1)byte – 8-bit integer type
  2)short – 16-bit integer type
  3)int – 32-bit integer type
  4)long – 64-bit integer type
  5)float – 32-bit floating-point type
  6)double – 64-bit floating-point type
  7)char – symbols in a character set
  8)boolean – logical values true and false

- byte: 8-bit integer type.

  Range: -128 to 127.

  Example: byte b = -15;

  Usage: particularly when working with data streams.
- short: 16-bit integer type.

  Range: -32768 to 32767.

  Example: short c = 1000;

  Usage: probably the least used simple type.

- **int:** 32-bit integer type.

  Range: -2147483648 to 2147483647.

  Example: int b = -50000;

  Usage:

  1) Most common integer type.

  2) Typically used to control loops and to index arrays.

  3) Expressions involving the byte, short and int values are promoted to int before calculation.

- **long:** 64-bit integer type.

  Range: -9223372036854775808 to 9223372036854775807.

  Example: long l = 10000000000000000;

  Usage: 1) useful when int type is not large enough to hold the desired value

- **float:** 32-bit floating-point number.

  Range: 1.4e-045 to 3.4e+038.

  Example: float f = 1.5;

  Usage:

  1) fractional part is needed

  2) large degree of precision is not required

- **double:** 64-bit floating-point number.

  Range: 4.9e-324 to 1.8e+308.

  Example: double pi = 3.1416;

  Usage:

  1) accuracy over many iterative calculations

  2) manipulation of large-valued numbers

char: 16-bit data type used to store characters.

Range: 0 to 65536.

Example: char c = ‘a’;

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a

character set with characters found in (almost) all human languages.

2) Not the same as in C/C++ where char is 8-bit and represents ASCII only.

- **boolean:** Two-valued type of logical values.

  Range: values true and false.

  Example: boolean b = (1<2);

  Usage:

  1) returned by relational operators, such as 1<2

  2) required by branching expressions such as if or for

Variables

- declaration – how to assign a type to a variable
- initialization – how to give an initial value to a variable
- scope – how the variable is visible to other parts of the program
- lifetime – how the variable is created, used and destroyed
- type conversion – how Java handles automatic type conversion
- type casting – how the type of a variable can be narrowed down

Variables

- Java uses variables to store data.
- To allocate memory space for a variable JVM requires:
  1) to specify the data type of the variable
  2) to associate an identifier with the variable
  3) optionally, the variable may be assigned an initial value
- All done as part of variable declaration.

Basic Variable Declaration

- datatype identifier [=value];
- datatype must be
  - A simple datatype
  - User defined datatype (class type)
- Identifier is a recognizable name confirm to identifier rules
- Value is an optional initial value.

Variable Declaration

- We can declare several variables at the same time:

  type identifier [=value][, identifier [=value] …];

  Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte g = 22;
double pi = 3.14159;
char ch = 'x';
```

Variable Scope

- Scope determines the visibility of program elements with respect to other program elements.
- In Java, scope is defined separately for classes and methods:

  1) variables defined by a class have a global scope

  2) variables defined by a method have a local scope

  A scope is defined by a block:

```
{
…
}
```

  A variable declared inside the scope is not visible outside:

```
{
int n;
}
n = 1;// this is illegal
```

Variable Lifetime

- Variables are created when their scope is entered by control flow and destroyed when their scope is left:
- A variable declared in a method will not hold its value between different invocations of this method.
- A variable declared in a block looses its value when the block is left.
- Initialized in a block, a variable will be re-initialized with every re-entry. Variables lifetime is confined to its scope!

Operators Types

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
    1) assignment
    2) arithmetic
    3) relational
    4) logical
    5) bitwise

Arithmetic assignments

+=	v += expr;	v = v + expr ;
-=	v -=expr;	v = v - expr ;
*=	v *= expr;	v = v * expr ;
/=	v /= expr;	v = v / expr ;
%=	v %= expr;	v = v % expr ;

Basic Arithmetic Operators

+	op1 + op2	ADD
-	op1 - op2	SUBSTRACT
*	op1 * op2	MULTIPLY
/	op1 / op2	DIVISION
%	op1 % op2	REMAINDER

Relational operator

==	Equals to	Apply to any type
!=	Not equals to	Apply to any type
>	Greater than	Apply to numerical type
<	Less than	Apply to numerical type
>=	Greater than or equal	Apply to numerical type
<=	Less than or equal	Apply to numerical type

Logical operators

&	op1 & op2	Logical AND
|	op1 | op2	Logical OR
&&	op1 && op2	Short-circuit AND
||	op1 || op2	Short-circuit OR
!	! op	Logical NOT
^	op1 ^ op2	Logical XOR

Bit wise operators

~	~op	Inverts all bits
&	op1 & op2	Produces 1 bit if both operands are 1
|	op1 |op2	Produces 1 bit if either operand is 1
^	op1 ^ op2	Produces 1 bit if exactly one operand is 1
>>	op1 >> op2	Shifts all bits in op1 right by the value of op2
<<	op1 << op2	Shifts all bits in op1 left by the value of op2

Operator Hierarchy

Operators	Precedence
postfix increment and decrement	`++ --`
prefix increment and decrement, and unary	`++ -- + - ~ !`
multiplicative	`* / %`
additive	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
bitwise AND	`&`
bitwise exclusive OR	`^`
bitwise inclusive OR	`|`
logical AND	`&&`
logical OR	`||`
ternary	`? :`
assignment	`= += -= *= /= %= &= ^= |= <<= >>=>>>=`

Expressions

- An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.
- Examples of expressions are in bold below:

```
int number = 0;
 anArray[0] = 100;
System.out.println ("Element 1 at index 0: " + anArray[0]);
 int result = 1 + 2; // result is now 3 if(value1 == value2)
 System.out.println("value1 == value2");
```

Expressions

- The data type of the value returned by an expression depends on the elements used in the expression.
- The expression number = 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, number is an int.
- As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String. The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Here's an example of a compound expression: 1 * 2 * 3

Type Conversion

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting

Type Conversion

- Widening Type Converstion
  - Implicit conversion by compiler automatically

byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

Type Conversion

- Narrowing Type Conversion
  - Programmer should describe the conversion explicitly

byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float

Type Conversion

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double

Type Casting

- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

Control Statements

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:

  1) selection statements allow the program to choose different parts of the execution based on the outcome of an expression

  2) iteration statements enable program execution to repeat one or more statements

  3) jump statements enable your program to execute in a non-linear fashion

Selection Statements

- Java selection statements allow to control the flow of program's execution based upon conditions known only during run-time.
- Java provides four selection statements:

    1) if

    2) if-else

    3) if-else-if

    4) switch

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:
    1) while
    2) do-while
    3) for

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
    1) break
    2) continue
    3) return
- In addition, Java supports exception handling that can also alter the control flow of a program.

Simple Java Program

- A class to display a simple message:

```
class MyProgram
{
public static void main(String[] args)
 {
System.out.println(“First Java program.");
}
}
```

Arrays

- An array is a group of liked-typed variables referred to by a common
- name, with individual variables accessed by their index.
- Arrays are:
    1) declared
    2) created
    3) initialized
    4) used
- Also, arrays can have one or several dimensions.

Array Declaration

- Array declaration involves:

1) declaring an array identifier
2) declaring the number of dimensions
3) declaring the data type of the array elements

- Two styles of array declaration:

type array-variable[];

or

type [] array-variable;

Array Creation

- After declaration, no array actually exists.
- In order to create an array, we use the new operator:

  type array-variable[];

  array-variable = new type[size];
- This creates a new array to hold size elements of type type, which reference will be kept in the variable array-variable.

Array Indexing

- Later we can refer to the elements of this array through their indexes:
- array-variable[index]
- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

Array Initialization

- Arrays can be initialized when they are declared:
- int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
- Note:

1) there is no need to use the new operator

2) the array is created large enough to hold all specified elements

Multidimensional Arrays

- Multidimensional arrays are arrays of arrays:

  1) declaration: int array[][];

  2) creation: int array = new int[2][3];

  3) initialization

  int array[][] = { {1, 2, 3}, {4, 5, 6} };

What is an Object?

- Real world objects are things that have:

  1) state

  2) behavior

  Example: your dog:
- state – name, color, breed, sits?, barks?, wages tail?, runs?
- behavior – sitting, barking, waging tail, running
- A software object is a bundle of variables (state) and methods (operations).

What is a Class?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

Object Creation

- A variable is declared to refer to the objects of type/class String:

  String s;

- The value of s is null; it does not yet refer to any object.
- A new String object is created in memory with initial “abc” value:
- String s = new String(“abc”);
- Now s contains the address of this new object.

Object Destruction

- A program accumulates memory through its execution.
- Two mechanism to free memory that is no longer need by the program:
    1) manual – done in C/C++
    2) automatic – done in Java
- In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.
- Garbage collector is parts of the Java Run-Time Environment.

Class

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:
- A class is a template for objects
- An object is an instance of a class

Class Definition

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:

```
class classname {
        type instance-variable-1;
        …
        type instance-variable-n;
        type method-name-1(parameter-list) { … }
        type method-name-2(parameter-list) { … }
        …
        type method-name-m(parameter-list) { … }
            }
```

Example: Class Usage

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
vol = mybox.width * mybox.height * mybox.depth;
System.out.println ("Volume is " + vol);
} }
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.
    1) it is syntactically similar to a method:
    2) it has the same name as the name of its class
    3) it is written without return type; the default return type of a class
- constructor is the same classWhen the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

```
class Box {
double width;
double height;
double depth;
Box() {
System.out.println("Constructing Box");
width = 10; height = 10; depth = 10;
}
double volume() {
return width * height * depth;
}
}
```

Parameterized Constructor

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
double volume()
 { return width * height * depth;
}
}
```

Methods

- General form of a method definition:

```
type name(parameter-list) {
                    … return value;
                …
            }
```

- Components:

  1) type - type of values returned by the method. If a method does not return any value, its return type must be void.

  2) name is the name of the method

  3) parameter-list is a sequence of type-identifier lists separated by commas

  4) return value indicates what value is returned by the method.

Example: Method

- Classes declare methods to hide their internal data structures, as well as for their own internal use: Within a class, we can refer directly to its member variables:

```
class Box {
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Parameterized Method

- Parameters increase generality and applicability of a method:
- 1) method without parameters

  int square() { return 10*10; }
- 2) method with parameters

  int square(int i) { return i*i; }
- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

Access Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods.
- *Encapsulation*, safely sealing data within the *capsule* of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

Access Modifiers: Public, Private, Protected

- *Public:* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- Default(No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.

- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Visibility

```
public class Circle {
    private double x,y,r;

        // Constructor
     public Circle (double x, double y, double r) {
            this.x = x;
            this.y = y;
            this.r = r;
     }
     //Methods to return circumference and area
    public double circumference() { return 2*3.14*r;}
    public double area() { return 3.14 * r * r; }
}
```

String Handling

- **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects.
- For example, in the statement

  System.out.println("This is a String, too");

  the string "This is a String, too" is a **String** constant

- Java defines one operator for **String** objects: **+**.
- It is used to concatenate two strings. For example, this statement
- String myString = "I" + " like " + "Java.";

  results in **myString** containing

  "I like Java."

- The **String** class contains several methods that you can use. Here are a few. You can
- test two strings for equality by using

  equals( ). You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:
- // Demonstrating some String methods.

```
class StringDemo2 {
public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;          System.out.println("Length of strOb1: " +
                                                    strOb1.length());
```

```
System.out.println ("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))

System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
} }
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

UNIT-2

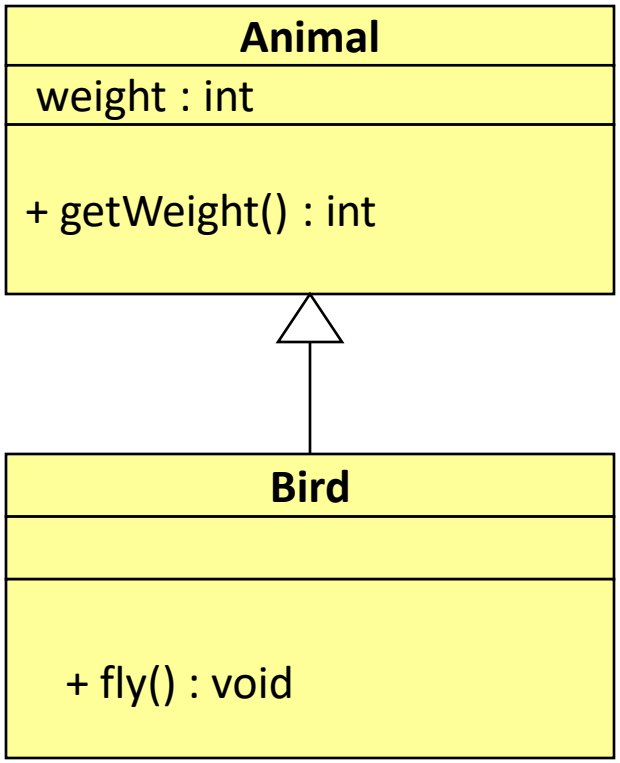
MULTIPLE INHERITANCE ,INTERFACES AND PACKAGES

Inheritance

- Methods allows a software developer to reuse a sequence of statements
- *Inheritance* allows a software developer to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class *inherits* the methods and data defined for the parent class

Inheritance

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Deriving Subclasses

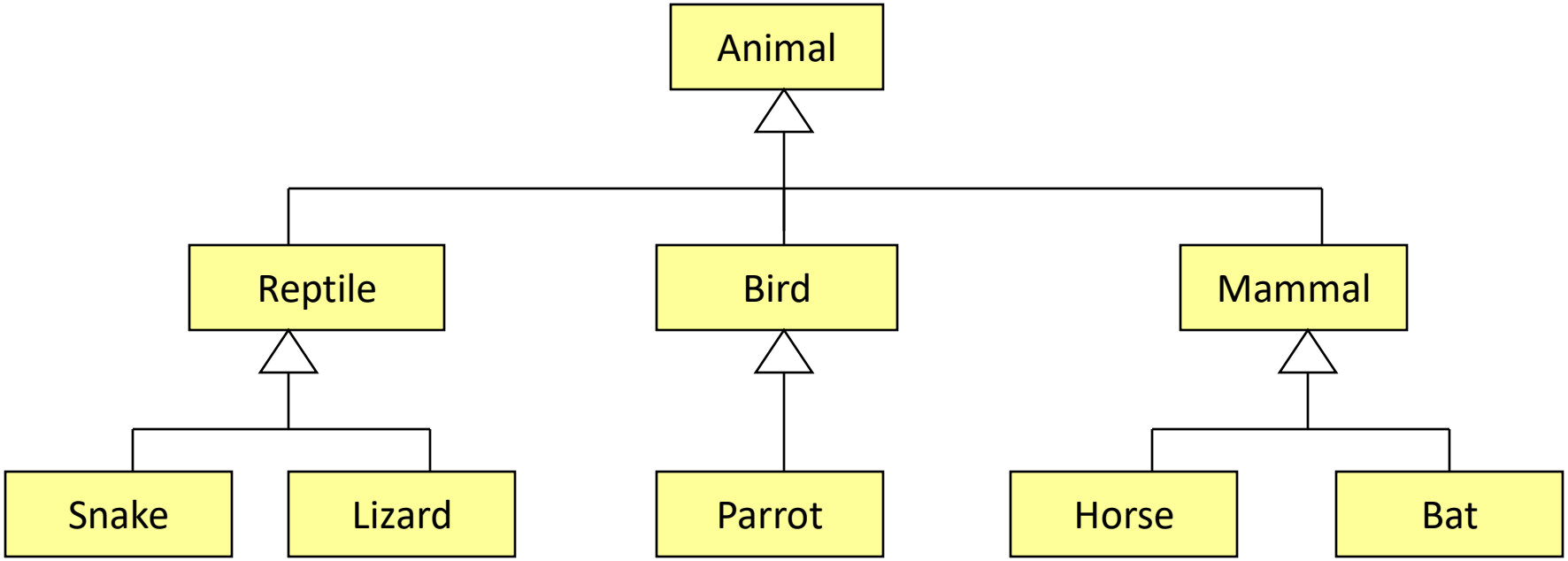
- In Java, we use the reserved word `extends` to establish an inheritance relationship

```
class Animal
{
  // class contents
   int weight;
   public void int getWeight() {…}
}

class Bird extends Animal
{
   // class contents
   public void fly() {…};
}
```

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies*



- At the top of the hierarchy there's a default class called *Object*.

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- inheritance is transitive
  - An instance of class Parrot is also an instance of Bird, an instance of Animal, …, and an instance of class Object
- The class hierarchy determines how methods are executed:
  - Previously, we took the simplified view that when variable v is an instance of class C, then a procedure call v.proc1() invokes the method proc1() defined in class C
  - However, if C is a child of some superclass C' (and hence v is both an instance of C *and an instance* of C'), the picture becomes more complex, because methods of class C can *override* the methods of class C' (next two slides).

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body

- In java, all methods except of constructors override the methods of their ancestor class by replacement. E.g.:
  - the Animal class has method eat()
  - the Bird class has method eat() and Bird extends Animal
  - variable *b* is of class Bird, i.e. Bird b = …
  - b.eat() simply invokes the eat() method of the Bird class

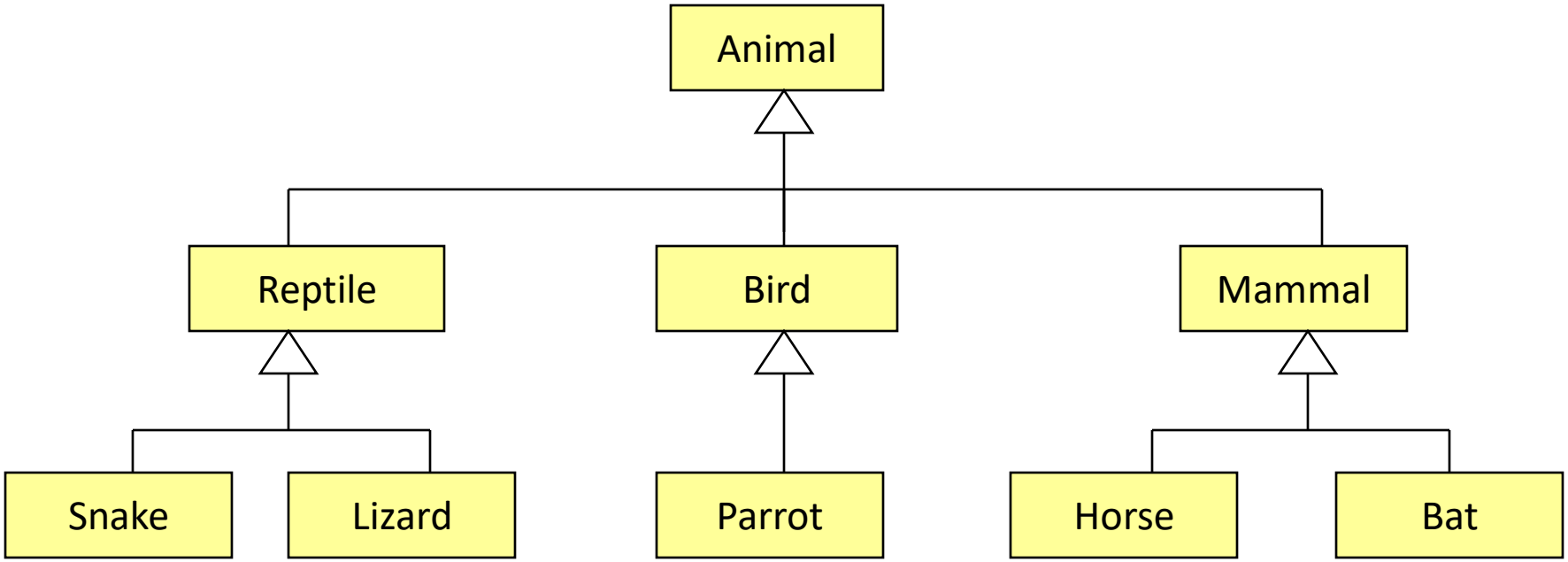
- If a method is declared with the `final` modifier, it cannot be overridden

Defining Methods in the Child Class: Overriding by Refinement

- Constructors in a subclass *override* the definition of an inherited constructor method by *refining* them (instead of replacing them)
  - Assume class Animal has constructors
    Animal(), Animal(int weight), Animal(int weight, int livespan)
  - Assume class Bird which extends Animal has constructors
    Bird(), Bird(int weight), Bird(int weight, int livespan)
  - Let's say we create a Bird object, e.g. Bird b = Bird(5)
  - This will invoke ***first*** the constructor of the Animal (the superclass of Bird) and ***then*** the constructor of the Bird
- This is called *constructor chaining:* If class C0 extends C1 and C1 extends C2 and … Cn-1 extends Cn = Object then when creating an instance of object C0 first constructor of Cn is invoked, then constructors of Cn-1, …, C2, C1, and finally the constructor of C
  - The constructors (in each case) are chosen by their signature, e.g. (), (int), etc…
    - If no constructor with matching signature is found in any of the class Ci for i>0 then the default constructor is executed for that class
    - If no constructor with matching signature is found in the class C0 then this causes a compiler errorFirst the new method must have the same signature as the parent's method, but can have different code in the body

Recap: Class Hierarchy

- In Java, a **class can extend a single other class**
  (If none is stated then it implicitly extends an Object class)



- Imagine what would happen to method handling rules if every class could extend two others…
  (Answer: It would create multiple problems!)

Hierarchical Abstraction

- An essential element of object-oriented programming is *abstraction.*

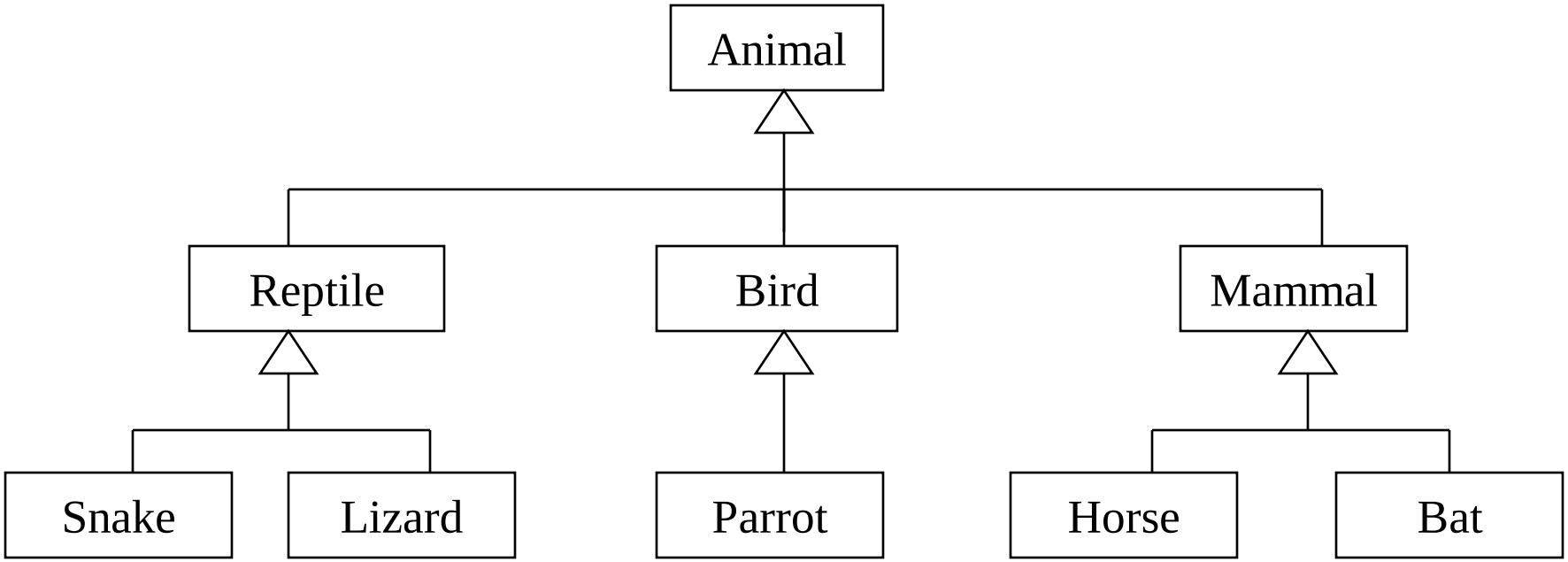
- Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior.

- This abstraction allows people to use a car without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work.

- Instead they are free to utilize the object as a whole.

Class Hierarchy

- A child class of one parent can be the parent of another child, forming *class hierarchies*



  - At the top of the hierarchy there's a default class called *Object*.

Class Hierarchy

- Good class design puts all common features as high in the hierarchy as reasonable
- The class hierarchy determines how methods are executed
- inheritance is transitive
  - An instance of class Parrot is also an instance of Bird, an instance of Animal, …, and an instance of class Object

Base Class **Object**

- In Java, all classes use inheritance.
- If no parent class is specified explicitly, the base class **Object** is implicitly inherited.
- All classes defined in Java, is a child of **Object** class, which provides minimal functionality guaranteed to e common to all objects.
- Methods defined in Object class are;

1. equals(Object obj): Determine whether the argument object is the same as the receiver.
2. getClass(): Returns the class of the receiver, an object of type Class.
3. hashCode(): Returns a hash value for this object. Should be overridden when the equals method is changed.
4. toString(): Converts object into a string value. This method is also often overridden.

Base class

1) a class obtains variables and methods from another class

2) the former is called subclass, the latter super-class (Base class)

3) a sub-class provides a specialized behavior with respect to its super-class

4) inheritance facilitates code reuse and avoids duplication of data

Extends

- Is a keyword used to inherit a class from another class
- Allows to extend from only one class

```
class One
{       int a=5;
}
```

```
class Two extends One
{
        int b=10;
}
```

Subclass, Subtype and Substitutability

- A subtype is a class that satisfies the principle of substitutability.
- A subclass is something constructed using inheritance, whether or not it satisfies the principle of substitutability.
- The two concepts are independent. Not all subclasses are subtypes, and (at least in some languages) you can construct subtypes that are not subclasses.
- Substitutability is fundamental to many of the powerful software development techniques in OOP.
- The idea is that, declared a variable in one type may hold the value of different type.
- Substitutability can occur through use of inheritance, whether using **extends**, or using **implements** keywords.

When new classes are constructed using inheritance, the argument used to justify the validity of substitutability is as follows;

- Instances of the subclass must possess all data fields associated with its parent class.
- Instances of the subclass must implement, through inheritance at least, all functionality defined for parent class. (Defining new methods is not important for the argument.)
- Thus, an instance of a child class can mimic the behavior of the parent class and should be *indistinguishable* from an instance of parent class if substituted in a similar situation.

Subclass, Subtype, and Substitutability

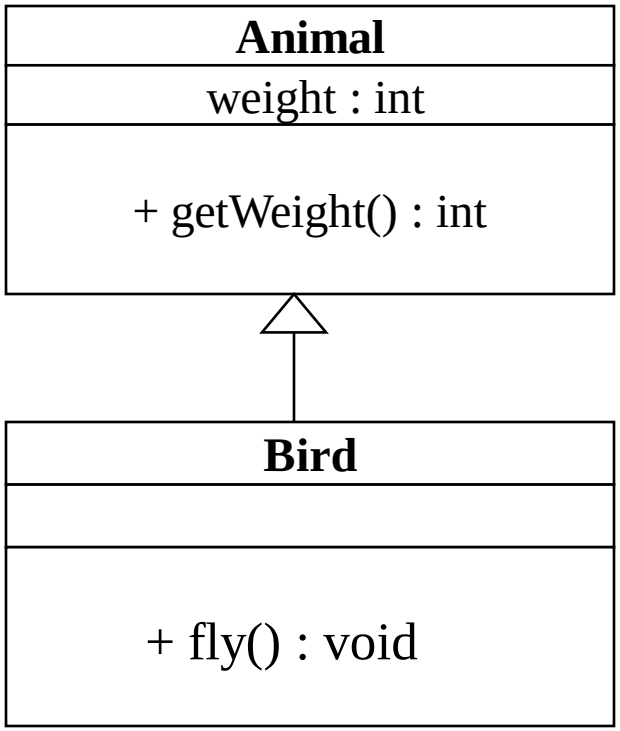
- The term *subtype* is used to describe the relationship between types that explicitly recognizes the principle of *substitution*. A type B is considered to be a subtype of A if an instances of B can legally be assigned to a variable declared as of type A.
- The term *subclass* refers to inheritance mechanism made by extends keyword.
- Not all *subclasses* are *subtypes*. *Subtypes* can also be formed using *interface*, linking types that have no inheritance relationship.

Subclass

- Methods allows to reuse a sequence of statements
- *Inheritance* allows to reuse classes by deriving a new class from an existing one
- The existing class is called the *parent class,* or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent(i.e the child class *inherits* the methods and data defined for the parent class

Subtype

- Inheritance relationships are often shown graphically in a *class diagram*, with the arrow pointing to the parent class



Substitutability (Deriving Subclasses)

- In Java, we use the reserved word extends to establish an inheritance relationship

```
class Animal
    {
      // class contents
       int weight;
 public void int getWeight() {…}
    }

    class Bird extends Animal
    {
       // class contents
 public void fly() {…};
    }
```

Defining Methods in the Child Class: Overriding by Replacement

- A child class can *override* the definition of an inherited method in favor of its own
  - that is, a child can redefine a method that it inherits from its parent
  - the new method must have the same signature as the parent's method, but can have different code in the body

- In java, all methods except of constructors override the methods of their ancestor class by replacement. E.g.:
  - the Animal class has method eat()
  - the Bird class has method eat() and Bird extends Animal
  - variable *b* is of class Bird, i.e. Bird b = …
  - b.eat() simply invokes the eat() method of the Bird class

- If a method is declared with the `final` modifier, it cannot be overridden

Forms of Inheritance

Inheritance is used in a variety of way and for a variety of different purposes .

- Inheritance for Specialization
- Inheritance for Specification
- Inheritance for Construction
- Inheritance for Extension
- Inheritance for Limitation
- Inheritance for Combination

One or many of these forms may occur in a single case.

Forms of Inheritance

(- *Inheritance for Specialization* -)

Most commonly used inheritance and sub classification is for specialization.

Always creates a subtype, and the principles of substitutability is explicitly upheld.

It is the most ideal form of inheritance.

An example of subclassification for specialization is;

```
public class PinBallGame extends Frame {

// body of class

}
```

Specialization

- By far the most common form of inheritance is for specialization.
  - Child class is a specialized form of parent class
  - Principle of substitutability holds
- A good example is the Java hierarchy of Graphical components in the AWT:
  - Component
    - Label
    - Button
    - TextComponent
      - TextArea
      - TextField
    - CheckBox
    - ScrollBar

Forms of Inheritance
(- ***Inheritance for Specification*** -)

This is another most common use of inheritance. Two different mechanisms are provided by Java, ***interface*** and ***abstract***, to make use of *subclassification for specification*. Subtype is formed and substitutability is explicitly upheld.

Mostly, not used for refinement of its parent class, but instead is used for definitions of the properties provided by its parent.

```
class FireButtonListener implements ActionListener {

// body of class

}

class B extends A {

// class A is defined as abstract specification class

}
```

Specification

- The next most common form of inheritance involves specification. The parent class specifies some behavior, but does not implement the behavior
  - Child class implements the behavior
  - Similar to Java interface or abstract class
  - When parent class does not implement actual behavior but merely defines the behavior that will be implemented in child classes
- Example, Java 1.1 Event Listeners:

  ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

Forms of Inheritance

(- *Inheritance for Construction* -)

Child class inherits most of its functionality from parent, but may change the name or parameters of methods inherited from parent class to form its interface.

This type of inheritance is also widely used for code reuse purposes. It simplifies the construction of newly formed abstraction but is not a form of subtype, and often violates substitutability.

Example is ***Stack*** class defined in Java libraries.

Construction

- The parent class is used only for its behavior, the child class has no *is-a* relationship to the parent.
  - Child modify the arguments or names of methods
  -
- An example might be subclassing the idea of a *Set* from an existing *List* class.
  - Child class is not a more specialized form of parent class; no substitutability

Forms of Inheritance
(- ***Inheritance for Extension*** -)

Subclassification for extension occurs when a child class only adds new behavior to the parent class and does not modify or alter any of the inherited attributes.

Such subclasses are always subtypes, and substitutability can be used.

Example of this type of inheritance is done in the definition of the class Properties which is an extension of the class HashTable.

Generalization or Extension

- The child class generalizes or extends the parent class by providing more functionality
  - In some sense, opposite of subclassing for specialization
- The child doesn't change anything inherited from the parent, it simply adds new features
  - Often used when we cannot modify existing base parent class
- Example, ColoredWindow inheriting from Window
  - Add additional data fields
  - Override window display methods

Forms of Inheritance
(- ***Inheritance for Limitation*** -)

Subclassification for limitation occurs when the behavior of the subclass is smaller or more restrictive that the behavior of its parent class.

Like subclassification for extension, this form of inheritance occurs most frequently when a programmer is building on a base of existing classes.

Is not a subtype, and substitutability is not proper.

Limitation

- The child class limits some of the behavior of the parent class.
- Example, you have an existing List data type, and you want a Stack
- Inherit from List, but override the methods that allow access to elements other than top so as to produce errors.

This types of inheritance is known as *multiple inheritance* in Object Oriented Programming.

Although the Java does not permit a subclass to be formed be inheritance from more than one parent class, several approximations to the concept are possible.

Example of this type is Hole class defined as;

```
class  Hole extends Ball implements PinBallTarget{

// body of class

}
```

Combination

- Two or more classes that seem to be related, but its not clear who should be the parent and who should be the child.
- Example: Mouse and TouchPad and JoyStick
- Better solution, abstract out common parts to new parent class, and use subclassing for specialization.

Summary of Forms of Inheritance

- Specialization. The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class.
- Specification. The parent class defines behavior that is implemented in the child class but not in the parent class.
- Construction. The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class.
- Generalization. The child class modifies or overrides some of the methods of the parent class.
- Extension. The child class adds new functionality to the parent class, but does not change any inherited behavior.
- Limitation. The child class restricts the use of some of the behavior inherited from the parent class.
- Variance. The child class and parent class are variants of each other, and the class-subclass relationship is arbitrary.
- Combination. The child class inherits features from more than one parent class. This is multiple inheritance and will be the subject of a later chapter.

The Benefits of Inheritance

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

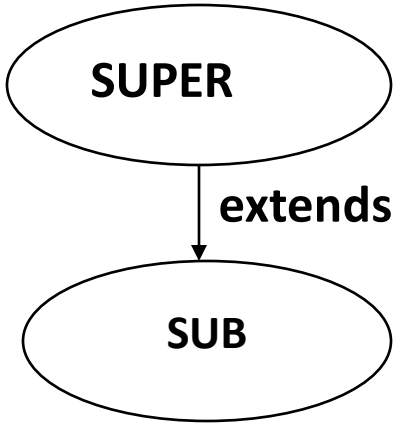
The Costs of Inheritance

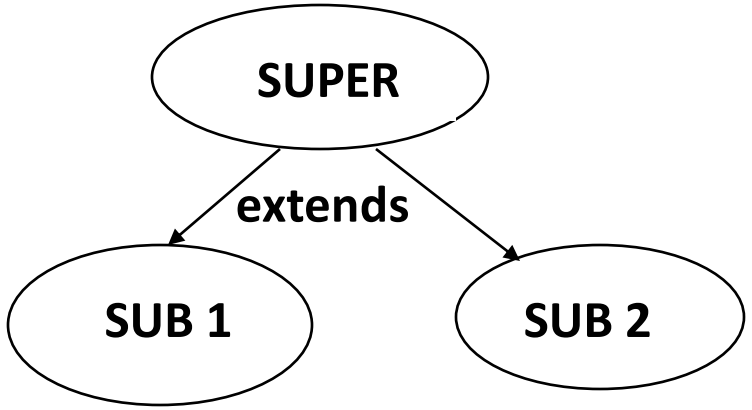
- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)

Types of inheritance

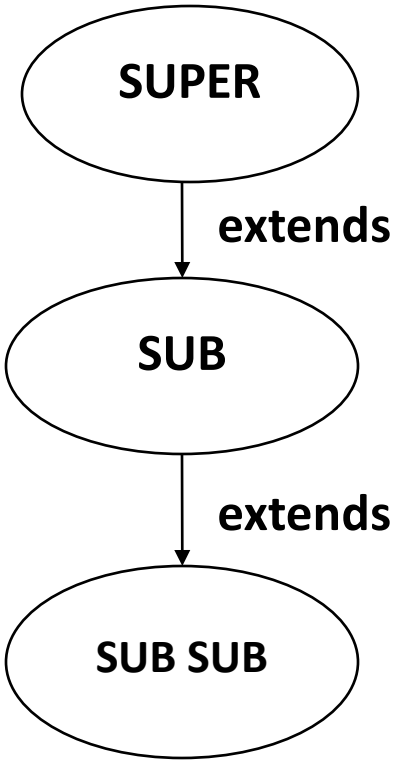
- Acquiring the properties of an existing Object into newly creating Object to overcome the re-declaration of properties in deferent classes.
- These are 3 types:

1.Simple Inheritance

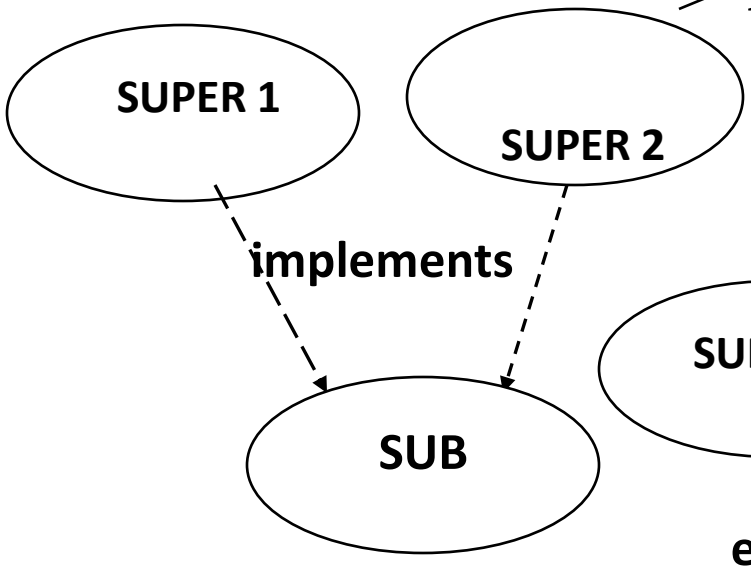


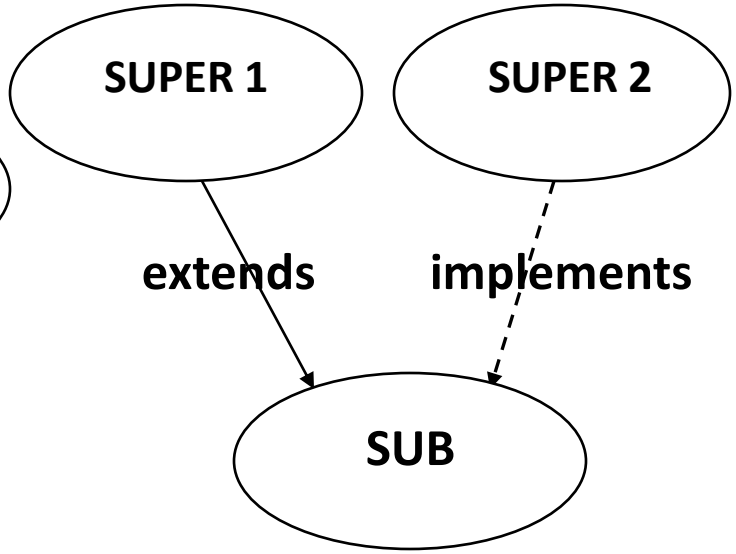


2. Multi Level Inheritance



3. Multiple Inheritance





Member access rules

- Visibility modifiers determine which class members are accessible and which do not
- Members (variables and methods) declared with public visibility are accessible, and those with private visibility are not
- Problem: How to make class/instance variables visible only to its subclasses?
- Solution: Java provides a third visibility modifier that helps in inheritance situations: protected

Modifiers and Inheritance (cont.)

Visibility Modifiers for class/interface:

`public` : can be accessed from outside the class definition.

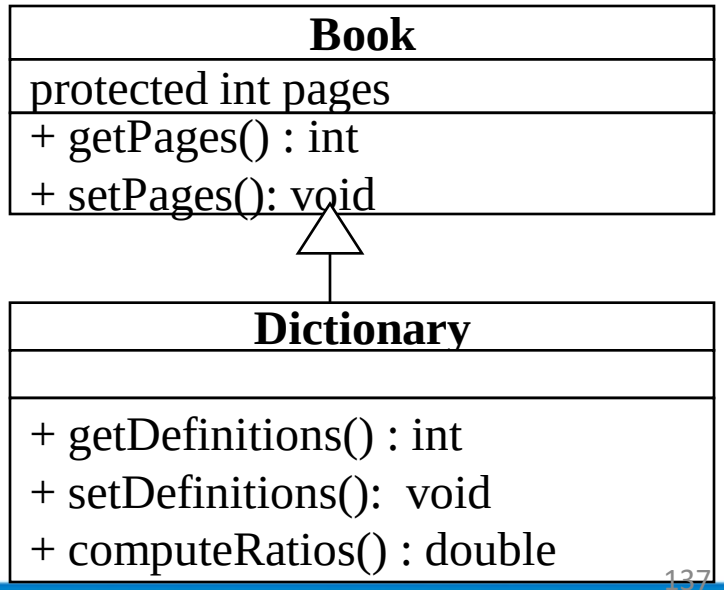
`protected` : can be accessed only within the class definition in which it appears, within other classess in the same package, or within the definition of subclassess.

`private` : can be accessed only within the class definition in which it appears.

default-access (if omitted) features accessible from inside the current Java package

The `protected` Modifier

- The `protected` visibility modifier allows a member of a base class to be accessed in the child
  - `protected` visibility provides more encapsulation than `public` does
  - `protected` visibility is not as tightly encapsulated as `private` visibility



“super” uses

- ‘super’ is a keyword used to refer to hidden variables of super class from sub class.
  - super.a=a;
- It is used to call a constructor of super class from constructor of sub class which should be first statement.
  - super(a,b);
- It is used to call a super class method from sub class method to avoid redundancy of code
  - super.addNumbers(a, b);

Super and Hiding

- Why is super needed to access super-class members?
- When a sub-class declares the variables or methods with the same names and types as its super-class:

```
class A {
int i = 1;
}
class B extends A {
int i = 2;
System.out.println("i is " + i);
}
```

- The re-declared variables/methods hide those of the super-class.

Example: Super and Hiding

```
class A {
int i;
}
class B extends A {
int i;
B(int a, int b) {
super.i = a; i = b;
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
```

Example: Super and Hiding

- Although the i variable in B hides the i variable in A, super allows access to the hidden variable of the super-class:

```
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Using final with inheritance

- final keyword is used declare constants which can not change its value of definition.
- final Variables can not change its value.
- final Methods can not be Overridden or Over Loaded
- final Classes can not be extended or inherited

Preventing Overriding with final

- A method declared final cannot be overridden in any sub-class:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
```

This class declaration is illegal:

```
class B extends A {
void meth() {
System.out.println("Illegal!");
}
}
```

Preventing Inheritance with final

- A class declared final cannot be inherited – has no sub-classes.

  final class A { … }

- This class declaration is considered illegal:

  class B extends A { … }

- Declaring a class final implicitly declares all its methods final.

- It is illegal to declare a class as both abstract and final.

Object class and its methods

- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Polymorphism

- Polymorphism is one of three pillars of object-orientation.
- Polymorphism: many different (poly) forms of objects that share a common interface respond differently when a method of that interface is invoked:
  1) a super-class defines the common interface
  2) sub-classes have to follow this interface (inheritance), but are also permitted to provide their own implementations (overriding)
- A sub-class provides a specialized behaviors relying on the common elements defined by its super-class.

Polymorphism

- A polymorphic reference can refer to different types of objects at different times
  - In java every reference can be polymorphic except of references to base types and final classes.

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
  - Polymorphic references are therefore resolved at run-time, not during compilation; this is called *dynamic binding*

- Careful use of polymorphic references can lead to elegant, robust software designs

Method Overriding

- When a method of a sub-class has the same name and type as a method of the super-class, we say that this method is overridden.

- When an overridden method is called from within the sub-class:

  1) it will always refer to the sub-class method

  2) super-class method is hidden

Example: Hiding with Overriding 1

```
class A {
int i, j;
A(int a, int b) {
i = a; j = b;
}
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
```

Example: Hiding with Overriding 2

```
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
void show() {
System.out.println("k: " + k);
}
}
```

Example: Hiding with Overriding 3

- When show() is invoked on an object of type B, the version of show() defined in B is used:

```
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show();
}
}
```

- The version of show() in A is hidden through overriding.

Overloading vs. Overriding

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overloading lets you define a similar operation in different ways for different data

- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

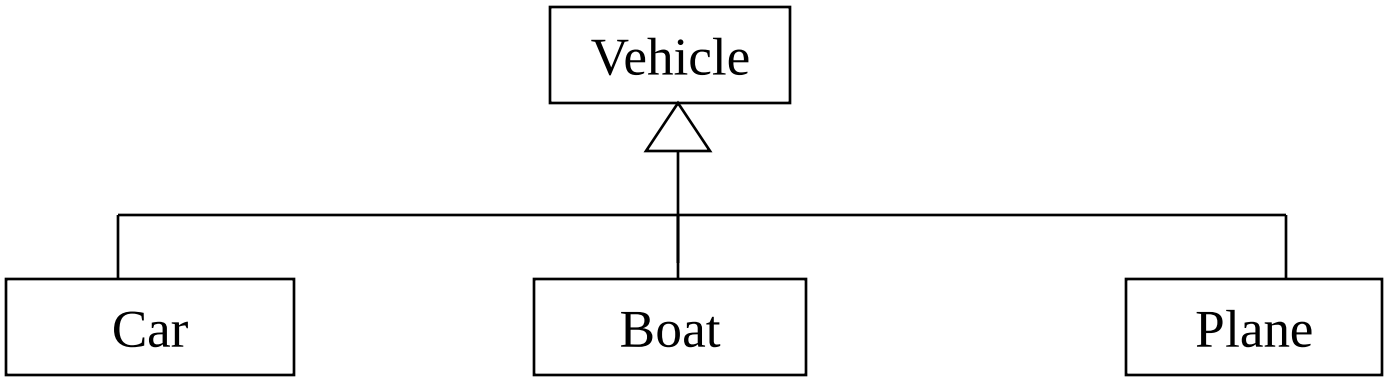
o Overriding lets you define a similar operation in different ways for different object types

Abstract Classes

- Java allows abstract classes
  - use the modifier `abstract` on a class header to declare an abstract class

```
abstract class Vehicle
{ … }
```

- An abstract class is a placeholder in a class hierarchy that represents a generic concept



Abstract Class: Example

- An abstract class often contains *abstract methods*, though it doesn't have to
  - Abstract methods consist of only methods *declarations*, without any method body

```
public abstract class Vehicle
{
   String name;
   public String getName()
         { return name; } \\ method body

   abstract public void move();
                              \\ no body!
}
```

Abstract Classes

- An abstract class often contains *abstract methods*, though it doesn't have to
  - Abstract methods consist of only methods *declarations,* without any method body
- The non-abstract child of an abstract class must override the abstract methods of the parent
- An abstract class cannot be instantiated
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

Abstract Method

- Inheritance allows a sub-class to override the methods of its super-class.
- A super-class may altogether leave the implementation details of a method and declare such a method abstract:
- abstract type name(parameter-list);
- Two kinds of methods:
  1) concrete – may be overridden by sub-classes
  2) abstract – must be overridden by sub-classes
- It is illegal to define abstract constructors or static methods.

Defining a Package

- A package is both a naming and a visibility control mechanism:

  1) divides the name space into disjoint subsets It is possible to define classes within a package that are not accessible by code outside the package.

  2) controls the visibility of classes and their members It is possible to define class members that are only exposed to other members of the same package.

- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages

Creating a Package

- A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- means that all classes in this file belong to the myPackage package.

- The package statement creates a name space where such classes are stored.

- When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

- Other files may include the same package instruction:

1. package myPackage;

   class MyClass1 { … }

   class MyClass2 { … }

2. package myPackage;

   class MyClass3{ … }

- A package may be distributed through several source files

Packages and Directories

- Java uses file system directories to store packages.

- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The byte code files MyClass1.class and MyClass2.class must be stored in a directory myPackage.

- Case is significant! Directory names must match package names exactly.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:
  1) Unix myPackage1/myPackage2/myPackage3
  2) Windows myPackage1\myPackage2\myPackage3
  3) Macintosh myPackage1:myPackage2:myPackage3

- You cannot rename a package without renaming its directory!

Accessing a Package

- As packages are stored in directories, how does the Java run-time system know where to look for packages?

- Two ways:

  1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.

  2) Specify a directory path or paths by setting the CLASSPATH environment variable.

CLASSPATH Variable

- CLASSPATH - environment variable that points to the root directory of the system's package hierarchy.
- Several root directories may be specified in CLASSPATH,
- e.g. the current directory and the C:\raju\myJava directory:

  .;C:\raju\myJava
- Java will search for the required packages by looking up subsequent directories described in the CLASSPATH variable.

Finding Packages

- Consider this package statement:

  package myPackage;

In order for a program to find myPackage, one of the following must be true:

1) program is executed from the directory immediately above myPackage (the parent of myPackage directory)

2) CLASSPATH must be set to include the path to myPackage

Example: Package

```
package MyPack;

class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n; bal = b;
}
void show() {
if (bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}   }
```

Example: Package

```
class AccountBalance
{
public static void main(String args[])
{
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for (int i=0; i<3; i++) current[i].show();
}
}
```

Example: Package

- Save, compile and execute:
  1) call the file AccountBalance.java
  2) save the file in the directory MyPack
  3) compile; AccountBalance.class should be also in MyPack
  4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
  5) run: java MyPack.AccountBalance

- Make sure to use the package-qualified class name.

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.

- The import statement allows to use classes or whole packages directly.

- Importing of a concrete class:

  import myPackage1.myPackage2.myClass;

- Importing of all classes within a package:

  import myPackage1.myPackage2.*;

Import Statement

- The import statement occurs immediately after the package statement and before the class statement:

  package myPackage;

- import otherPackage1;otherPackage2.otherClass;
  class myClass { … }

- The Java system accepts this import statement by default:
  import java.lang.*;

- This package includes the basic language functions. Without such functions, Java is of no much use.

Example: Packages 1

- A package MyPack with one public class Balance.

  The class has two same-package variables: public constructor and a public show method.

```
package MyPack;
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n; bal = b;
}
public void show() {
if (bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}
}
```

Example: Packages 2

The importing code has access to the public class Balance of the MyPack package and its two public members:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show();
}
}
```

Java Source File

Finally, a Java source file consists of:

1) a single package instruction (optional)

2) several import statements (optional)

3) a single public class declaration (required)

4) several classes private to the package (optional)

At the minimum, a file contains a single public class declaration.

Differences between classes and interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.
- Interface is little bit like a class... but interface is lack in instance variables....that's u can't create object for it.....
- Interfaces are developed to support multiple inheritance...
- The methods present in interfaces r pure abstract..
- The access specifiers public,private,protected are possible with classes, but the interface uses only one spcifier public.....
- interfaces contains only the method declarations.... no definitions.....
- A interface defines, which method a class has to implement. This is way - if you want to call a method defined by an interface - you don't need to know the exact class type of an object, you only need to know that it implements a specific interface.
- Another important point about interfaces is that a class can implement multiple interfaces.

Defining an interface

- Using interface, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.

❖ An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body.

For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {
// constant declarations double E = 2.718282;
// base of natural logarithms //
//method signatures
void doSomething (int i, double x);
int doSomethingElse(String s);
}
```

❖ The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

❖ An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends

Implementing interface

General format:

access interface name {

type method-name1(parameter-list);

type method-name2(parameter-list);

…

type var-name1 = value1;

type var-nameM = valueM;

…

}

- Two types of access:

  1) public – interface may be used anywhere in a program

  2) default – interface may be used in the current package only

- Interface methods have no bodies – they end with the semicolon after the parameter list.

- They are essentially abstract methods.

- An interface may include variables, but they must be final, static and initialized with a constant value.

- In a public interface, all members are implicitly public.

Interface Implementation

- A class implements an interface if it provides a complete set of methods defined by this interface.

  1) any number of classes may implement an interface

  2) one class may implement any number of interfaces

- Each class is free to determine the details of its implementation.

- Implementation relation is written with the implements keyword.

Implementation Format

- General format of a class that includes the implements clause:
- Syntax:

  access class name extends super-class implements interface1, interface2, …, interfaceN {

  …

  }
- Access is public or default.

Implementation Comments

- If a class implements several interfaces, they are separated with a comma.
- If a class implements two interfaces that declare the same method, the same method will be used by the clients of either interface.
- The methods that implement an interface must be declared public.
- The type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: Interface

Declaration of the Callback interface:

```
interface Callback
{
void callback(int param);
}
```

Client class implements the Callback interface:

```
class Client implements Callback
{
public void callback(int p)
{
System.out.println("callback called with " + p);
}
}
```

More Methods in Implementation

- An implementing class may also declare its own methods:

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement “ +
“interfaces may also define ” +
“other members, too.");
}
}
```

Applying interfaces

- A Java *interface* declares a set of method signatures i.e., says what behavior exists Does not say how the behavior is implemented

  i.e., does not give code for the methods
- Does not describe any state (but may include “final” constants)
- A concrete class that implements an interface Contains “implements *InterfaceName*” in the class declaration
- Must provide implementations (either directly or inherited from a superclass) of all methods declared in the interface
- An abstract class can also implement an interface
- Can optionally have implementations of some or all interface methods

- Interfaces and Extends both describe an “is- a” relation.
- If B *implements* interface A, then B inherits the (abstract) method signatures in A
- If B *extends* class A, then B inherits everything in A.
- which can include method code and instance variables as well as abstract method signatures.
- Inheritance” is sometimes used to talk about the superclass / subclass “extends” relation only

Variables in interface

- Variables declared in an interface must be constants.
- A technique to import shared constants into multiple classes:

  1) declare an interface with variables initialized to the desired values

  2) include that interface in a class through implementation.

- As no methods are included in the interface, the class does not implement.
- anything except importing the variables as constants.

Example: Interface Variables 1

An interface with constant values:

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
```

Example: Interface Variables 2

- Question implements SharedConstants, including all its constants.
- Which constant is returned depends on the generated random number:

```
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30) return NO;
else if (prob < 60) return YES;
else if (prob < 75) return LATER;
else if (prob < 98) return SOON;
else return NEVER;
}
}
```

Example: Interface Variables 3

- AskMe includes all shared constants in the same way, using them to display the result, depending on the value received:

```
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO: System.out.println("No"); break;
case YES: System.out.println("Yes"); break;
case MAYBE: System.out.println("Maybe"); break;
case LATER: System.out.println("Later"); break;
case SOON: System.out.println("Soon"); break;
case NEVER: System.out.println("Never"); break;
}
}
```

Example: Interface Variables 4

- The testing function relies on the fact that both ask and answer methods.

- defined in different classes, rely on the same constants:

```
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Extending interfaces

- One interface may inherit another interface.

- The inheritance syntax is the same for classes and interfaces.

```
interface MyInterface1 {
void myMethod1(…) ;
}
interface MyInterface2 extends MyInterface1 {
void myMethod2(…) ;
}
```

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example: Interface Inheritance 1

- Consider interfaces A and B.

```
interface A {
void meth1();
void meth2();
}
```

B extends A:

```
interface B extends A {
void meth3();
}
```

Example: Interface Inheritance 2

- MyClass must implement all of A and B methods:

```
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
} }
```

Example: Interface Inheritance 3

- Create a new MyClass object, then invoke all interface methods on it:

```
class IFExtend {
public static void main(String arg[]) {
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

UNIT-3

EXCEPTION HANDLING AND MULTITHREADING

Exceptions

- Exception is an abnormal condition that arises when executing a program.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:

  1) provides syntactic mechanisms to signal, detect and handle errors

  2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors

  3) brings run-time error management into object-oriented programming

Exception Handling

- An exception is an object that describes an exceptional condition (error) that has occurred when executing a program.

- Exception handling involves the following:
  1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it
  2) that method may choose to handle the exception itself or pass it on
  3) either way, at some point, the exception is caught and processed

Exception Sources

- Exceptions can be:
  1) generated by the Java run-time system Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
  2) manually generated by programmer's code Such exceptions are typically used to report some error conditions to the caller of a method.

Exception Constructs

- Five constructs are used in exception handling:
  1) try – a block surrounding program statements to monitor for exceptions
  2) catch – together with try, catches specific kinds of exceptions and handles them in some way
  3) finally – specifies any code that absolutely must be executed whether or not an exception occurs
  4) throw – used to throw a specific exception from the program
  5) throws – specifies which exceptions a given method can throw

Exception-Handling Block

General form:

```
try { … }
catch(Exception1 ex1) { … }
catch(Exception2 ex2) { … }
…
finally { … }
```

where:

1) try { … } is the block of code to monitor for exceptions
2) catch(Exception ex) { … } is exception handler for the exception Exception
3) finally { … } is the block of code to execute before the try block ends

Benefits of exception handling

- Separating Error-Handling code from “regular” business logic code
- Propagating errors up the call stack
- Grouping and differentiating error types

Using Java Exception Handling

```
method1 {
try {
call method2;
} catch (exception e) {
doErrorProcessing;
}
}
method2 throws exception {
call method3;
}
method3 throws exception {
call readFile;
}
```

❖ Any checked exceptions that can be thrown within a method must be specified in its throws clause.

Grouping and Differentiating Error Types

- Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy.
- An example of a group of related exception classes in the Java platform are those defined in java.io.IOException and its descendants.
- IOException is the most general and represents any type of error that can occur when performing I/O.
- Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.

- A method can write specific handlers that can handle a very specific exception.

- The FileNotFoundException class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
...
}
```

- A method can catch an exception based on its group or general type by specifying any of the exception's super classes in the catch statement.

- For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an IOException argument.

```
// Catch all I/O exceptions, including
// FileNotFoundException, EOFException, and so on.
catch (IOException e) {
...
}
```

Termination vs. Resumption

- There are two basic models in exception-handling theory.
- In *termination* the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don't *want* to come back.
- The alternative is called *resumption*. It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled.

- In resumption a method call that want resumption-like behavior (i.e don't throw an exception all a method that fixes the problem.)

- Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

- Operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption.

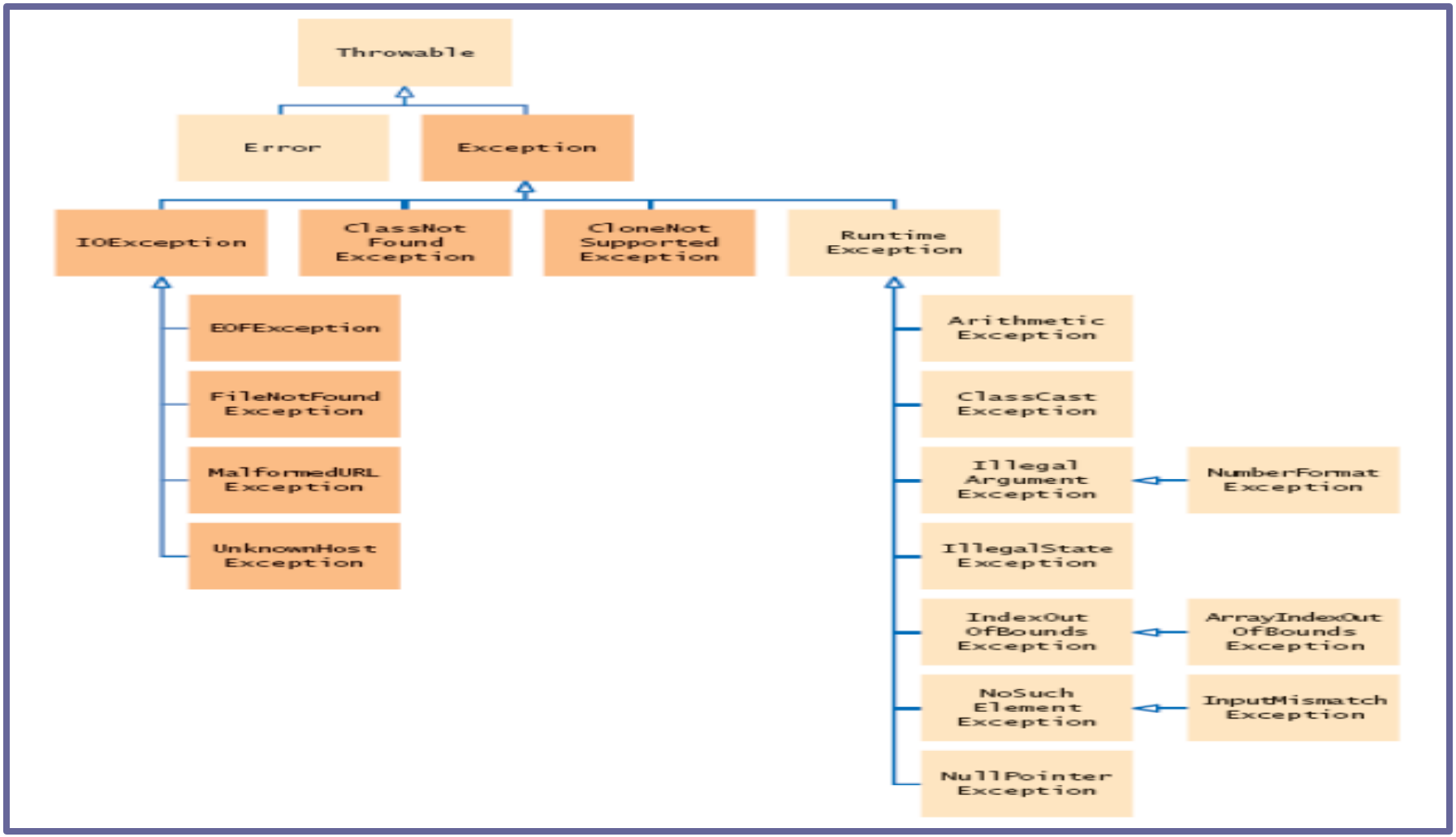
Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:

1) Exception – exceptional conditions that programs should catch
   The class includes:
   a) RuntimeException – defined automatically for user programs to include: division by zero, invalid array indexing, etc.
   b) use-defined exception classes

2) Error – exceptions used by Java to indicate errors with the runtime environment; user programs are not supposed to catch them

Hierarchy of Exception Classes



Usage of *try-catch* Statements

- Syntax:

```
try {
<code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
<handler if ExceptionType1 occurs>
} ...
} catch (<ExceptionTypeN> <ObjName>) {
<handler if ExceptionTypeN occurs>
}
```

Catching Exceptions: The *try-catch* Statements

```
class DivByZero {
public static void main(String args[]) {
try {
System.out.println(3/0);
System.out.println("Please print me.");
} catch (ArithmeticException exc) {
//Division by zero is an ArithmeticException
System.out.println(exc);
}
System.out.println("After exception.");
}
}
```

Catching Exceptions: Multiple catch

```
class MultipleCatch {
public static void main(String args[]) {
try {
int den = Integer.parseInt(args[0]);
System.out.println(3/den);
} catch (ArithmeticException exc) {
System.out.println("Divisor was 0.");
} catch (ArrayIndexOutOfBoundsException exc2) {
System.out.println("Missing argument.");
}
System.out.println("After exception.");
}
}
```

Catching Exceptions: Nested try's

```
class NestedTryDemo {
public static void main(String args[]){
try {
int a = Integer.parseInt(args[0]);
try {
int b = Integer.parseInt(args[1]);
System.out.println(a/b);
} catch (ArithmeticException e) {
System.out.println(“Div by zero error!");
} } catch (ArrayIndexOutOfBoundsException) {
System.out.println(“Need 2 parameters!");
} }   }
```

Catching Exceptions: Nested try's with methods

```
class NestedTryDemo2 {
 static void nestedTry(String args[]) {
 try {
 int a = Integer.parseInt(args[0]);
 int b = Integer.parseInt(args[1]);
 System.out.println(a/b);
 } catch (ArithmeticException e) {
System.out.println("Div by zero error!");
 }    }
public static void main(String args[]){
 try {
 nestedTry(args);
 } catch (ArrayIndexOutOfBoundsException e) {
 System.out.println("Need 2 parameters!");
 }  }   }
```

Throwing Exceptions(throw)

- So far, we were only catching the exceptions thrown by the Java system.
- In fact, a user program may throw an exception explicitly:

  throw ThrowableInstance;
- ThrowableInstance must be an object of type Throwable or its subclass.

Once an exception is thrown by:

  throw ThrowableInstance;

1) the flow of control stops immediately.

2) the nearest enclosing try statement is inspected if it has a catch statement that matches the type of exception:

1) if one exists, control is transferred to that statement

2) otherwise, the next enclosing try statement is examined

3) if no enclosing try statement has a corresponding catch clause, the default exception handler halts the program and prints the stack

Creating Exceptions

Two ways to obtain a Throwable instance:

1) creating one with the new operator

All Java built-in exceptions have at least two Constructors:

One without parameters and another with one String parameter:

```
throw new NullPointerException("demo");
```

2) using a parameter of the catch clause

try { … } catch(Throwable e) { … e … }

Example: throw 1

```
class ThrowDemo {
//The method demoproc throws a NullPointerException
  exception which is immediately caught in the try block and
  re-thrown:
static void demoproc() {
try {
throw new NullPointerException("demo");
} catch(NullPointerException e) {
System.out.println("Caught inside demoproc.");
throw e;
}
}
```

Example: throw 2

The main method calls demoproc within the try block which catches and handles the NullPointerException exception:

```
public static void main(String args[]) {
try {
demoproc();
} catch(NullPointerException e) {
System.out.println("Recaught: " + e);
}
}
}
```

throws Declaration

- If a method is capable of causing an exception that it does not handle, it must specify this behavior by the throws clause in its declaration:

```
type name(parameter-list) throws exception-list {
                    …
}
```

- where exception-list is a comma-separated list of all types of exceptions that a method might throw.

- All exceptions must be listed except Error and RuntimeException or any of their subclasses, otherwise a compile-time error occurs.

Example: throws 1

- The throwOne method throws an exception that it does not catch, nor declares it within the throws clause.

```
class ThrowsDemo {
static void throwOne() {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
throwOne();
}
}
```

- Therefore this program does not compile.

Example: throws 2

- Corrected program: throwOne lists exception, main catches it:

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

finally

- When an exception is thrown:
  1) the execution of a method is changed
  2) the method may even return prematurely.

- This may be a problem is many situations.

- For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.

- The finally block is used to address this problem.

finally Clause

- The try/catch statement requires at least one catch or finally clause, although both are optional:

```
try { … }
catch(Exception1 ex1) { … } …
finally { … }
```

- Executed after try/catch whether of not the exception is thrown.

- Any time a method is to return to a caller from inside the try/catch block via:

  1) uncaught exception or

  2) explicit return

  the finally clause is executed just before the method returns.

Example: finally 1

- Three methods to exit in various ways.

```
class FinallyDemo {
//procA prematurely breaks out of the try by throwing an
    exception, the finally clause is executed on the way out:
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}  }
```

Example: finally 2

// procB's try statement is exited via a return statement, the finally clause is executed before procB returns:

```
static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
```

Example: finally 3

- In procC, the try statement executes normally without error, however the finally clause is still executed:

```
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
```

Example: finally 4

- Demonstration of the three methods:

```
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

Java Built-In Exceptions

- The default java.lang package provides several exception classes, all sub-classing the RuntimeException class.

- Two sets of build-in exception classes:
  1. unchecked exceptions – the compiler does not check if a method handles or throws there exceptions

  2. checked exceptions – must be included in the method's throws clause if the method generates but does not handle them

Unchecked Built-In Exceptions

Methods that generate but do not handle those exceptions need not declare them in the throws clause:

1) ArithmeticException
2) ArrayIndexOutOfBoundsException
3) ArrayStoreException
4) ClassCastException
5) IllegalStateException
6) IllegalMonitorStateException
7) IllegalArgumentException
8. StringIndexOutOfBounds
9. UnsupportedOperationException
10. SecurityException
11. NumberFormatException
12. NullPointerException
13. NegativeArraySizeException
14. IndexOutOfBoundsException
15. IllegalThreadStateException

Checked Built-In Exceptions

Methods that generate but do not handle those exceptions must declare them in the throws clause:

1. NoSuchMethodException NoSuchFieldException
2. InterruptedException
3. InstantiationException
4. IllegalAccessException
5. CloneNotSupportedException
6. ClassNotFoundException

Creating Own Exception Classes

- Build-in exception classes handle some generic errors.

- For application-specific errors define your own exception classes. How? Define a subclass of Exception:

  class MyException extends Exception { … }

- MyException need not implement anything – its mere existence in the type system allows to use its objects as exceptions.

Example: Own Exceptions 1

- A new exception class is defined, with a private detail variable, a one parameter constructor and an overridden toString method:

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
```

Example: Own Exceptions 2

```
class ExceptionDemo {
The static compute method throws the MyException
exception whenever its a argument is greater than 10:
static void compute(int a) throws MyException {
System.out.println("Called compute(" + a + ")");
if (a > 10) throw new MyException(a);
System.out.println("Normal exit");
}
```

Example: Own Exceptions 3

The main method calls compute with two arguments within a try block that catches the MyException exception:

```
public static void main(String args[]) {
try {
compute(1);
compute(20);
} catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

Exception Summary

- **FileNotFoundException:** Signals that an attempt to open the file denoted by a specified pathname has failed.
- **InterruptedIOException:** Signals that an I/O operation has been interrupted
- **InvalidClassException:** Thrown when the Serialization runtime detects one of the following problems with a Class.
- **InvalidObjectException:** Indicates that one or more deserialized objects failed validation tests.
- **IOException:** Signals that an I/O exception of some sort has occurred.

Differences between multi threading and multitasking

Multi-Tasking

- Two kinds of multi-tasking:
  1) process-based multi-tasking
  2) thread-based multi-tasking

- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.

- Processes are heavyweight tasks:
  1) that require their own address space
  2) inter-process communication is expensive and limited
  3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
- several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:

  1) they share the same address space

  2) they cooperatively share the same process

  3) inter-thread communication is inexpensive

  4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:

  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

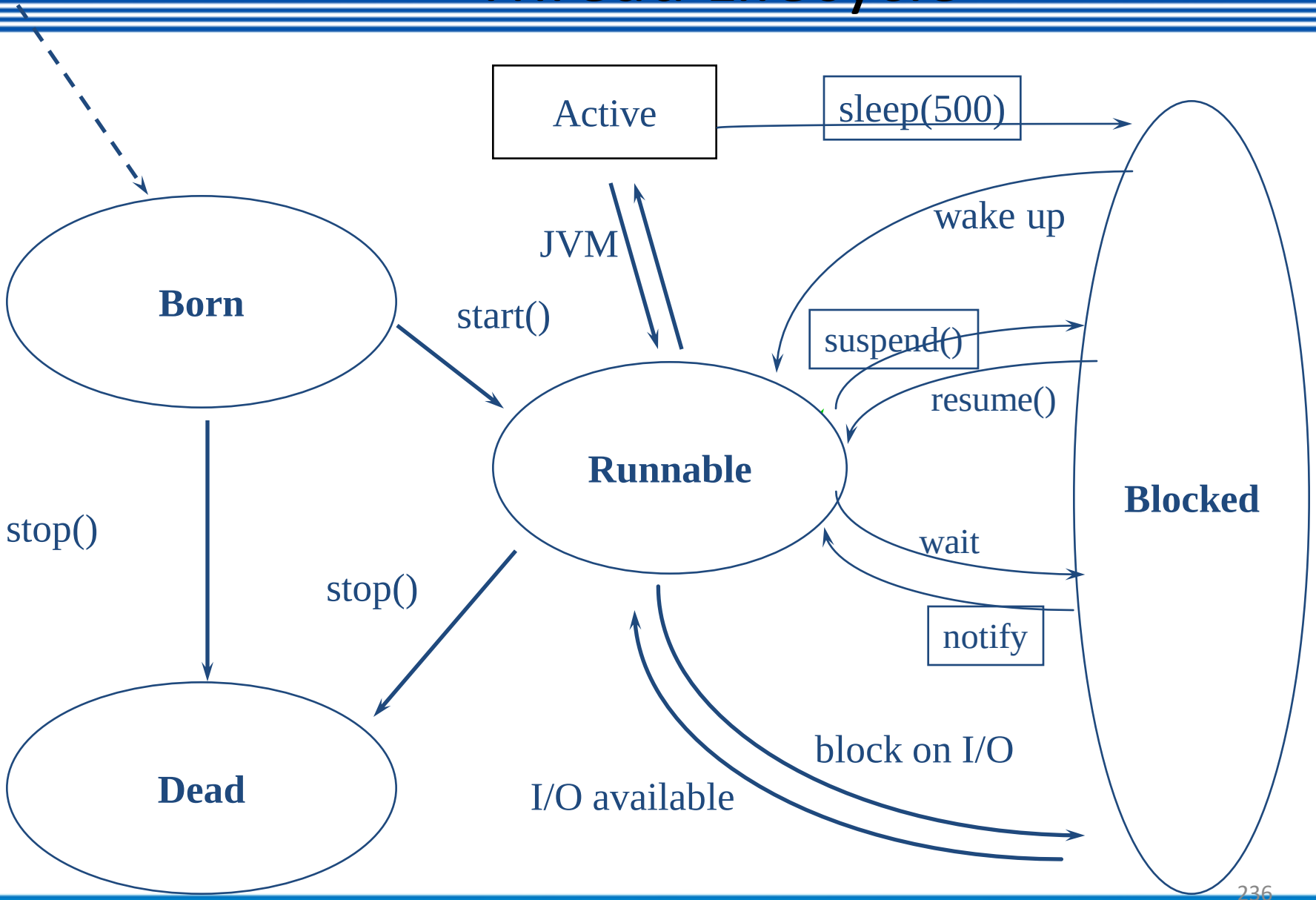
  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU

  3) of course, user input is much slower than the computer

Thread Lifecycle

- Thread exist in several states:
  1) ready to run
  2) running
  3) a running thread can be suspended
  4) a suspended thread can be resumed
  5) a thread can be blocked when waiting for a resource
  6) a thread can be terminated
- Once terminated, a thread cannot be resumed.

Thread Lifecycle



Thread Lifecycle

- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Creating Threads

- To create a new thread a program will:

  1) extend the Thread class, or

  2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

- Start: a thread by calling start its run method
- Sleep: suspend a thread for a period of time
- Run: entry-point for a thread
- Join: wait for a thread to terminate
- isAlive: determine if a thread is still running
- getPriority: obtain a thread's priority
- getName: obtain a thread's name

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:

  1) create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

  public void run()

  2) instantiate a Thread object within that class, a possible constructor is:

  Thread(Runnable threadOb, String threadName)

  3) call the start method on this object (start calls run):

  void start()

Example: New Thread 1

- A class NewThread that implements Runnable:

```
class NewThread implements Runnable {
Thread t;
//Creating and starting a new thread. Passing this to the
// Thread constructor – the new thread will call this
// object's run method:
NewThread() {
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
```

Example: New Thread 2

```
//This is the entry point for the newly created thread – a five-iterations loop
//with a half-second pause between the iterations all within try/catch:
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

Example: New Thread 3

```
class ThreadDemo {
public static void main(String args[]) {
//A new thread is created as an object of
// NewThread:
new NewThread();
//After calling the NewThread start method,
// control returns here.
```

Example: New Thread 4

```
//Both threads (new and main) continue concurrently.
//Here is the loop for the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

New Thread: Extend Thread

- The second way to create a new thread:

  1) create a new class that extends Thread

  2) create an instance of that class

- Thread provides both run and start methods:

  1) the extending class must override run

  2) it must also call the start method

Example: New Thread 1

- The new thread class extends Thread:

```
class NewThread extends Thread {
//Create a new thread by calling the Thread's
// constructor and start method:
NewThread() {
super("Demo Thread");
System.out.println("Child thread: " + this);
start();
}
```

Example: New Thread 2

NewThread overrides the Thread's run method:

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

Example: New Thread 3

```
class ExtendThread {
public static void main(String args[]) {
//After a new thread is created:
new NewThread();
//the new and main threads continue
//concurrently…
```

Example: New Thread 4

```
//This is the loop of the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.
- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:

  1) classes can define so-called synchronized methods

  2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called

  3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Synchronization

- Language keyword: synchronized
- Takes out a monitor lock on an object
  - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a “force field” around the locked object so no other threads can enter
  - Actually, it only blocks access to other synchronizing threads

Thread Priorities

- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Thread Priorities

Example of priority of a Thread:

```
1.  class TestMultiPriority1 extends Thread{
2.   public void run(){
3.     System.out.println("running thread name is:"+Thread.currentThread().getName());
4.     System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
6.    }
7.   public static void main(String args[]){
8.    TestMultiPriority1 m1=new TestMultiPriority1();
9.    TestMultiPriority1 m2=new TestMultiPriority1();
10.   m1.setPriority(Thread.MIN_PRIORITY);
11.   m2.setPriority(Thread.MAX_PRIORITY);
12.   m1.start();
13.   m2.start();
14.
15.  }
16. }
```

- Output:running thread name is:
- Thread-0 running thread priority is:10
- running thread name is:Thread-1
- running thread priority is:1

Inter thread Communication

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:
  - wait()
  - notify()
  - notifyAll()

1) wait() method

- Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

2) notify() method

- Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.
- Syntax: public final void notify()

3) notifyAll() method

- Wakes up all threads that are waiting on this object's monitor. Syntax:
- public final void notifyAll()

Daemon Threads

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- For example, you can start or suspend all the threads within a group with a single method call.
- Java thread groups are implemented by the “ThreadGroup” class in the java.lang package.
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created

The ThreadGroup Class

- The “ThreadGroup” class manages groups of threads for Java applications.
- A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- ThreadGroups can contain not only threads but also other ThreadGroups.
- The top-most thread group in a Java application is the thread group named main.
- You can create threads and thread groups in the main group.
- You can also create threads and thread groups in subgroups of main.

Creating a Thread Explicitly in a Group

- A thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread.
- The Thread class has three constructors that let you set a new thread's group:

  public Thread(ThreadGroup *group*, Runnable *target*) public Thread(ThreadGroup *group*, String *name*)

  public Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group.

  For example:

  ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");

  Thread myThread = new Thread(myThreadGroup, "a thread for my group");

UNIT-4
FILES AND CONNECTING TO DATABASE

Package java.io

- Provides for system input and output through data streams, serialization and the file system.

Interface Summary

- **DataInput** The Data Input interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.

- **DataOutput** The Data Output interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream

- **Externalizable** Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.

- **Serializable** Serializability of a class is enabled by the class implementing the java.io.Serializable interface.

Class Summary

- **BufferedInputStream:** A BufferedInputStream adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset methods.
- **BufferedOutputStream:** The class implements a buffered output stream.
- **BufferedReader:** Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- **BufferedWriter:** Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings
- **ByteArrayInputStream:** A ByteArrayInputStream contains an internal buffer that contains bytes that may be read from the stream.
- **ByteArrayOutputStream:** This class implements an output stream in which the data is written into a byte array.

- **CharArrayReader:** This class implements a character buffer that can be used as a character-input stream

- .**CharArrayWriter:** This class implements a character buffer that can be used as an Writer

- **Console:** Methods to access the character-based console device, if any, associated with the current Java virtual machine.

- **DataInputStream:** A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way.

- **DataOutputStream:** A data output stream lets an application write primitive Java data types to an output stream in a portable way.

- **File:** An abstract representation of file and directory pathnames.
- **FileInputStream:** A FileInputStream obtains input bytes from a file in a file system.
- **FileOutputStream:** A file output stream is an output stream for writing data to a File or to a FileDescriptor.
- **FileReader:** Convenience class for reading character files.
- **FileWriter:** Convenience class for writing character files.
- **FilterInputStream:** A FilterInputStream contains some other input stream, which it uses as its basic source of data, possibly transforming the data along the way or providing additional functionality.
- **FilterOutputStream:** This class is the superclass of all classes that filter output streams
- .**FilterReader:** Abstract class for reading filtered character streams
- .**FilterWriter:** Abstract class for writing filtered character streams
- .**InputStream:** This abstract class is the superclass of all classes representing an input stream of bytes.
- **InputStreamReader:** An InputStreamReader is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset.

- **ObjectInputStream:** An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream
- **ObjectOutputStream:** An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream.
- **OutputStream:** This abstract class is the superclass of all classes representing an output stream of bytes.
- **OutputStreamWriter:** An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.
- **PrintWriter:** Prints formatted representations of objects to a text-output stream.
- **RandomAccessFile:** Instances of this class support both reading and writing to a random access file.
- **StreamTokenizer:** The StreamTokenizer class takes an input stream and parses it into "tokens", allowing the tokens to be read one at a time.

JDBC connectivity

Introduction

- JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

- The JDBC library includes APIs for each of the tasks commonly associated with database usage:

➢ Making a connection to a database

➢ Creating SQL or MySQL statements

➢ Executing that SQL or MySQL queries in the database

➢ Viewing & Modifying the resulting records

Required Steps:

- There are following steps required to create a new Database using JDBC application:
- **Import the packages .** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver .** Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection .** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with database server.
- To create a new database, you need not to give any database name while preparing database URL as mentioned in the below example.
- **Execute a query .** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Clean up the environment .** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Creating JDBC Application, connecting a Database

- There are six steps involved in building a JDBC application which I'm going to brief in this tutorial:

1.Import the packages:

- To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code:
- //STEP 1. Import required packages
- Syntax :import java.sql.*;

2.Register the JDBC driver:

- This requires that you initialize a driver so you can open a communications channel with the database.
- Registering the driver is the process by which the Oracle driver's class file is loaded into memory so it can be utilized as an implementation of the JDBC interfaces.
- You need to do this registration only once in your program
- //STEP 2: Register JDBC driver
- Syntax:Class.forName("com.mysql.jdbc.Driver");

Open a connection:

- After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method, which represents a physical connection with the database as follows:
- //STEP 3: Open a connection // Database credentials
  - static final String USER = "username";
  - static final String PASS = "password";
  - System.out.println("Connecting to database...");
  - conn = DriverManager.getConnection(DB_URL,USER,PASS);

Execute a query:

- This requires using an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database as follows:
- //STEP 4: Execute a query
  - System.out.println("Creating statement...");
  - stmt = conn.createStatement();
  - String sql; sql = "SELECT id, first, last, age FROM Employees";
  - ResultSet rs = stmt.executeQuery(sql);

- Following table lists down popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

- All the highlighted part in URL format is static and you need to change only remaining part as per your database setup.

- If there is an SQL UPDATE,INSERT or DELETE statement required, then following code snippet would be required:
- //STEP 4: Execute a query

- System.out.println("Creating statement...");
- stmt = conn.createStatement();
- String sql;
- sql = "DELETE FROM Employees";
- ResultSet rs = stmt.executeUpdate(sql);

Extract data from result set:

- This step is required in case you are fetching data from the database. You can use the appropriate ResultSet.getXXX() method to retrieve the data from the result set as follows:
- //STEP 5: Extract data from result set

```
while(rs.next())
{
//Retrieve by column name
int id = rs.getInt("id");
 int age = rs.getInt("age");
 String first = rs.getString("first");
String last = rs.getString("last");
//Display values
System.out.print("ID: " + id);
 System.out.print(", Age: " + age);
System.out.print(", First: " + first);
System.out.println(", Last: " + last); }
```

- **Clean up the environment:**
- You should explicitly close all database resources versus relying on the JVM's garbage collection as follows:

- //STEP 6: Clean-up environment

- rs.close();
- stmt.close();
- conn.close();

JDBC Driver

- JDBC drivers implement the defined interfaces in the JDBC API for interacting with your database server.
- For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.
- The *Java.sql* package that ships with JDK contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers.
- Third party vendors implements the *java.sql.Driver* interface in their database driver.

JDBC Drivers Types:

- JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4

Type 1: JDBC-ODBC Bridge Driver

- In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine.
- Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.
- When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.
- The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 1: JDBC-ODBC Bridge Driver

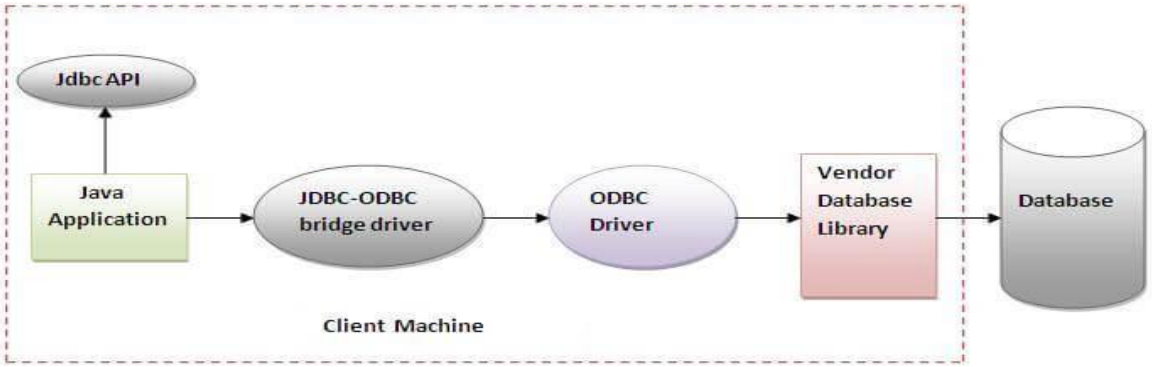


Figure- JDBC-ODBC Bridge Driver

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Type 2: JDBC-Native API

- In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database.
- These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.
- If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.
- The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 2: JDBC-Native API

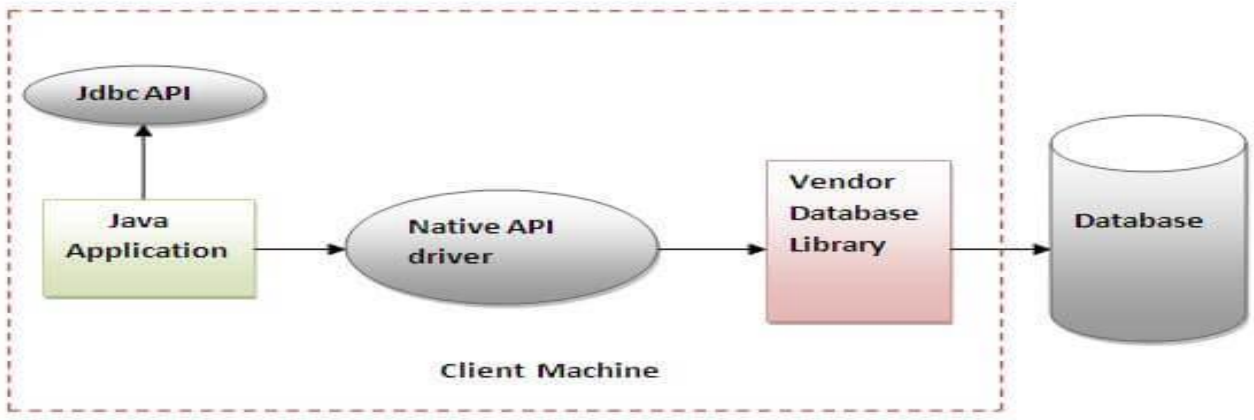


Figure- Native API Driver

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Type 3: JDBC-Net pure Java/ Network Protocol Driver

- In a Type 3 driver, a three-tier approach is used to accessing databases.
- The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.
- This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases

Type 3: JDBC-Net pure Java/ Network Protocol Driver

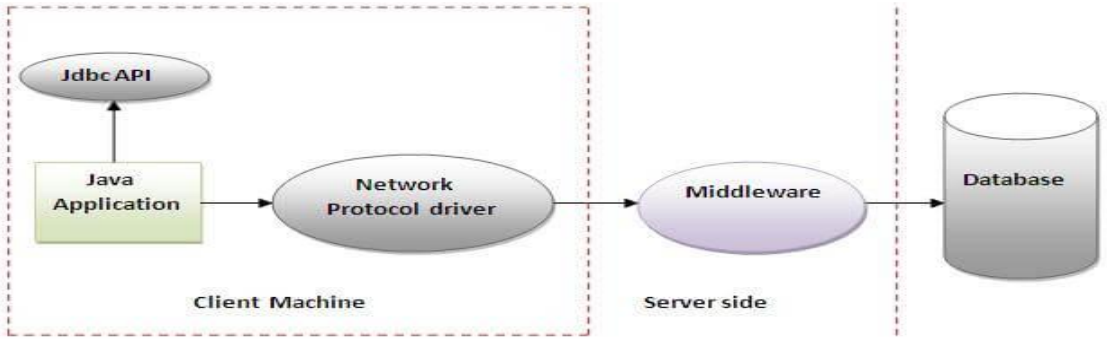


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Type 4: 100% pure Java / thin driver

- In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection.
- This is the highest performance driver available for the database and is usually provided by the vendor itself.
- This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.
- MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Type 4: 100% pure Java / thin driver

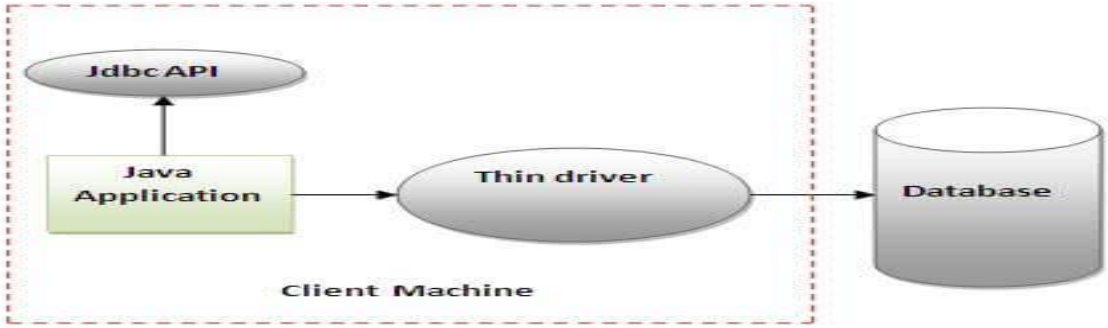


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver Type is 4.
- If your Java application is accessing multiple types of databases at the same time, Type 3 is the preferred driver.
- Type 2 drivers are useful in situations where a type 3 or Type 4 driver is not available yet for your database.
- The Type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

Java Database Connectivity

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

Java Database Connectivity

1) Register the driver class

- The **forName()** method of Class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

- **public static void** forName(String className)**throws** ClassNotFoundException

Example to register the OracleDriver class

- Here, Java program is loading oracle driver to establish database connection.
- Class.forName("oracle.jdbc.driver.OracleDriver");

Java Database Connectivity

2) Create the connection object

- The **getConnection()** method of DriverManager class is used to establish connection with the database.
- Syntax of getConnection() method

1) **public static** Connection getConnection(String url)**throws** SQLException

2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

Example to establish connection with the Oracle database

- Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","password");

Java Database Connectivity

3) Create the Statement object

- The createStatement() method of Connection interface is used to create statement. The object of
- statement is responsible to execute queries with the database.

Syntax of createStatement() method

- **public** Statement createStatement()**throws** SQLException

Example to create the statement object

- Statement stmt=con.createStatement();

Java Database Connectivity

4) Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database.
- This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

- **public** ResultSet executeQuery(String sql)**throws** SQLException

Example to execute query

- ResultSet rs=stmt.executeQuery("select * from emp");
- **while**(rs.next()){
- System.out.println(rs.getInt(1)+" "+rs.getString(2));
- }

Java Database Connectivity

5) Close the connection object

- By closing connection object statement and ResultSet will be closed automatically.
- The close() method of Connection interface is used to close the connection.

Syntax of close() method

- **public void** close()**throws** SQLException

Example to close connection

- con.close();

Querying a Database and Processing Result

In general, to process any SQL statement with JDBC, you follow these steps:

- Establishing a connection.
- Create a statement.
- Execute the query.
- Process the result set object
- Close the connection.

Querying a Database and Processing Result

Establishing Connections

- First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. This connection is represented by a Connection object.

Creating Statements

- A Statement is an interface that represents a SQL statement. You execute Statement objects, and they generate ResultSet objects, which is a table of data representing a database result set. You need a Connection object to create a Statement object.
- For example, CoffeesTables.viewTable creates a Statement object with the following code:
- stmt = con.createStatement();

Querying a Database and Processing Result

There are three different kinds of statements:

- Statement: Used to implement simple SQL statements with no parameters.
- PreparedStatement: (Extends Statement.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.
- CallableStatement: (Extends PreparedStatement.) Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

Querying a Database and Processing Result

Executing Queries

- To execute a query, call an execute method from Statement such as the following:
- execute: Returns true if the first object that the query returns is a ResultSet object. Use this method if the query could return one or more ResultSet objects. Retrieve the ResultSet objects returned from the query by repeatedly calling Statement.getResultSet.
- executeQuery: Returns one ResultSet object.
- executeUpdate: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using INSERT, DELETE, or UPDATE SQL statements.
- For example, CoffeesTables.viewTable executed a Statement object with the following code:
- ResultSet rs = stmt.executeQuery(query); See Retrieving and Modifying Values from Result Sets for more information.

Querying a Database and Processing Result

Processing ResultSet Objects

- access the data in a ResultSet object through a cursor. Note that this cursor is not a database cursor.
- This cursor is a pointer that points to one row of data in the ResultSet object. Initially, the cursor is positioned before the first row. You call various methods defined in the ResultSet object to move the cursor.

Closing Connections

- When you are finished using a Statement, call the method Statement.close to immediately release the resources it is using. When you call this method, its ResultSet objects are closed.
- For example, the method CoffeesTables.viewTable ensures that the Statement object is closed at the end of the method, regardless of any SQLException objects thrown, by wrapping it in a finally block:
- } finally { if (stmt != null) { stmt.close(); }}

Updating Data With JDBC:

The following steps are required to create a new Database using JDBC application

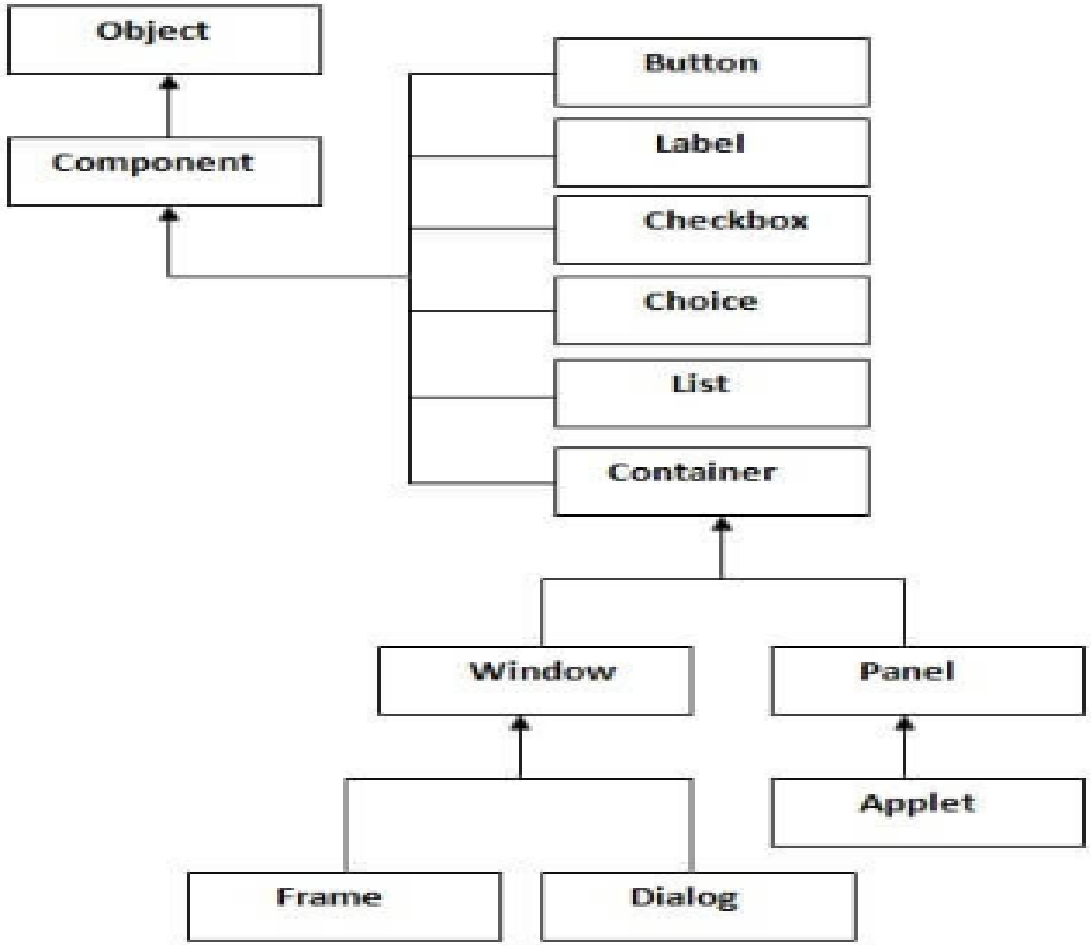
- **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver:** Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with a database server.
- **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to update records in a table. This Query makes use of **IN** and **WHERE** clause to update conditional records.
- **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

UNIT-5
GUI PROGRAMMING AND APPLETS

The AWT CLASS HIERARCHY

- **Java AWT** (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The AWT CLASS HIERARCHY



The AWT CLASS HIERARCHY

- The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. some of the AWT classes.

AWT Classes

- **Component –** It is an object that is represented in graphical form and then displayed n the users screen.
- **Container –** It consists of AWT components
- **Panel -** It is a type of container class
- **Window -** It is a top-level window without any borders and members.
- **Dialog -** It is a top-level window containing a title and border.
- **Frame -** It is a top-level window containing a title and border. It can also contain other components

The AWT CLASS HIERARCHY

- **Text Component -** It is the base of TextField and TextArea.
- **TextArea -** It is an object in which a user can enter or modify the multi line input
- **TextField -** It is a text area which a user can enter or modify single line input.
- **Button -** It is used to create a button.
- **Canvas -** It is a blank rectangular area where a user can draw different shapes or can even trap the input events generated user.
- **Checkbox -** It is a component that can be in either ‘on’ or ‘off’ state at certain point of time.
- **Choice –** It opens a pop-up menu containing choices.
- **Label -** It places the text in container.
- **List -** It is a list containing string items. It is scrollable.
- **Scrollbar -** It contains a vertical or horizontal scrollbar.

The AWT CLASS HIERARCHY

AWT Example by Inheritance

- Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.
- **import** .*;
- **class** First **extends** Frame{
- First(){
- Button b=**new** Button("click me");
- b.setBounds(30,100,80,30);// setting button position
- add(b);//adding button into frame
- setSize(300,300);//frame size 300 width and 300 height
- setLayout(**null**);//no layout manager
- setVisible(**true**);//now frame will be visible, by default not visible
- }
- **public static void** main(String args[]){
- First f=**new** First();
- }}

INTRODUCTION TO SWING

- Swing API is a set of extensible GUI Components to ease the developer's life to create JAVA based Front End/GUI Applications. It is build on top of AWT API and acts as a replacement of AWT API, since it has almost every control corresponding to AWT controls. Swing component follows a Model-View-Controller architecture to fulfill the following criterias.
- A single API is to be sufficient to support multiple look and feel.
- API is to be model driven so that the highest level API is not required to have data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers for use.

INTRODUCTION TO SWING

MVC Architecture

- Swing API architecture follows loosely based MVC architecture in the following manner.
- Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.
- Swing component has Model as a seperate element, while the View and Controller part are clubbed in the User Interface elements. Because of which, Swing has a pluggable look-and-feel architecture.

INTRODUCTION TO SWING

Swing Features

- **Light Weight** – Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich Controls** – Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.
- **Highly Customizable** – Swing controls can be customized in a very easy way as visual apperance is independent of internal representation.
- **Pluggable look-and-feel** – SWING based GUI Application look and feel can be changed at run-time, based on available values.

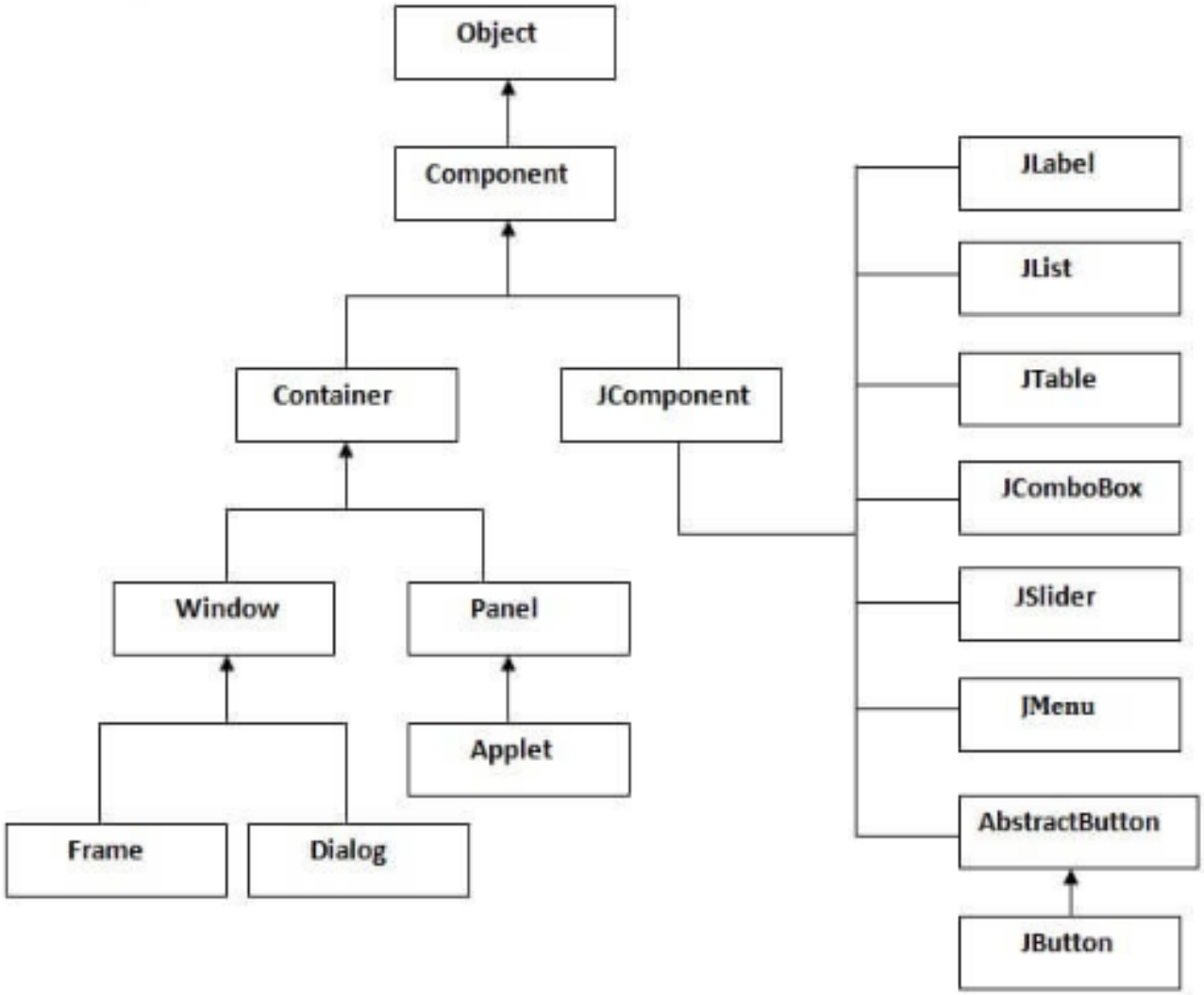
SWING VS AWT

Differentiation between Swing and AWT	
Swing	**AWT**
Swing is used for building highly interactive GUI applications	AWT is used for building GUI applications.
The component of SWING are light weight	The components of AWT are heavy weight
The look and feel is not based on OS	The look and fees is based on OS
It is purely based on Java	It is not purely based on Java
It is slower in performance	It is faster in performance
The features like icons and tool-tips are supported	The feature like icons and tool-tips are not supported
A Plug-in is needed for applets	The web browser supports applets
Platform imposes very less limitations for components	Platform imposes many limitations for components
BorderLayout is the default layout for content Pane	FlowLayout and BorderLayout are default layout for applet and Frame respectively

HIERARCHY FOR SWING COMPONENTS

- Swing components are enhanced components compared to that of AWT.
- They are even driven and provide a good programming approach through OOP concepts. They are light weight components which makes use of model-view controller architecture.
- All the components of swing are contained in javax.swing package.
- Unlike AWT, Java Swing provides platform-independent and lightweight components.
- The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

HIERARCHY FOR SWING COMPONENTS



HIERARCHY FOR SWING COMPONENTS

- The hierarchy of swing components consists of following elements.

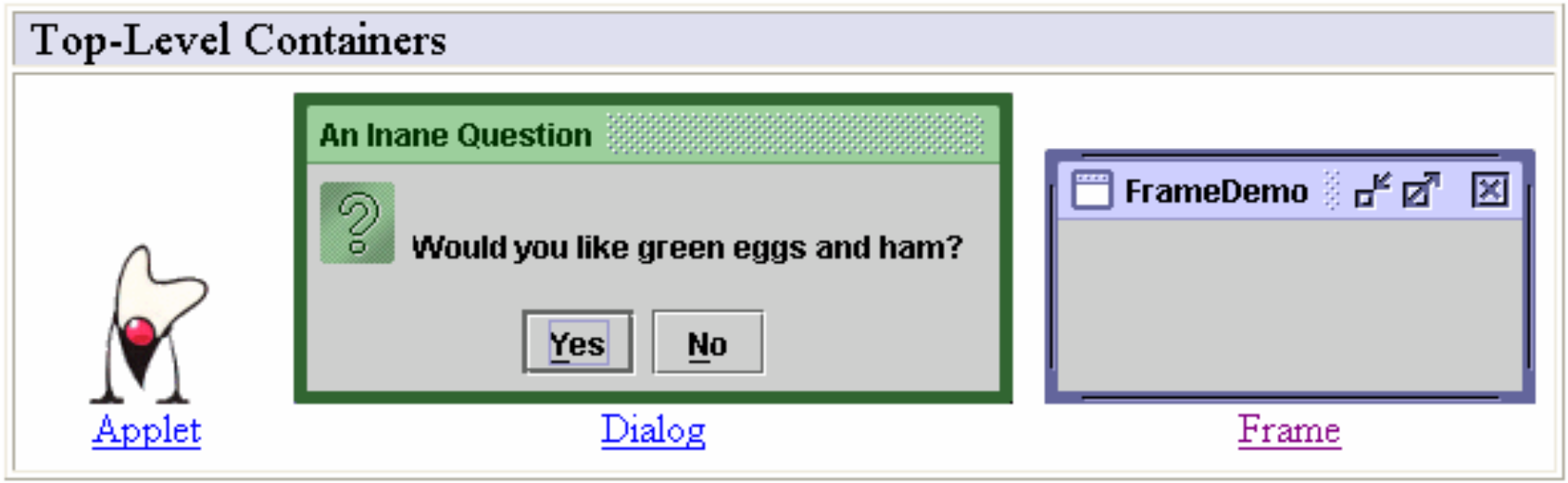
Component	It is a visual control which is independent
Container	It is a type of component that contains other components
JComponent	It is a class that holds all the other components
JPanel	It is a light weight swing container. It is a component that can be added to contentPane of Jframe
Jlabel	It is a easiet-to-use component that is used to create a component
JTextComponent	It is the base class for JtextField and JTextArea
JTextField	It allows the user to enter or edit single line of text
JTextArea	It allows the user to enter or edit multiple line of text
Abstract Button	It is the base clas of Jbutton and JMenuItem
JButton	It is a push button associated with icon, a string or both.
JMenuItem	It represents an list of item in the form of menu

CONTAINERS- JFRAME, JAPPLET

- A container is a component which it can contain other components inside itself. It is also an instance of a subclass of java.awt.Container.
- java.awt.Container extends java.awt.Component so containers are themselves components.
- In general components are contained in a container. An applet is a container. Other containers include windows, frames, dialogs, and panels. Containers may contain other containers.
- Every container has a LayoutManager that determines how different components are positioned within the container.
- Applets provide a ready-made container and a default LayoutManager, a FlowLayout.

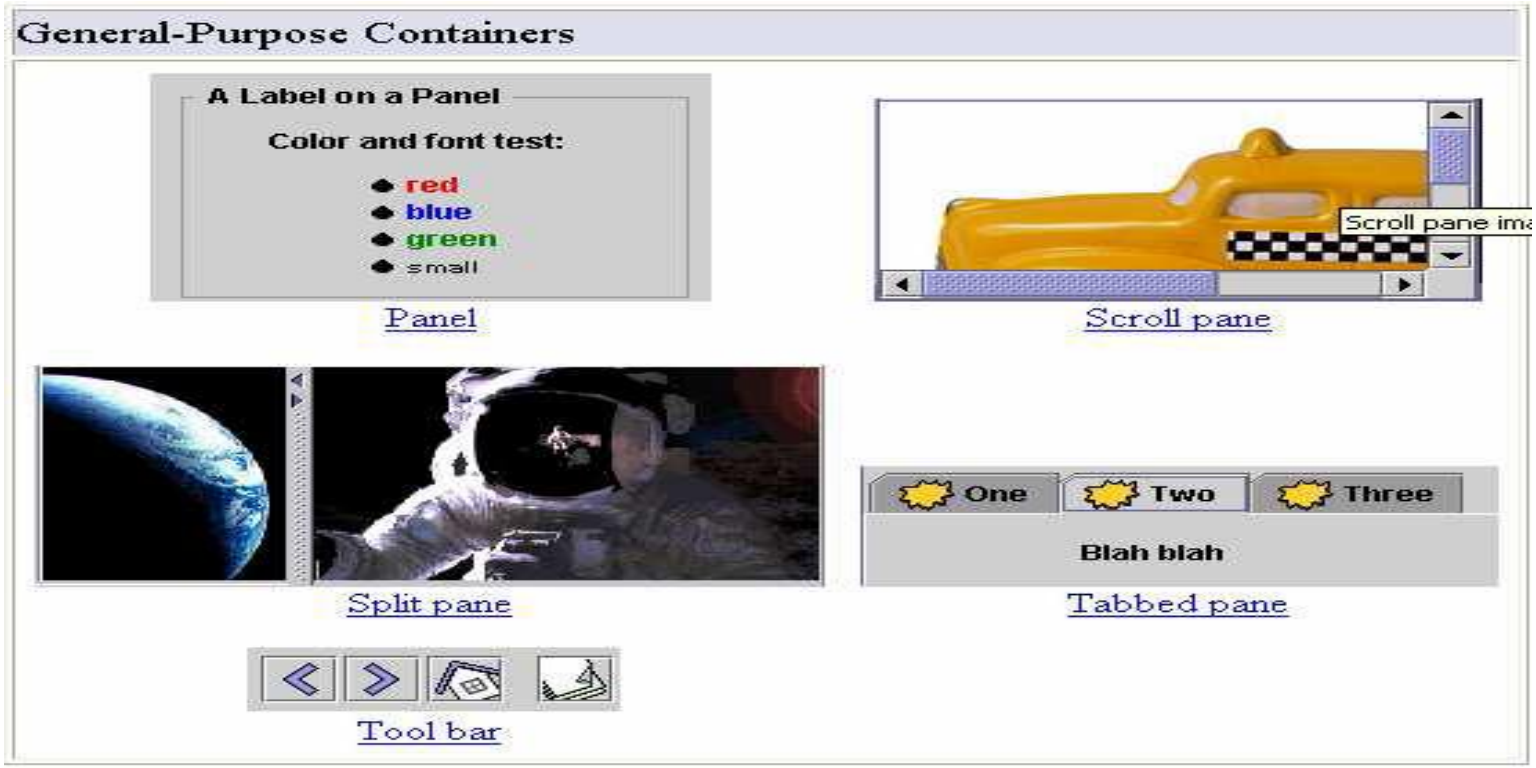
Containers

- Top-Level Containers
- The components at the top of any Swing containment hierarchy



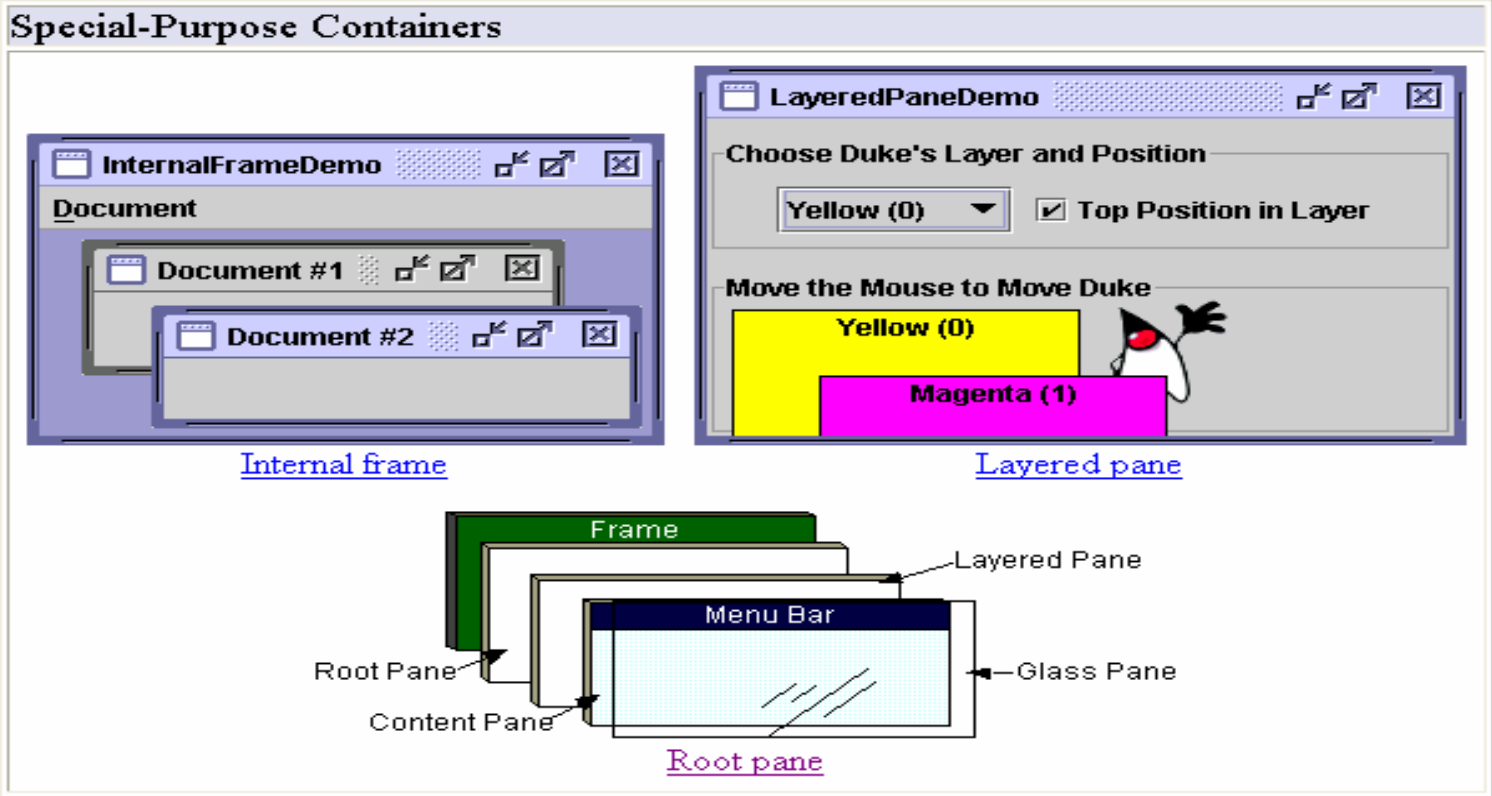
General Purpose Containers

- Intermediate containers that can be used under many different circumstances.



Special Purpose Container

- Intermediate containers that play specific roles in the UI.



CONTAINERS- JFRAME, JAPPLET

ABOUT JFrame

- JFrame is used for creating new windows called Jframe objects.
- They are displayed on the screen and are intractable with the users.
- It is similar to that of AWT where Frames user interact with these frames through buttons.
- A frame consists of a title which is a string ad a menu bar.
- Several menus can be added using menu bar.
- JFrame receives events from windows. For example window closing will generate an event. It provides several methods inorder to control the attributes of the window.
- JFrame works like the main window where components like Labels, Buttons, Textfields are added to create a GUI.
- Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

CONTAINERS – JFRAME, JAPPLET

Example program for JFrame

```
import javax.swing.*;
class Frame
{
   public static void main (String args[])
   {
         JFrame f= new Jframe (“Demo”);
         f.setSize(100, 50);
         f.setVisible(true);
   }
}
```

CONTAINERS – JFRAME, JAPPLET

ABOUT JApplet

- JApplet is a swing container that is used for creating applets
- It is similar to applet of AWT.
- It allows the users to add components of it.
- The methods of it are init(), start(), stop() and destroy()
- The JApplet class extends the Applet class.
- If using Swing components in an applet, subclass *JApplet*, not *Applet*
  - *JApplet* is a subclass of *Applet*
  - Sets up special internal component event handling, among other things
  - Can have a *JMenuBar*
  - Default *LayoutManager* is *BorderLayout*

CONTAINERS – JFRAME, JAPPLET

Example program for JApplet

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener
{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
```

CONTAINERS – JFRAME, JAPPLET

```
add(b);add(tf);
b.addActionListener(this);
setLayout(null);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
} }
```

myapplet.html

```
<html>
<body>
<applet code="EventJApplet.class" width="300" height="300">
</applet>
</body>
</html>
```

CONTAINERS – JDialog, JPanel

About JDialog

- JDialog is a container that is used to create and manage a dialog.'
- It inherits the properties of container, component, window and Dialog of AWT and allow user to create model or modeless dialog.
- A modal dialog pauses the application until the dialog is closed.
- Whereas the modeless dialog activates the remaining part of the application.
- A dialog can be created with the below statement

JDialog (Frame owner, String title, boolean is Modal)

CONTAINERS – JDialog, JPanel

Example program on JDialog

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solve extends JFrame implements ActionListener {
  static JFrame f;
     public static void main(String[] args)
   {
        f = new JFrame("frame");
       solve s = new solve();
        JPanel p = new JPanel();
        JButton b = new JButton("click");
         b.addActionListener(s);
       p.add(b);
        f.add(p);
```

CONTAINERS – JDialog, JPanel

```
        f.setSize(400, 400);
         f.show();
    }
    public void actionPerformed(ActionEvent e)
    {
       String s = e.getActionCommand();
       if (s.equals("click")) {
             JDialog d = new JDialog(f, "dialog Box");
           JLabel l = new JLabel("this is a dialog box");
           d.add(l);
           d.setSize(100, 100);
           d.setVisible(true);
       }
    }
}
```

CONTAINERS – JDialog, JPanel

About JPanel

- JPanel, a part of Java Swing package, is a container that can store a group of components.
- The main task of JPanel is to organize components, various layouts can be set in JPanel which provide better organization of components, however it does not have a title bar.

Constructor of JPanel are :

- **JPanel()** : creates a new panel with flow layout
- **JPanel(LayoutManager l)** : creates a new JPanel with specified layoutManager
- **JPanel(boolean isDoubleBuffered)** : creates a new JPanel with a specified buffering strategy
- **JPanel(LayoutManager l, boolean isDoubleBuffered)** : creates a new JPanel with specified layoutManager and a specified buffering strategy

CONTAINERS – JDialog, JPanel

Example program of Jpanel

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solution extends JFrame {
    static JFrame f;
    static JButton b, b1, b2;
    static JLabel l;
    public static void main(String[] args) {
   f = new JFrame("panel");
   l = new JLabel("panel label");
   b = new JButton("button1");
```

CONTAINERS – JDialog, JPanel

Example program of Jpanel

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class solution extends JFrame {
    static JFrame f;
    static JButton b, b1, b2;
    static JLabel l;
    public static void main(String[] args) {
   f = new JFrame("panel");
   l = new JLabel("panel label");
```

CONTAINERS – JDialog, JPanel

```
        b = new JButton("button1");
        b1 = new JButton("button2");
        b2 = new JButton("button3");
       JPanel p = new JPanel();
        p.add(b);
        p.add(b1);
        p.add(b2);
        p.add(l);
          p.setBackground(Color.red);
        f.add(p);
        f.setSize(300, 300);
        f.show();
    } }
```

GridBag Layout

- The Java GridBagLayout class is used to align components vertically, horizontally or along their baseline.
- The components may not be of same size. Each GridBagLayout object maintains a dynamic, rectangular grid of cells.
- Each component occupies one or more cells known as its display area. Each component associates an instance of GridBagConstraints.
- With the help of constraints object we arrange component's display area on the grid. The GridBagLayout manages each component's minimum and preferred sizes in order to determine component's size.
- The constructors are

  GridLayout( )

  GridLayout(int numRows, int numColumns)

  GridLayout(int numRows, int numColumns, int horz, int vert)

Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.
- AWT component is converted by the native code of the operating system.

Components

- Lowest Common Denominator
  - If not available natively on one Java platform, not available on any Java platform
- Simple Component Set
- Components Peer-Based
  - Platform controls component appearance
  - Inconsistencies in implementations
  - Interfacing to native platform error-prone

Components

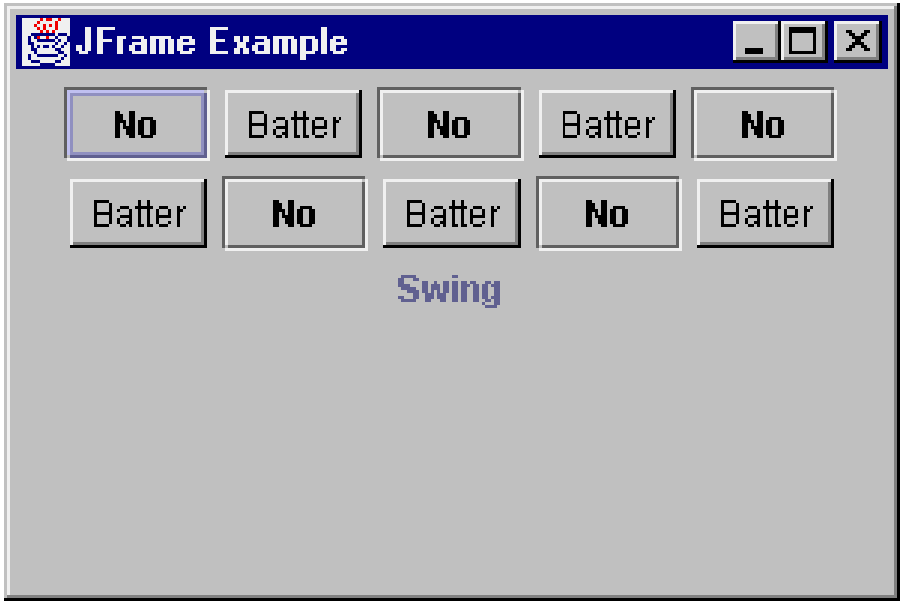
- Container
  - JComponent
    - AbstractButton
      - JButton
      - JMenuItem
        - » JCheckBoxMenuItem
        - » JMenu
        - » JRadioButtonMenuItem
      - JToggleButton
        - » JCheckBox
        - » JRadioButton

- JComponent
  - JComboBox
  - JLabel
  - JList
  - JMenuBar
  - JPanel
  - JPopupMenu
  - JScrollBar
  - JScrollPane

- JComponent
  - JTextComponent
    - JTextArea
    - JTextField
      - JPasswordField
    - JTextPane
      - JHTMLPane

JFrame

```
public class FrameTest {
  public static void main (String args[]) {
    JFrame f = new JFrame ("JFrame Example");
    Container c = f.getContentPane();
    c.setLayout (new FlowLayout());
    for (int i = 0; i < 5; i++) {
      c.add (new JButton ("No"));
      c.add (new Button ("Batter"));
    }
    c.add (new JLabel ("Swing"));
    f.setSize (300, 200);
    f.show();
  }
}
```



JComponent

- JComponent supports the following components.
- JComponent
  - JComboBox
  - JLabel
  - JList
  - JMenuBar
  - JPanel
  - JPopupMenu
  - JScrollBar
  - JScrollPane
  - JTextComponent
    - JTextArea
    - JTextField
      - JPasswordField
    - JTextPane
      - JHTMLPane

- Swing labels are instances of the **JLabel** class, which extends **JComponent**.
- It can display text and/or an icon.
- Constructors are:

    1. JLabel(Icon i) 2. JLabel(String s) 3. JLabel(String s, Icon i, int align)

- Here, *s* and *i* are the text and icon used for the label. The *align* argument is either **LEFT**, **RIGHT**, or **CENTER**. These constants are defined in the **SwingConstants** interface,
- Methods are:

    1. Icon getIcon( )
    2. String getText( )
    3. void setIcon(Icon i)
    4. void setText(String s)

- Here, *i* and *s* are the icon and text, respectively.

Buttons

- Swing buttons provide features that are not found in the **Button** class defined by the AWT.
- Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**.
- **AbstractButton** contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.
- Methods are:
    1. void setDisabledIcon(Icon di)
    2. void setPressedIcon(Icon pi)
    3. void setSelectedIcon(Icon si)
    4. void setRolloverIcon(Icon ri)
- Here, *di*, *pi*, *si*, and *ri* are the icons to be used for these different conditions.
- The text associated with a button can be read and written via the following methods:
    1. String getText( ) 2. void setText(String s)
- Here, *s* is the text to be associated with the button.

JButton

- The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.
- The **JButton** class provides the functionality of a push button.
- **JButton** allows an icon, a string, or both to be associated with the push button.
- Some of its constructors are :

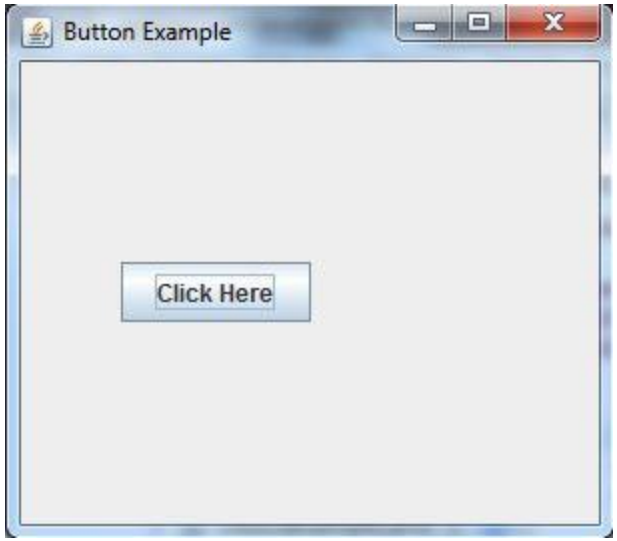
  JButton(Icon i)

  JButton(String s)

  JButton(String s, Icon i)
- Here, *s* and *i* are the string and icon used for the button.

JButton

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
    JFrame f=new JFrame("Button Example");
    JButton b=new JButton("Click Here");
    b.setBounds(50,100,95,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```



JLabel

- The object of JLabel class is a component for placing text in a container.
- It is used to display a single line of read only text.
- The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

Commonly used Constructors:

JLabel(): Creates a JLabel instance with no image and with an empty string for the title.

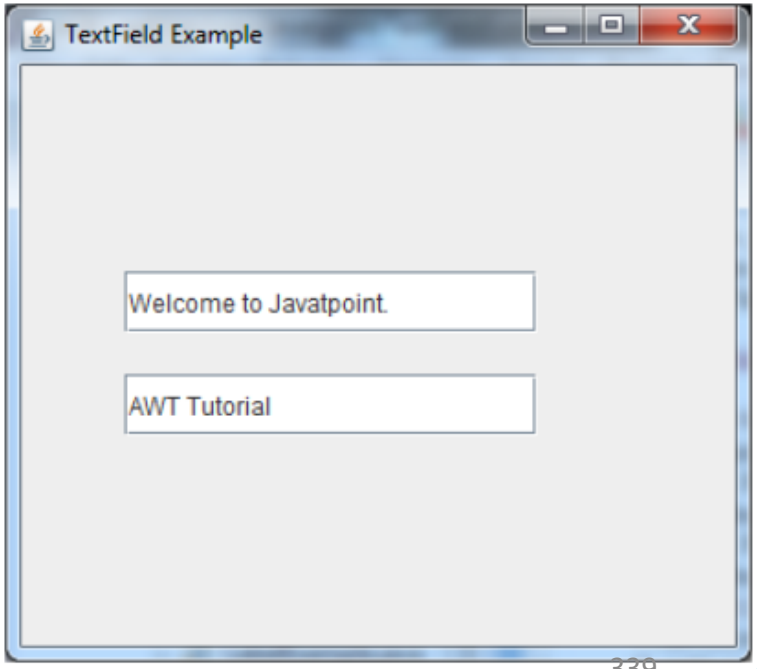
JLabel(String s): Creates a JLabel instance with the specified text.

JLabel(Icon i): Creates a JLabel instance with the specified image.

JLabel(String s, Icon i, int horizontalAlignment): Creates a JLabel instance with the specified text, image, and horizontal alignment.

JLabel

```
import javax.swing.*;
class TextFieldExample  {
public static void main(String args[])  {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    } }
```



JTextField

- JTextField is a part of javax.swing package. The class JTextField is a component that allows editing of a single line of text.
- JTextField inherits the JTextComponent class and uses the interface SwingConstants.

The constructor of the class are :

- **JTextField()** : constructor that creates a new TextField
- **JTextField(int columns)** : constructor that creates a new empty TextField with specified number of columns.
- **JTextField(String text)** : constructor that creates a new empty text field initialized with the given string.
- **JTextField(String text, int columns)** : constructor that creates a new empty textField with the given string and a specified number of columns .
- **JTextField(Document doc, String text, int columns)** : constructor that creates a textfield that uses the given text storage model and the given number of columns.

JTextField

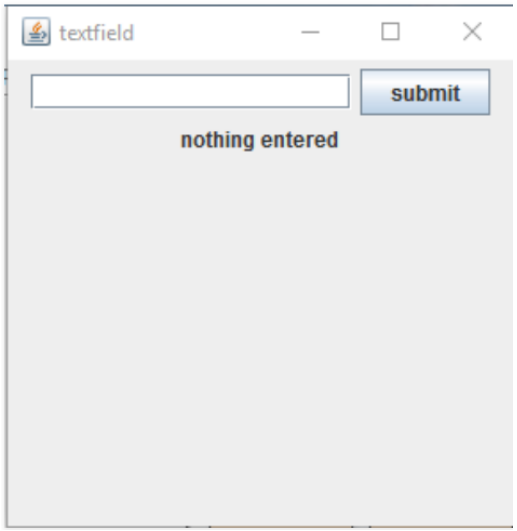
Java program to create a blank text field of definite number of columns.

```
import java.awt.event.*;
import javax.swing.*;
class text extends JFrame implements ActionListener {
    static JTextField t;
    static JFrame f;
    static JButton b;
   static JLabel l;
    text()
    { }
```

JTextField

```
public static void main(String[] args) {
f = new JFrame("textfield");
l = new JLabel("nothing entered");
b = new JButton("submit");
text te = new text();  b.addActionListener(te);
  t = new JTextField(16); JPanel p = new JPanel();
  p.add(t);  p.add(b); p.add(l);  f.add(p); f.setSize(300, 300);
  f.show(); }
public void actionPerformed(ActionEvent e)
{
   String s = e.getActionCommand();
   if (s.equals("submit")) {  l.setText(t.getText());
      t.setText("  ");
   } }}
```

output:



JTextArea

- JTextArea is a part of java Swing package . It represents a multi line area that displays text. It is used to edit the text.
- JTextArea inherits JComponent class. The text in JTextArea can be set to different available fonts and can be appended to new text . A text area can be customized to the need of user.

Constructors of JTextArea are:

- **JTextArea() :** constructs a new blank text area .
- **JTextArea(String s) :** constructs a new text area with a given initial text.
- **JTextArea(int row, int column) :** constructs a new text area with a given number of rows and columns.
- **JTextArea(String s, int row, int column) :** constructs a new text area with a given number of rows and columns and a given initial text.

JTextArea

```
// Java Program to create a simple JTextArea
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class text extends JFrame implements ActionListener {
    static JFrame f; static JButton b; static JLabel l;  static JTextArea jt;
    text()
    { }
    public static void main(String[] args)
    {
        f = new JFrame("textfield");
        l = new JLabel("nothing entered");
        b = new JButton("submit");
        text te = new text();
        b.addActionListener(te);
```

JTextArea

```
        jt = new JTextArea(10, 10);
        JPanel p = new JPanel();
        p.add(jt);
        p.add(b);
        p.add(l);
        f.add(p);
        f.setSize(300, 300);
        f.show();
    }
    public void actionPerformed(ActionEvent e) {
      String s = e.getActionCommand();
        if (s.equals("submit")) {
          l.setText(jt.getText());
        } } }
```

Layout Manager

- The Layout Managers are used to arrange components in a particular manner.
- LayoutManager is an interface that is implemented by all the classes of layout managers.
- it is very tedious to manually lay out a large number of components and sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.
- Each **Container** object has a layout manager associated with it.
- A layout manager is an instance of any class that implements the **LayoutManager** interface.
- The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used.
- Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

Layout manager Types

Layout manager class defines the following types of layout managers

- BoarderLayout
- GridLayout
- FlowLayout
- CardLayout
- GridBagLayout

Boarder layout

- The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred to as north, south, east, and west. The middle area is called the center.
- The constructors defined by **BorderLayout**:

  (a) BorderLayout( ) (b) BorderLayout(int horz, int vert)
- **BorderLayout** defines the following constants that specify the regions:

  public static final int NORTH

  public static final int SOUTH

  public static final int EAST

  public static final int WEST

  public static final int CENTER

Components can be added by

void add(Component compObj, Object region);

Boarder layout

```
import java.awt.*;
import javax.swing.*;
public class Border {
JFrame f;
Border(){
   f=new JFrame();
 JButton b1=new JButton("NORTH");;
 JButton b2=new JButton("SOUTH");;
JButton b3=new JButton("EAST");;
 JButton b4=new JButton("WEST");;
 JButton b5=new JButton("CENTER");;
 f.add(b1,BorderLayout.NORTH);
 f.add(b2,BorderLayout.SOUTH);
f.add(b3,BorderLayout.EAST);
f.add(b4,BorderLayout.WEST);
f.add(b5,BorderLayout.CENTER);
f.setSize(300,300);
f.setVisible(true);  }
public static void main(String[] args) {
    new Border();
}
}
```

Grid layout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, define the number of rows and columns.
- The constructors supported by GridLayout is:

  GridLayout( )

  GridLayout(int numRows, int numColumns)

  GridLayout(int numRows, int numColumns, int horz, int vert)
- The first form creates a single-column grid layout.
- The second form creates a grid layout
- with the specified number of rows and columns.
- The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.
- Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-lengthrows.

Grid layout

```
import java.awt.*;
import javax.swing.*;
public class MyGridLayout{
JFrame f;
MyGridLayout(){
    f=new JFrame();
    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
    JButton b6=new JButton("6");
    JButton b7=new JButton("7");
    JButton b8=new JButton("8");
    JButton b9=new JButton("9");
```

Grid layout

```
f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.add(b6);f.add(b7);f.add(b8);f.add(b9);

    f.setLayout(new GridLayout(3,3));
    //setting grid layout of 3 rows and 3 columns

    f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new MyGridLayout();
}
}
```

Flow layout

- **FlowLayout** is the default layout manager.
- Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.
- The constructors are

  FlowLayout( )

  FlowLayout(int how)

  FlowLayout(int how, int horz, int vert)
- The first form creates the default layout, which centers components and leaves five pixels of space between each component.
- The second form allows to specify how each line is aligned. Valid values for are:

  FlowLayout.LEFT

  FlowLayout.CENTER

  FlowLayout.RIGHT

  These values specify left, center, and right alignment, respectively.
- The third form allows to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively

Flow layout

```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
   f=new JFrame();
   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyFlowLayout();
} }
```

Card Layout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.
- **CardLayout** provides these two constructors:

  CardLayout( )

  CardLayout(int horz, int vert)
- The cards are held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager.
- Cards are added to panel using

  void add(Component panelObj, Object name);
- methods defined by **CardLayout**:

  void first(Container deck)

  void last(Container deck)

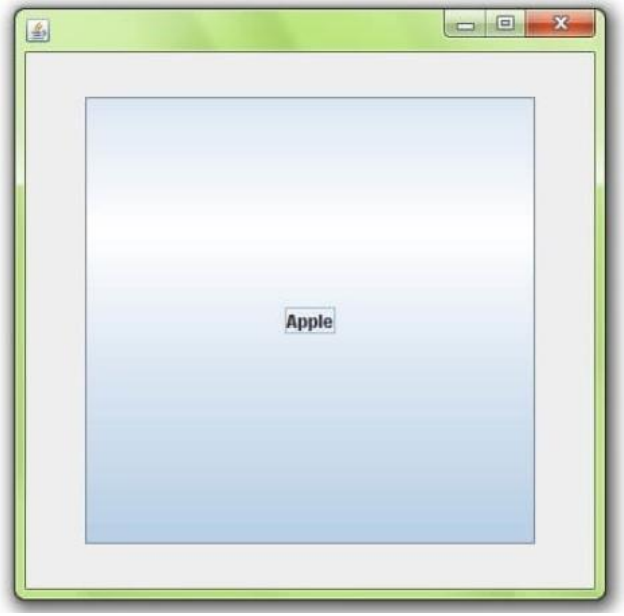
  void next(Container deck)

  void previous(Container deck)

  void show(Container deck, String cardName)

Card Layout

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
    CardLayoutExample(){
        c=getContentPane();
        card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
        c.setLayout(card);
        b1=new JButton("Apple");
        b2=new JButton("Boy");
        b3=new JButton("Cat");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
```



Card Layout

```
    c.add("a",b1);c.add("b",b2);c.add("c",b3);
    }
public void actionPerformed(ActionEvent e)
   {
    card.next(c);
    }
    public static void main(String[] args) {
        CardLayoutExample cl=new CardLayoutExample();
        cl.setSize(400,400);
        cl.setVisible(true);
        cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```

Event handling

- For the user to interact with a GUI, the underlying operating system must support event handling.
  1. operating systems constantly monitor events such as keystrokes, mouse clicks, voice command, etc.
  2. operating systems sort out these events and report them to the appropriate application programs
  3. each application program then decides what to do in response to these events

Events

- An *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.
- Events can be defined as needed and appropriate by application.

Event sources

- A *source* is an object that generates an event.
- This occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- General form is:

  public void add*Type*Listener(*Type*Listener *el*)

  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example,

  1. The method that registers a keyboard event listener is called **addKeyListener()**.
  2. The method that registers a mouse motion listener is called **addMouseMotionListener( )**.

- When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event.
- In all cases, notifications are sent only to listeners that register to receive them.
- Some sources may allow only one listener to register. The general form is:

  public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

  Here Type is the name of the event and *el* is a reference to the event listener.
- When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
- The general form is:
  public void remove*Type*Listener(*Type*Listener *el*)
  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example, to remove a keyboard listener, you would call **removeKeyListener( )**.
- The methods that add or remove listeners are provided by the source that generates events.
- For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event classes

- The Event classes that represent events are at the core of Java's event handling mechanism.
- Super class of the Java event class hierarchy is **EventObject**, which is in **java.util**. for all events.
- Constructor is :

  EventObject(Object *src*)

  Here, *src* is the object that generates this event.
- **EventObject** contains two methods: **getSource( )** and **toString( )**.
- 1. The **getSource( )** method returns the source of the event. General form is : Object getSource( )
- **2.** The **toString( )** returns the string equivalent of the event.

- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.
- The package **java.awt.event** defines several types of events that are generated by various user interface elements.

Event Classes in java.awt.event

- ActionEvent: Generated when a button is pressed, a list item is double clicked, or a menu item is selected.
- AdjustmentEvent: Generated when a scroll bar is manipulated.
- ComponentEvent: Generated when a component is hidden, moved, resized, or becomes visible.
- ContainerEvent: Generated when a component is added to or removed from a container.
- FocusEvent: Generated when a component gains or loses keyboard focus.

- InputEvent: Abstract super class for all component input event classes.
- ItemEvent: Generated when a check box or list item is clicked; also
- occurs when a choice selection is made or a checkable menu item is selected or deselected.
- KeyEvent: Generated when input is received from the keyboard.
- MouseEvent: Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
- TextEvent: Generated when the value of a text area or text field is changed.
- WindowEvent: Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listeners

- A *listener* is an object that is notified when an event occurs.
- Event has two major requirements.
  1. It must have been registered with one or more sources to receive
     notifications about specific types of events.
  2. It must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**.
- For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.
- Any object may receive and process one or both of these events if it provides an implementation of this interface.

Handling Mouse Events

- Mouse events can be handled by implementing the **MouseListener** and the **MouseMotionListener** interfaces.

- **MouseListener Interface** defines five methods. The general forms of these methods are:
    1. void mouseClicked(MouseEvent me)
    2. void mouseEntered(MouseEvent me)
    3. void mouseExited(MouseEvent me)
    4. void mousePressed(MouseEvent me)
    5. void mouseReleased(MouseEvent me)

- **MouseMotionListener Interface.** This interface defines two methods. Their general forms are :
    1. void mouseDragged(MouseEvent me)
    2. void mouseMoved(MouseEvent me)

Handling Keyboard Events

- Keyboard events, can be handled by implementing the **KeyListener** interface.
- **KeyListner** interface defines three methods. The general forms of these methods are :
    1. void keyPressed(KeyEvent ke)
    2. void keyReleased(KeyEvent ke)
    3. void keyTyped(KeyEvent ke)
- To implement keyboard events implementation to the above methods is needed.

Concepts of Applets

- *Applets* are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.
- After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.
- applets – Java program that runs within a Java-enabled browser, invoked through a "applet" reference on a web page, dynamically downloaded to the client computer

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

There are two ways to run an applet:

1. Executing the applet within a Java-compatible Web browser, such as NetscapeNavigator.

2. Using an applet viewer, such as the standard JDK tool, **appletviewer**.

An appletviewer executes your applet in a window. This is generally the fastest and easiest way to test an applet.

To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Differences between applets and applications

Applet	Application
An Applet is an small application program in java that can be embedded HTML page	An Application in java can run alone in client or server
It is derived from JApplet	It is extended from JFrame
It does not have main method	It has main method
It is displayed by HTML	It is displayed through serVisible() method
The size of it applet is specified in HTML	The size of application is defined with the help of seSize() method
It can be runs within a Graphical User Interface (GUI)	It runs without Graphical User Interface (GUI)

Differences between applets and applications

Applet	Application
It can be executed in a java compatible container like a browser or appletviewer	It is executed at command line through java.exe or jview.exe
It gets closed only when HTML document is closed	It is closed with close button
It is portable and can run on environment supported by java browsers	It runs on client machines that has JDK, JRE and JNM
It can be created by extending that clas java.applet.Applet	It can be created using the method public static void main (string args[])

Life cycle of an applet

- Applets life cycle includes the following methods
    1. **init( )**
    2. **start( )**
    3. **paint( )**
    4. **stop( )**
    5. **destroy( )**
- When an applet begins, the AWT calls the following methods, in this sequence:

    init( )

    start( )

    paint( )
- When an applet is terminated, the following sequence of method calls takes place:

    stop( )

    destroy( )

Life cycle of an applet

- **init( ):** The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.
- **start( ):** The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.
- **paint( ):** The **paint( )** method is called each time applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Life cycle of an applet

- **stop( ):** The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. Applet uses **stop( )** to suspend threads that don't need to run when the applet is not visible. To restart **start( )** is called if the user returns to the page.
- **destroy( ):** The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. The **stop( )** method is always called before **destroy( )**.

Types of applets

- Applets are two types
  1. Simple applets
  2. JApplets

- Simple applets can be created by extending Applet class
- JApplets can be created by extending JApplet class of javax.swing.JApplet package

Creating applets

- Applets are created by extending the Applet class.

```
import java.awt.*;
import java.applet.*;
/*<applet code="AppletSkel" width=300 height=100></applet> */
public class AppletSkel extends Applet {
public void init() {
// initialization
}
public void start() {
// start or resume execution
}
public void stop() {
// suspends execution
}
public void destroy() {
// perform shutdown activities
}
public void paint(Graphics g) {
// redisplay contents of window
}}
```

Passing Parameters to applets

- It is also possible to supply user-defined parameters to an applet using <PARAM> tags. Each <PARAM> tag has a name attribute and value attribute.
- Inside the applet code, the applet can refer to that parameter by name to find its value.
- APPLET tag in HTML allows you to pass parameters to applet.
- To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object.

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
```

```
public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
fontSize = -1;
}
param = getParameter("leading");
```

```
try {
if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}}
```