

JAVA PROGRAMMING

UNIT-I

Java History

- 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, Chris Warth, 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.

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 = 10000000000000000000;
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
- type promotion – how the type of a variable can be expanded

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 scopeA 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 loses 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!

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} };`

# 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

|                 |                         |                             |
|-----------------|-------------------------|-----------------------------|
| <code>+=</code> | <code>v += expr;</code> | <code>v = v + expr ;</code> |
| <code>-=</code> | <code>v -=expr;</code>  | <code>v = v - expr ;</code> |
| <code>*=</code> | <code>v *= expr;</code> | <code>v = v * expr ;</code> |
| <code>/=</code> | <code>v /= expr;</code> | <code>v = v / expr ;</code> |
| <code>%=</code> | <code>v %= expr;</code> | <code>v = v % expr ;</code> |

# Basic Arithmetic Operators

|   |                            |           |
|---|----------------------------|-----------|
| + | $\text{op1} + \text{op2}$  | ADD       |
| - | $\text{op1} - \text{op2}$  | SUBTRACT  |
| * | $\text{op1} * \text{op2}$  | MULTIPLY  |
| / | $\text{op1} / \text{op2}$  | DIVISION  |
| % | $\text{op1} \% \text{op2}$ | 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<br>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  |

# 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:  $1 * 2 * 3$

# 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.

# 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 Conversion
  - 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;`

# Simple Java Program

- A class to display a simple message:  
class MyProgram  
{  
public static void main(String[] args)  
{  
System.out.println("First Java program.");  
}  
}

# What is an Object?

- Real world objects are things that have:

- 1) state

- 2) behavior

- Example: your dog:

- state – name, color, breed, sits?, barks?, wags tail?, runs?
  - behavior – sitting, barking, wagging 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 class
- When 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; }
}
```

# Keyword this

- Can be used by any object to refer to itself in any class method
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors

# Keyword this

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
 this.width = width;
 this.height = height;
 this.depth = depth;
}
```

# Garbage Collection

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:
  - 1) The garbage collector keeps track of how many references an object has.
  - 2) It removes an object from memory when it has no longer any references.
  - 3) Thereafter, the memory occupied by the object can be allocated again.
  - 4) The garbage collector invokes the finalize method.

# finalize() Method

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:
- 1) implemented inside a class as:  

```
protected void finalize() { ... }
```
- 2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out



# Method Overloading

- It is legal for a class to have two or more methods with the same name.
- However, 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.

# Example: Overloading

```
class OverloadDemo {
 void test() {
 System.out.println("No parameters");
 }
 void test(int a) {
 System.out.println("a: " + a);
 }
 void test(int a, int b) {
 System.out.println("a and b: " + a + " " + b);
 }
 double test(double a) {
 System.out.println("double a: " + a); return a*a;
 }
}
```

# Constructor Overloading

```
class Box {
 double width, height, depth;
 Box(double w, double h, double d) {
 width = w; height = h; depth = d;
 }
 Box() {
 width = -1; height = -1; depth = -1;
 }
 Box(double len) {
 width = height = depth = len;
 }
 double volume() { return width * height * depth; }
}
```

# Parameter Passing

- Two types of variables:
  - 1) simple types
  - 2) class types
- Two corresponding ways of how the arguments are passed to methods:
  - 1) by value a method receives a copy of the original value; parameters of simple types
  - 2) by reference a method receives the memory address of the original value, not the value itself; parameters of class types

# Call by value

```
class CallByValue {
 public static void main(String args[]) {
 Test ob = new Test();
 int a = 15, b = 20;
 System.out.print("a and b before call: ");
 System.out.println(a + " " + b);
 ob.meth(a, b);
 System.out.print("a and b after call: ");
 System.out.println(a + " " + b);
 }
}
```

# Call by reference

- **As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:**

```
class CallByRef {
 public static void main(String args[]) {
 Test ob = new Test(15, 20);
 System.out.print("ob.a and ob.b before call: ");
 System.out.println(ob.a + " " + ob.b);
 ob.meth(ob);
 System.out.print("ob.a and ob.b after call: ");
 System.out.println(ob.a + " " + ob.b);
 }
}
```

# Recursion

- A recursive method is a method that calls itself:
  - 1) all method parameters and local variables are allocated on the stack
  - 2) arguments are prepared in the corresponding parameter positions
  - 3) the method code is executed for the new arguments
  - 4) upon return, all parameters and variables are removed from the stack
  - 5) the execution continues immediately after the invocation point

# Example: Recursion

```
class Factorial {
 int fact(int n) {
 if (n==1) return 1;
 return fact(n-1) * n;
 }
}

class Recursion {
 public static void main(String args[]) {
 Factorial f = new Factorial();
 System.out.print("Factorial of 5 is ");
 System.out.println(f.fact(5));
 } }
}
```



# 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");
} }
```

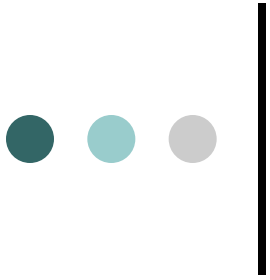
**This program generates the following output:**

**Length of strOb1: 12**

**Char at index 3 in strOb1: s**

**strOb1 != strOb2**

**strOb1 == strOb3**



# **JAVA PROGRAMMING**

## **UNIT-3**



# Concepts of exception handling

## 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



## Separating Error Handling Code from Regular Code

In traditional programming, error detection, reporting, and handling often lead to confusing code

Consider pseudocode method here that reads an entire file into memory

```
readFile {
 open the file;
 determine its size;
 allocate that much memory;
 read the file into memory;
 close the file;
}
```



## Traditional Programming: No separation of error handling code

- In traditional programming, To handle such cases, the *readFile* function must have more code to do error detection, reporting, and handling.

```
errorCodeType readFile {
 initialize errorCode = 0;
 open the file;
 if (theFileIsOpen) {
 determine the length of the file;
 if (gotTheFileLength) {
 allocate that much memory;
 if (gotEnoughMemory) {
 read the file into memory;
 if (readFailed) {
 errorCode = -1;
 }
 }
 }
 }
}
```



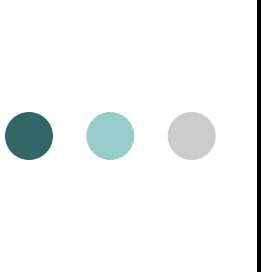
```
 else {
 errorCode = -2;
 }
} else {
 errorCode = -3;
}
close the file;
if (theFileDintClose && errorCode == 0) {
 errorCode = -4;
} else {
 errorCode = errorCode and -4;
}
} else {
 errorCode = -5;
}
return errorCode;
}
```



## Separating Error Handling Code from Regular Code (in Java)

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere

```
readFile {
 try {
 open the file;
 determine its size;
 allocate that much memory;
 read the file into memory;
 close the file;
 } catch (fileOpenFailed) {
 doSomething;
 }
}
```



```
catch (sizeDeterminationFailed) {
doSomething;
} catch (memoryAllocationFailed) {
doSomething;
} catch (readFailed) {
doSomething;
} catch (fileCloseFailed) {
doSomething;
}
}
```

- Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.





# Propagating Errors Up the Call Stack

- ❖ Suppose that the *readFile* method is the fourth method in a series of nested method calls made by the main program: *method1* calls *method2*, which calls *method3*, which finally calls *readFile*
- ❖ Suppose also that *method1* is the only method interested in the errors that might occur within *readFile*.

```
method1 {
 call method2;
}
method2 {
 call method3;
}
method3 {
 call readFile;
}
```



## Traditional Way of Propagating Errors

```
method1 {
 errorCodeType error;
 error = call method2;
 if (error)
 doErrorProcessing;
 else
 proceed;
}
errorCodeType method2 {
 errorCodeType error;
 error = call method3;
 if (error)
 return error;
 else
 proceed;
}
errorCodeType method3 {
 errorCodeType error;
 error = call readFile;
 if (error)
 return error;
 else
 proceed;
}
```

- Traditional error notification Techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.



## Using Java Exception Handling

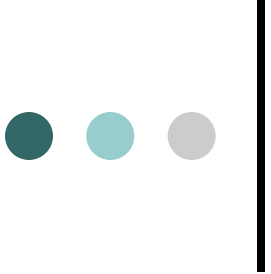
```
method1 {
 try {
 call method2;
 } catch (exception e) {
 doErrorProcessing;
 }
}
method2 throws exception {
 call method3;
}
method3 throws exception {
 call readFile;
}
```

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



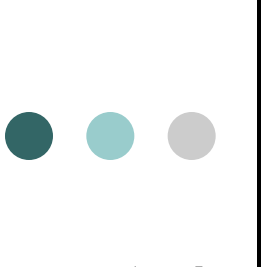
## Grouping and Differentiating Error Types

- ❖ 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) {

 ...
 }
```

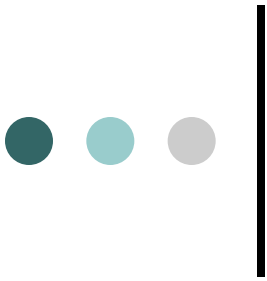
- 
- ❖ A method can catch an exception based on its group or general type by specifying any of the exception's super classes in the catch statement.
  - ❖ For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an `IOException` argument.

```
// 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.





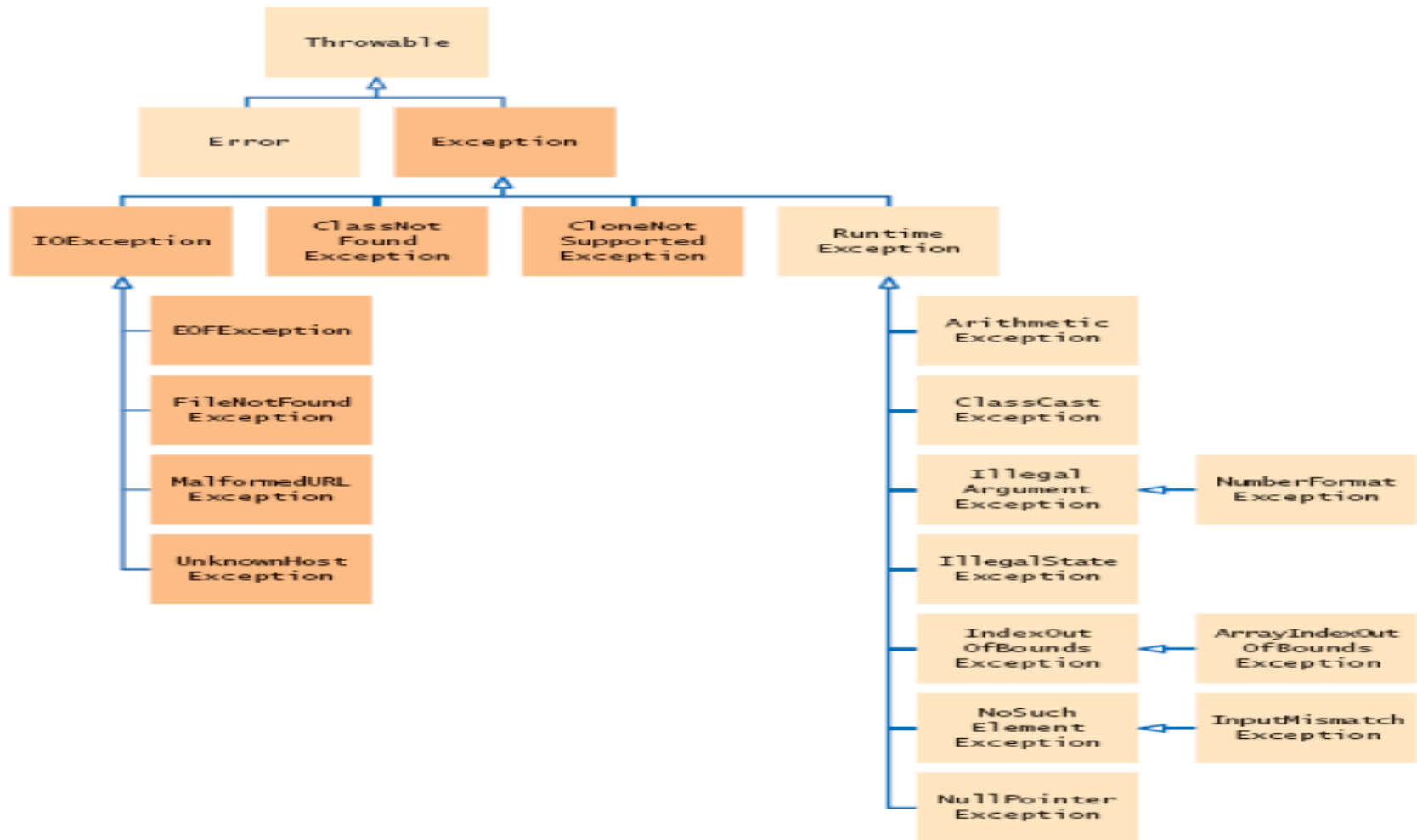
# Exception Hierarchy

- 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!");
 }
 }
}
```



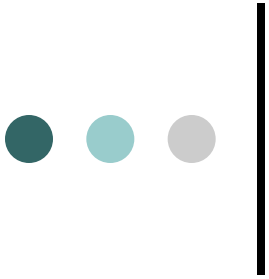
# Throwing Exceptions(throw)

- 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 in 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 or 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();
}
```



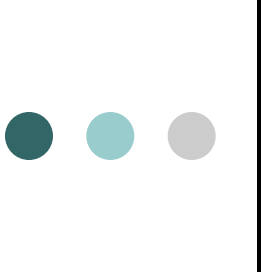
# Java Built-In Exceptions

- 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 these 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. StringIndexOutOfBoundsException
  9. UnsupportedOperationException
  10. SecurityException
  11. NumberFormatException
  12. NullPointerException
  13. NegativeArraySizeException
  14. IndexOutOfBoundsException
  15. IllegalStateException





# 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);
 }
}
```



# 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

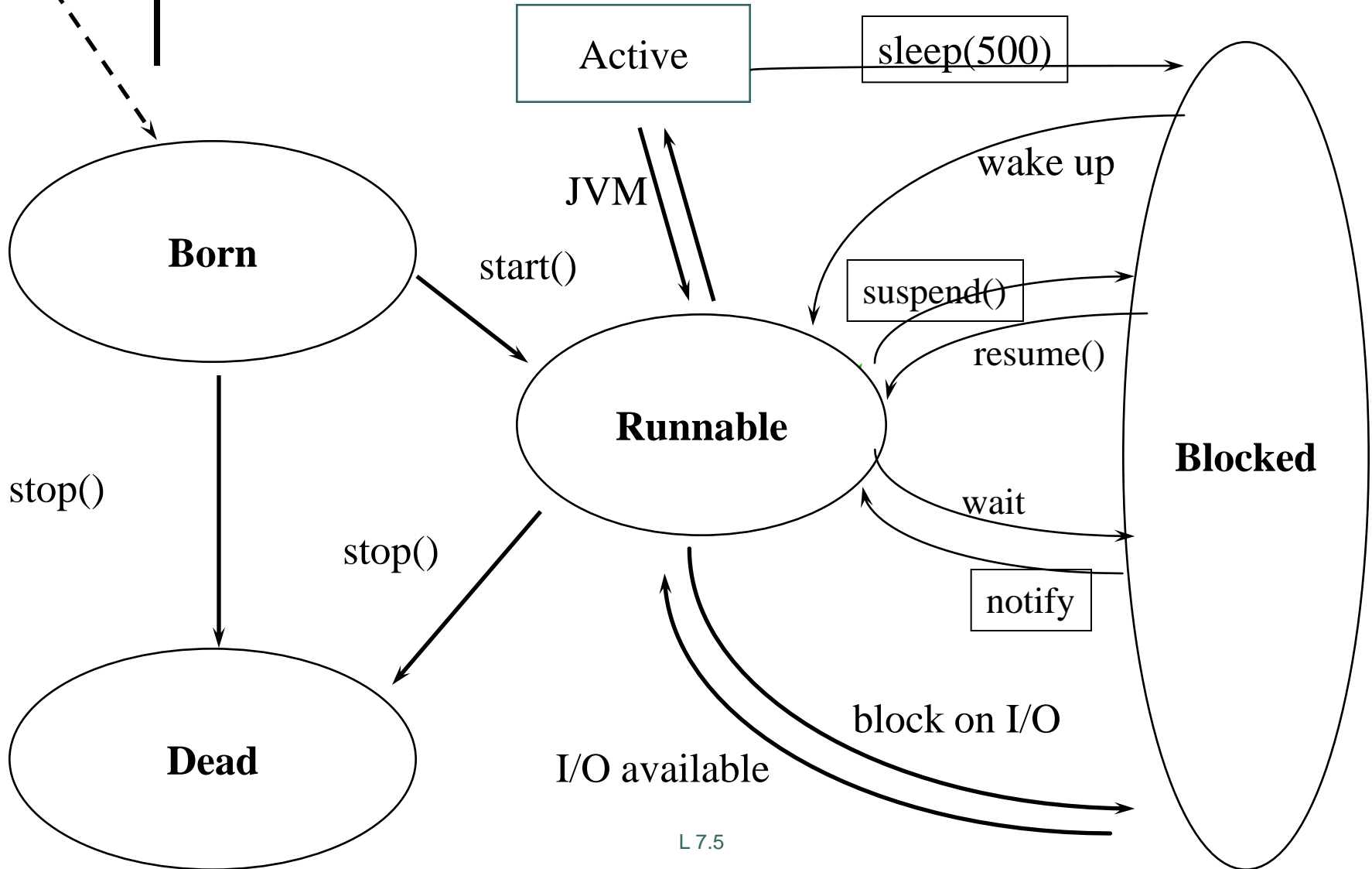




# Thread Lifecycle

- 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





- **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: start a thread by calling 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



# 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");
```

# JAVA PROGRAMMING

## UNIT-4

# APIs and Versions

- Number one hint for programming with Java Collections: use the API
  - <http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collection.html>
- Be sure to use the 1.5.0 APIs to get the version with generics



# Java Collections Framework

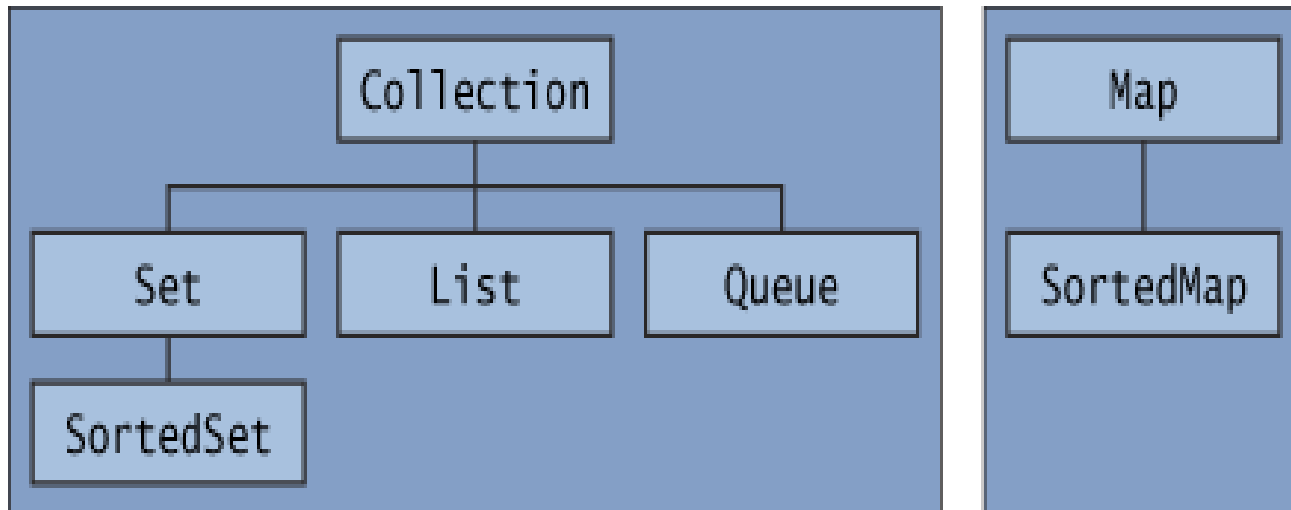
- The Java language API provides many of the data structures from this class for you.
- It defines a “collection” as “an object that represents a group of objects”.
- It defines a collections framework as “a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation.”

# Collections Framework (cont)

- **Collection Interfaces** - Represent different types of collections, such as sets, lists and maps. These interfaces form the basis of the framework.
- **General-purpose Implementations** - Primary implementations of the collection interfaces.
- **Legacy Implementations** - The collection classes from earlier releases, Vector and Hashtable, have been retrofitted to implement the collection interfaces.
- **Wrapper Implementations** - Add functionality, such as synchronization, to other implementations.
- **Convenience Implementations** - High-performance "mini-implementations" of the collection interfaces.
- **Abstract Implementations** - Partial implementations of the collection interfaces to facilitate custom implementations.
- **Algorithms** - Static methods that perform useful functions on collections, such as sorting a list.
- **Infrastructure** - Interfaces that provide essential support for the collection interfaces.
- **Array Utilities** - Utility functions for arrays of primitives and reference objects. Not, strictly speaking, a part of the Collections Framework, this functionality is being added to the Java platform at the same time and relies on some of the same infrastructure.

# Collection interfaces

- The core collection interfaces encapsulate different types of collections. They represent the abstract data types that are part of the collections framework. They are interfaces so they do not provide an implementation!



# public interface Collection<E> extends Iterable<E>

- Collection — the root of the collection hierarchy. A collection represents a group of objects known as its *elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

# public interface Collection<E> extends Iterable<E>

```
public interface Collection<E> extends Iterable<E> {
 // Basic operations
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element); //optional
 boolean remove(Object element); //optional
 Iterator<E> iterator();

 // Bulk operations
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c); //optional
 boolean removeAll(Collection<?> c); //optional
 boolean retainAll(Collection<?> c); //optional
 void clear(); //optional

 // Array operations
 Object[] toArray();
 <T> T[] toArray(T[] a);
}
```

# A note on iterators

- An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator()` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
 boolean hasNext();
 E next();
 void remove(); //optional
}
```

public interface **Set**<E>  
extends Collection<E>

- Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine.

# public interface **Set**<E> extends Collection<E>

```
public interface Set<E> extends Collection<E> {
 // Basic operations
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element); //optional
 boolean remove(Object element); //optional
 Iterator<E> iterator();

 // Bulk operations
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c); //optional
 boolean removeAll(Collection<?> c); //optional
 boolean retainAll(Collection<?> c); //optional
 void clear(); //optional

 // Array Operations
 Object[] toArray();
 <T> T[] toArray(T[] a);
}
```

Note: nothing added to Collection interface – except no duplicates allowed



public interface **List**<E>  
extends Collection<E>

- List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor of List.

# public interface **List**<E> extends Collection<E>

```
public interface List<E> extends Collection<E> {
 // Positional access
 E get(int index);
 E set(int index, E element); //optional
 boolean add(E element); //optional
 void add(int index, E element); //optional
 E remove(int index); //optional
 boolean addAll(int index,
 Collection<? extends E> c); //optional

 // Search
 int indexOf(Object o);
 int lastIndexOf(Object o);

 // Iteration
 ListIterator<E> listIterator();
 ListIterator<E> listIterator(int index);

 // Range-view
 List<E> subList(int from, int to);
}
```

# A note on ListIterators

- The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasNext and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.
- The nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous
- The set method overwrites the last element returned by next or previous with the specified element.
- The add method inserts a new element into the list immediately before the current cursor position.

```
public interface ListIterator<E> extends Iterator<E> {
 boolean hasNext();
 E next();
 boolean hasPrevious();
 E previous();
 int nextIndex();
 int previousIndex();
 void remove(); //optional
 void set(E e); //optional
 void add(E e); //optional
}
```



public interface **Queue**<E>  
extends Collection<E>

- Queue — a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations.

**public interface Queue<E>**  
**extends Collection<E>**

```
public interface Queue<E> extends
 Collection<E> {
 E element(); //throws
 E peek(); //null
 boolean offer(E e); //add - bool
 E remove(); //throws
 E poll(); //null
}
```

# public interface **Map**<**K**,**V**>

- Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map.

# public interface Map<K,V>

```
public interface Map<K,V> {

 // Basic operations
 V put(K key, V value);
 V get(Object key);
 V remove(Object key);
 boolean containsKey(Object key);
 boolean containsValue(Object value);
 int size();
 boolean isEmpty();

 // Bulk operations
 void putAll(Map<? extends K, ? extends V> m);
 void clear();

 // Collection Views
 public Set<K> keySet();
 public Collection<V> values();
 public Set<Map.Entry<K,V>> entrySet();

 // Interface for entrySet elements
 public interface Entry {
 K getKey();
 V getValue();
 V setValue(V value);
 }
}
```

```
public interface SortedSet<E>
 extends Set<E>
```

- SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.



# public interface **SortedSet**<E> extends Set<E>

```
public interface SortedSet<E> extends Set<E> {
 // Range-view
 SortedSet<E> subSet(E fromElement, E toElement);
 SortedSet<E> headSet(E toElement);
 SortedSet<E> tailSet(E fromElement);

 // Endpoints
 E first();
 E last();

 // Comparator access
 Comparator<? super E> comparator();
}
```

# Note on Comparator interface

- Comparator is another interface (in addition to Comparable) provided by the Java API which can be used to order objects.
- You can use this interface to define an order that is different from the Comparable (natural) order.

public interface **SortedMap**<K,V>  
extends Map<K,V>

- SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

public interface **SortedMap**<K,V>  
extends Map<K,V>

```
public interface SortedMap<K, V> extends Map<K, V>{

 SortedMap<K, V> subMap(K fromKey, K toKey);
 SortedMap<K, V> headMap(K toKey);
 SortedMap<K, V> tailMap(K fromKey);
 K firstKey();
 K lastKey();

 Comparator<? super K> comparator();
}
```

# General-purpose Implementations

| Interfaces | Implementations |                 |                            |             |                          |
|------------|-----------------|-----------------|----------------------------|-------------|--------------------------|
|            | Hash table      | Resizable array | Tree<br><u>(sorted)</u>    | Linked list | Hash table + Linked list |
| Set        | HashSet         |                 | TreeSet<br><u>(sorted)</u> |             | LinkedHashSet            |
| List       |                 | ArrayList       |                            | LinkedList  |                          |
| Queue      |                 |                 |                            |             |                          |
| Map        | HashMap         |                 | TreeMap<br><u>(sorted)</u> |             | LinkedHashMap            |

Note the naming convention

LinkedList also implements queue and there is a PriorityQueue implementation (implemented with heap)

# implementations

- Each of the implementations offers the strengths and weaknesses of the underlying data structure.
- What does that mean for:
  - Hashtable
  - Resizable array
  - Tree
  - LinkedList
  - Hashtable plus LinkedList
- **Think about these tradeoffs when selecting the implementation!**

# Choosing the datatype

- When you declare a Set, List or Map, you should use Set, List or Map interface as the datatype instead of the implementing class. That will allow you to change the implementation by changing a single line of code!

---

```
import java.util.*;

public class Test {
 public static void main(String[] args) {
 Set<String> ss = new LinkedHashSet<String>();

 for (int i = 0; i < args.length; i++)
 ss.add(args[i]);

 Iterator i = ss.iterator();
 while (i.hasNext())
 System.out.println(i.next());
 }
}
```

```
import java.util.*;

public class Test {

 public static void main(String[] args)
 {
 //map to hold student grades
 Map<String, Integer> theMap = new HashMap<String, Integer>();

 theMap.put("Korth, Evan", 100);
 theMap.put("Plant, Robert", 90);
 theMap.put("Coyne, Wayne", 92);
 theMap.put("Franti, Michael", 98);
 theMap.put("Lennon, John", 88);

 System.out.println(theMap);
 System.out.println("-----");
 System.out.println(theMap.get("Korth, Evan"));
 System.out.println(theMap.get("Franti, Michael"));

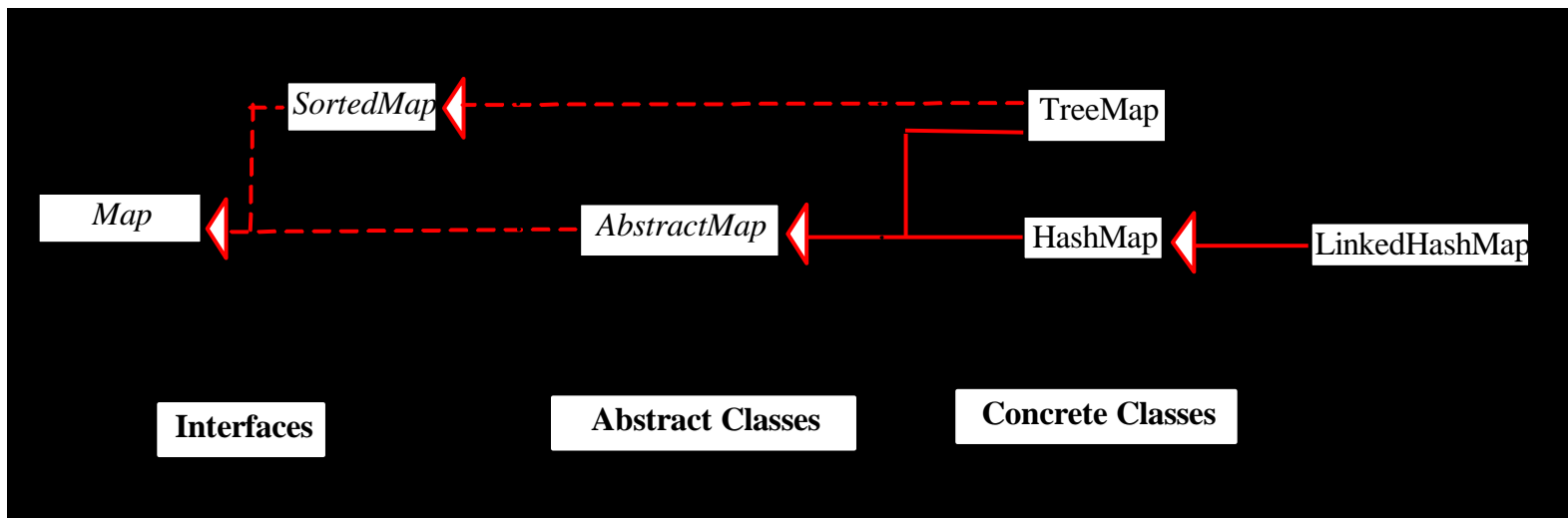
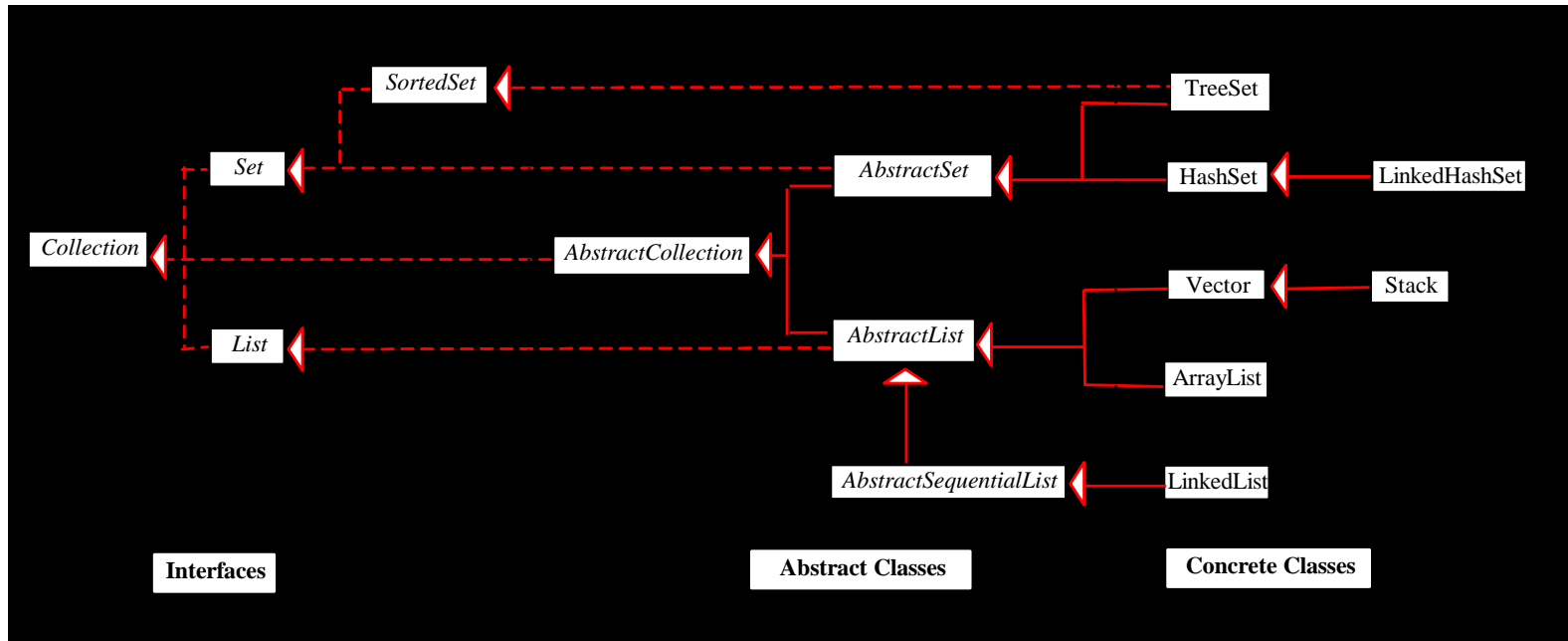
 }

}
```



# Other implementations in the API

- Wrapper implementations delegate all their real work to a specified collection but add (or remove) extra functionality on top of what the collection offers.
  - Synchronization Wrappers
  - Unmodifiable Wrappers
- Convenience implementations are mini-implementations that can be more convenient and more efficient than general-purpose implementations when you don't need their full power
  - List View of an Array
  - Immutable Multiple-Copy List
  - Immutable Singleton Set
  - Empty Set, List, and Map Constants



# Making your own implementations

- Most of the time you can use the implementations provided for you in the Java API.
- In case the existing implementations do not satisfy your needs, you can write your own by extending the abstract classes provided in the collections framework.

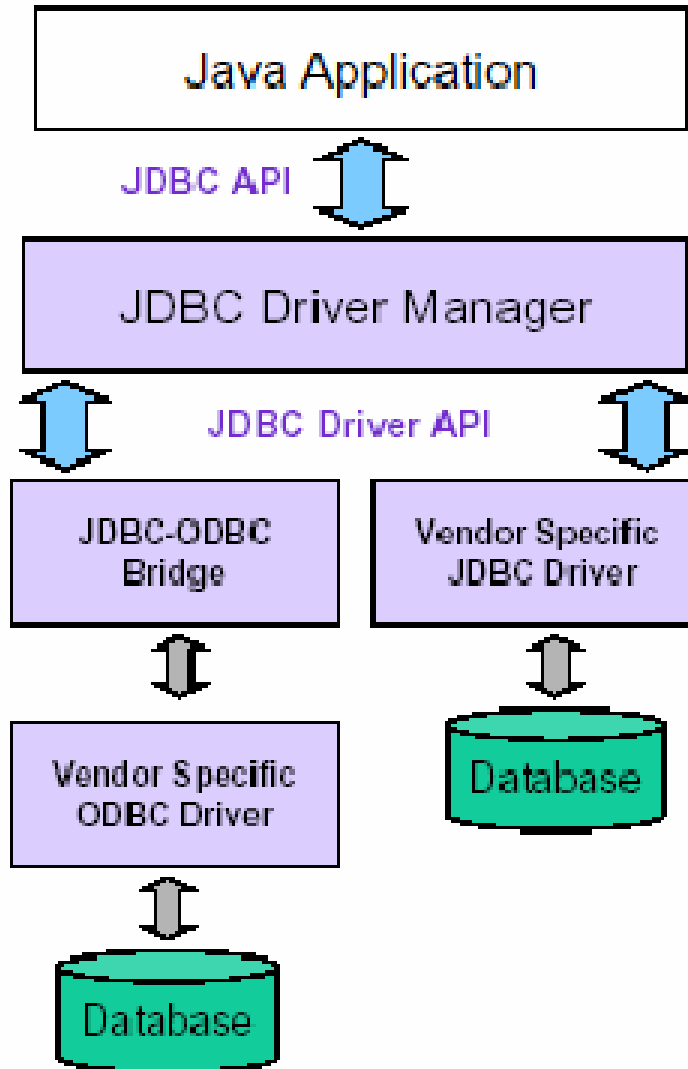
# algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
    - Reverse
    - Fill copy
    - etc.
  - Searching
    - Binary Search
  - Composition
    - Frequency
    - Disjoint
  - Finding extreme values
    - Min
    - Max

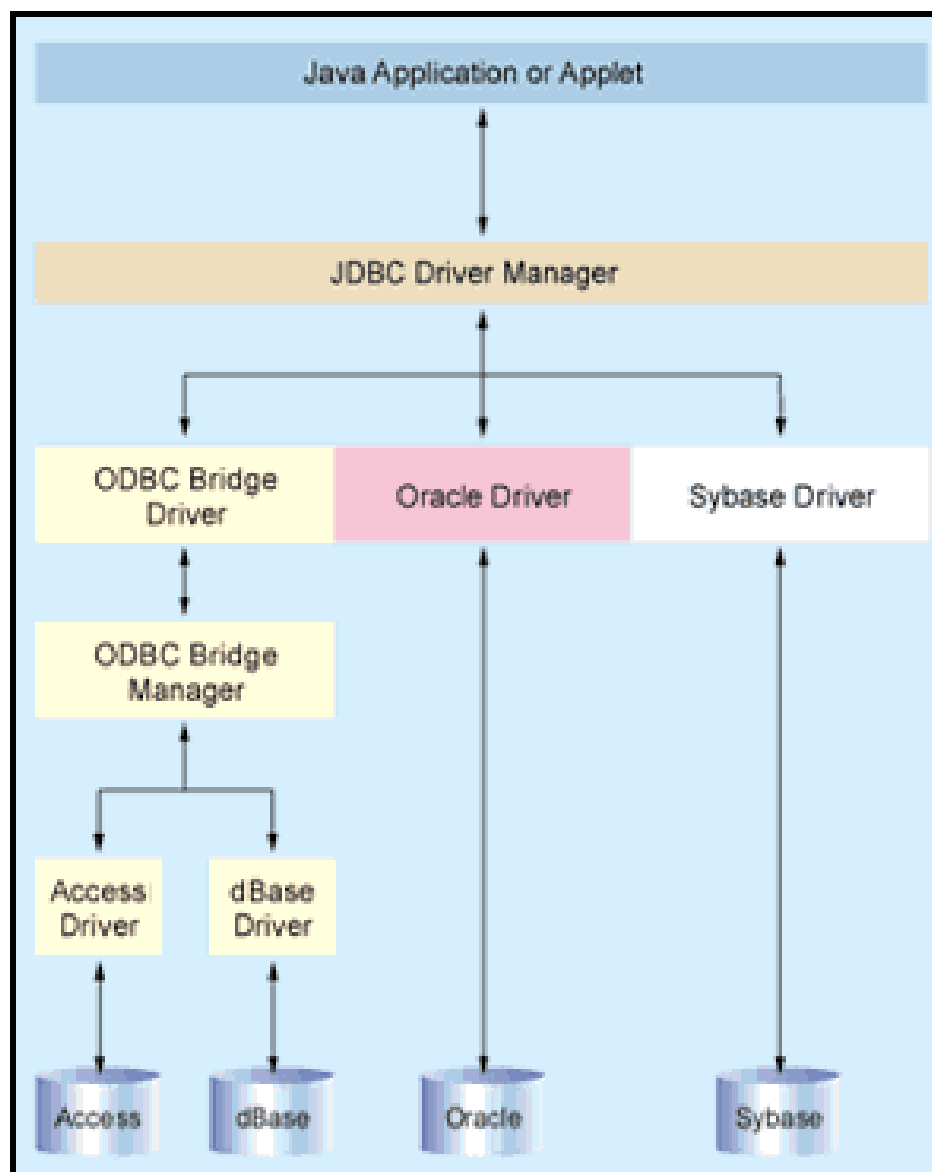
# What is JDBC?

- “An API that lets you access virtually **any tabular data source** from the Java programming language”
  - JDBC Data Access API – JDBC Technology Homepage
    - What’s an API?
      - See J2SE documentation
    - What’s a tabular data source?
- “... access virtually any data source, from **relational databases** to **spreadsheets** and **flat files**.”
  - JDBC Documentation
- We’ll focus on accessing Oracle databases

# General Architecture



- What design pattern is implied in this architecture?
- What does it buy for us?
- Why is this architecture also multi-tiered?



**Figure 1. Anatomy of Data Access.** The Driver Manager provides a consistent layer between your Java app and back-end database. JDBC works natively (such as with the Oracle driver in this example) or with any ODBC datasource.

# Basic steps to use a database in Java

- 1. Establish a **connection**
- 2. Create JDBC **Statements**
- 3. Execute **SQL** Statements
- 4. **GET ResultSet**
- 5. **Close** connections



# 1. Establish a connection

- **import java.sql.\*;**
- **Load the vendor specific driver**
  - `Class.forName("oracle.jdbc.driver.OracleDriver");`
    - What do you think this statement does, and how?
    - Dynamically loads a driver class, for Oracle database
- **Make the connection**
  - `Connection con = DriverManager.getConnection("jdbc:oracle:thin:@oracle-prod:1521:OPROD", username, passwd);`
    - What do you think this statement does?
    - Establishes connection to database by obtaining a *Connection* object

## 2. Create JDBC statement(s)

- `Statement stmt = con.createStatement() ;`
- Creates a Statement object for sending SQL statements to the database

# Executing SQL Statements

- String createLehigh = "Create table Lehigh " +  
"(SSN Integer not null, Name VARCHAR(32), " +  
"Marks Integer)";  
stmt.**executeUpdate**(createLehigh);  
//What does this statement do?
- String insertLehigh = "Insert into Lehigh values"  
+ "(123456789,abc,100)";  
stmt.**executeUpdate**(insertLehigh);

# Get ResultSet

```
String queryLehigh = "select * from Lehigh";
```

```
ResultSet rs = Stmt.executeQuery(queryLehigh);
```

```
//What does this statement do?
```

```
while (rs.next()) {
 int ssn = rs.getInt("SSN");
 String name = rs.getString("NAME");
 int marks = rs.getInt("MARKS");
}
```

# Close connection

- `stmt.close();`
- `con.close();`

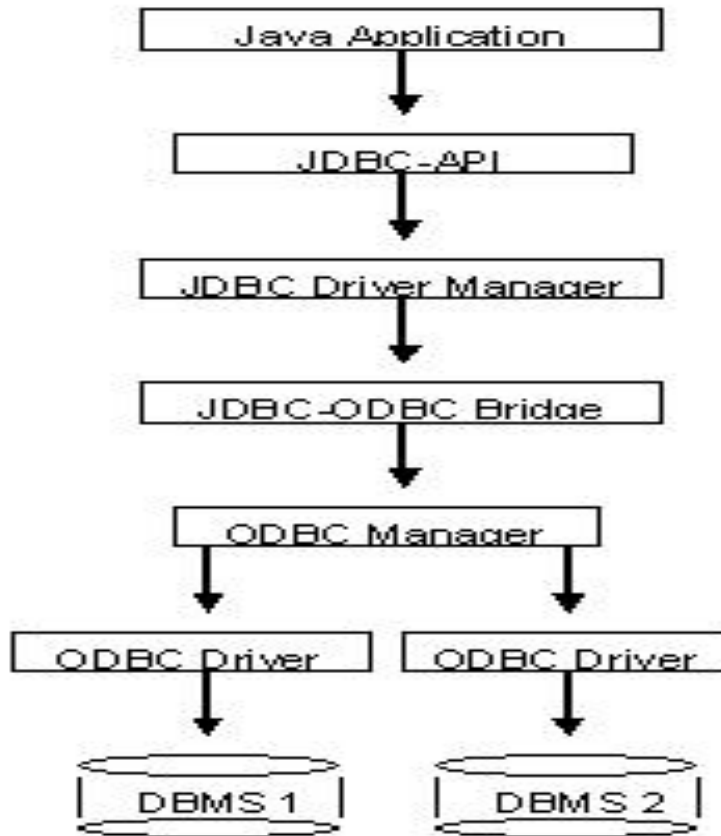
# Transactions and JDBC

- JDBC allows SQL statements to be grouped together into a single transaction
- Transaction control is performed by the `Connection` object, default mode is auto-commit, I.e., each sql statement is treated as a transaction
- We can turn off the auto-commit mode with `con.setAutoCommit(false);`
- And turn it back on with `con.setAutoCommit(true);`
- Once auto-commit is off, no SQL statement will be committed until an explicit is invoked `con.commit();`
- At this point all changes done by the SQL statements will be made permanent in the database.

# Handling Errors with Exceptions

- Programs should recover and leave the database in a consistent state.
- If a statement in the try block throws an exception or warning, it can be caught in one of the corresponding catch statements
- How might a `finally {...}` block be helpful here?
- E.g., you could rollback your transaction in a `catch { ...}` block or close database connection and free database related resources in `finally {...}` block

# Another way to access database (JDBC-ODBC)



What's a bit different about this architecture?

Why add yet another layer?



# Sample program

```
import java.sql.*;
class Test {
 public static void main(String[] args) {
 try {
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); //dynamic loading of driver
 String filename = "c:/db1.mdb"; //Location of an Access database
 String database = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=";
 database+= filename.trim() + ";DriverID=22;READONLY=true}"; //add on to end
 Connection con = DriverManager.getConnection(database , "", "");
 Statement s = con.createStatement();
 s.execute("create table TEST12345 (firstcolumn integer)");
 s.execute("insert into TEST12345 values(1)");
 s.execute("select firstcolumn from TEST12345");
```

# Sample program(cont)

```
ResultSet rs = s.getResultSet();
if (rs != null) // if rs == null, then there is no ResultSet to view
while (rs.next()) // this will step through our data row-by-row
{ /* the next line will get the first column in our current row's ResultSet
 as a String (getString(columnNumber)) and output it to the screen */
 System.out.println("Data from column_name: " + rs.getString(1));
}
s.close(); // close Statement to let the database know we're done with it
con.close(); //close connection
}
catch (Exception err) { System.out.println("ERROR: " + err); }
}
}
```

# Mapping types JDBC - Java

| JDBC Type     | Java Type |
|---------------|-----------|
| BIT           | boolean   |
| TINYINT       | byte      |
| SMALLINT      | short     |
| INTEGER       | int       |
| BIGINT        | long      |
| REAL          | float     |
| FLOAT         | double    |
| DOUBLE        |           |
| BINARY        | byte[]    |
| VARBINARY     |           |
| LONGVARBINARY |           |
| CHAR          | String    |
| VARCHAR       |           |
| LONGVARCHAR   |           |

| JDBC Type   | Java Type                  |
|-------------|----------------------------|
| NUMERIC     | BigDecimal                 |
| DECIMAL     |                            |
| DATE        | java.sql.Date              |
| TIME        | java.sql.Timestamp         |
| TIMESTAMP   |                            |
| CLOB        | Clob*                      |
| BLOB        | Blob*                      |
| ARRAY       | Array*                     |
| DISTINCT    | mapping of underlying type |
| STRUCT      | Struct*                    |
| REF         | Ref*                       |
| JAVA_OBJECT | underlying Java class      |

\*SQL3 data type supported in JDBC 2.0

# JDBC 2 – Scrollable Result Set

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_READ_ONLY);
```

```
String query = "select students from class where type='not sleeping'";
ResultSet rs = stmt.executeQuery(query);
```

```
rs.previous(); // go back in the RS (not possible in JDBC 1...)
rs.relative(-5); // go 5 records back
rs.relative(7); // go 7 records forward
rs.absolute(100); // go to 100th record
```

```
...
```

# JDBC 2 – Updateable ResultSet

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
 ResultSet.CONCUR_UPDATABLE);
String query = " select students, grade from class
 where type='really listening this presentation😊' ";
ResultSet rs = stmt.executeQuery(query);
...
while (rs.next())
{
 int grade = rs.getInt("grade");
 rs.updateInt("grade", grade+10);
 rs.updateRow();
}
```

# Metadata from DB

- A **Connection's** database is able to provide **schema** information describing its tables, its supported SQL grammar, its stored procedures the capabilities of this connection, and so on
  - What is a **stored procedure**?
  - Group of SQL statements that form a logical unit and perform a particular task

This information is made available through a **DatabaseMetaData** object.

# Metadata from DB - example

...

```
Connection con = ;
```

```
DatabaseMetaData dbmd = con.getMetaData();
```

```
String catalog = null;
```

```
String schema = null;
```

```
String table = "sys%";
```

```
String[] types = null;
```

```
ResultSet rs =
```

```
 dbmd.getTables(catalog , schema , table , types);
```

...

# JDBC – Metadata from RS

```
public static void printRS(ResultSet rs) throws SQLException
{
 ResultSetMetaData md = rs.getMetaData();
 // get number of columns
 int nCols = md.getColumnCount();
 // print column names
 for(int i=1; i < nCols; ++i)
 System.out.print(md.getColumnName(i)+",");
 // output resultset
 while (rs.next())
 {
 for(int i=1; i < nCols; ++i)
 System.out.print(rs.getString(i)+",");
 System.out.println(rs.getString(nCols));
 }
}
```



# JDBC and beyond

- (JNDI) Java Naming and Directory Interface
  - API for network-wide sharing of information about users, machines, networks, services, and applications
  - Preserves Java's object model
- (JDO) Java Data Object
  - Models persistence of objects, using RDBMS as repository
  - Save, load objects from RDBMS
- (SQLJ) Embedded SQL in Java
  - Standardized and optimized by Sybase, Oracle and IBM
  - Java extended with directives: `# sql`
  - SQL routines can invoke Java methods
  - Maps SQL types to Java classes

# JDBC references

- JDBC Data Access API – JDBC Technology Homepage
  - <http://java.sun.com/products/jdbc/index.html>
- JDBC Database Access – The Java Tutorial
  - <http://java.sun.com/docs/books/tutorial/jdbc/index.html>
- JDBC Documentation
  - <http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/index.html>
- java.sql package
  - <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html>
- JDBC Technology Guide: Getting Started
  - <http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>
- JDBC API Tutorial and Reference (book)
  - <http://java.sun.com/docs/books/jdbc/>

# JDBC

- JDBC Data Access API – JDBC Technology Homepage
  - <http://java.sun.com/products/jdbc/index.html>
- JDBC Database Access – The Java Tutorial
  - <http://java.sun.com/docs/books/tutorial/jdbc/index.html>
- JDBC Documentation
  - <http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/index.html>
- java.sql package
  - <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html>
- JDBC Technology Guide: Getting Started
  - <http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>
- JDBC API Tutorial and Reference (book)
  - <http://java.sun.com/docs/books/jdbc/>

# **JAVA PROGRAMMING**

## **UNIT-V**

# 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 addTypeListener(TypeListener e/)
```

Here, *Type* is the name of the event and *e/* 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 addTypeListener(TypeListener e)  
throws java.util.TooManyListenersException  
*Here Type* is the name of the event and *e* 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 removeTypeListener(TypeListener e/)`  
Here, *Type* is the name of the event and *e/* 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.

# Delegation event model

- The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*.
- In this scheme, the listener simply waits until it receives an event.
- Once received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to "delegate" the processing of an event to a separate piece of code.



- In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.
- This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component.
- This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

### **Note**

- Java also allows you to process events without using the delegation event model.
- This can be done by extending an AWT component.

# 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.
- **KeyListener** 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.

# Adapter classes

- Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers.
- An adapter class provides an empty implementation of all methods in an event listener interface.
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

- adapter classes in **java.awt.event** are.

### **Adapter Class**

ComponentAdapter

ContainerAdapter

FocusAdapter

KeyAdapter

MouseAdapter

MouseMotionAdapter

WindowAdapter

### **Listener Interface**

ComponentListener

ContainerListener

FocusListener

KeyListener

MouseListener

MouseMotionListener

WindowListener

# Inner classes

- Inner classes, which allow one class to be defined within another.
- An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.
- An inner class is fully within the scope of its enclosing class.
- an inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class

# 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**
  1. AWTEvent:Encapsulates AWT events.
  2. AWTEventMulticaster: Dispatches events to multiple listeners.
  3. BorderLayout: The border layout manager. Border layouts use five components: North, South, East, West, and Center.
  4. Button: Creates a push button control.
  5. Canvas: A blank, semantics-free window.
  6. CardLayout: The card layout manager. Card layouts emulate index cards. Only the one on top is showing.

7. Checkbox: Creates a check box control.
8. CheckboxGroup: Creates a group of check box controls.
9. CheckboxMenuItem: Creates an on/off menu item.
10. Choice: Creates a pop-up list.
11. Color: Manages colors in a portable, platform-independent fashion.
12. Component: An abstract super class for various AWT components.
13. Container: A subclass of Component that can hold other components.
14. Cursor: Encapsulates a bitmapped cursor.
15. Dialog: Creates a dialog window.
16. Dimension: Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.
17. Event: Encapsulates events.
18. EventQueue: Queues events.
19. FileDialog: Creates a window from which a file can be selected.
20. FlowLayout: The flow layout manager. Flow layout positions components left to right, top to bottom.



- 21. `Font`: Encapsulates a type font.
- 22. `FontMetrics`: Encapsulates various information related to a font. This information helps you display text in a window.
- 23. `Frame`: Creates a standard window that has a title bar, resize corners, and a menu bar.
- 24. `Graphics`: Encapsulates the graphics context. This context is used by various output methods to display output in a window.
- 25. `GraphicsDevice`: Describes a graphics device such as a screen or printer.
- 26. `GraphicsEnvironment`: Describes the collection of available `Font` and `GraphicsDevice` objects.
- 27. `GridBagConstraints`: Defines various constraints relating to the `GridBagLayout` class.
- 28. `GridBagLayout`: The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by `GridBagConstraints`.
- 29. `GridLayout`: The grid layout manager. Grid layout displays components in a two-dimensional grid.

- 30. Scrollbar: Creates a scroll bar control.
- 31. ScrollPane: A container that provides horizontal and/or vertical scrollbars for another component.
- 32. SystemColor: Contains the colors of GUI widgets such as windows, scrollbars, text, and others.
- 33. TextArea: Creates a multiline edit control.
- 34. TextComponent: A super class for TextArea and TextField.
- 35. TextField: Creates a single-line edit control.
- 36. Toolkit: Abstract class implemented by the AWT.
- 37. Window: Creates a window with no frame, no menu bar, and no title.

# user interface components

- **Labels:** Creates a label that displays a string.
- A *label* is an object of type **Label**, and it contains a string, which it displays.
- Labels are passive controls that do not support any interaction with the user.
- **Label** defines the following constructors:
  1. `Label( )`
  2. `Label(String str)`
  3. `Label(String str, int how)`
- The first version creates a blank label.
- The second version creates a label that contains the string specified by *str*. This string is left-justified.
- The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

- Set or change the text in a label is done by using the **setText( )** method.
- Obtain the current label by calling **getText( )**.
- These methods are shown here:  
    void setText(String str)  
    String getText( )
- For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.
- To set the alignment of the string within the label by calling **setAlignment( )**.
- To obtain the current alignment, call **getAlignment( )**.
- The methods are as follows:  
    void setAlignment(int how)  
    int getAlignment( )

Label creation: Label one = new Label("One");

# button

- The most widely used control is the push button.
- A *push button* is a component that contains a label and that generates an event when it is pressed.
- Push buttons are objects of type **Button**. **Button** defines these two constructors:  
    Button( )  
    Button(String str)
- The first version creates an empty button. The second creates a button that contains *str* as a label.
- After a button has been created, you can set its label by calling **setLabel( )**.
- You can retrieve its label by calling **getLabel( )**.
- These methods are as follows:  
    void setLabel(String str)  
    String getLabel( )  
    Here, *str* becomes the new label for the button.

Button creation:      Button yes = new Button("Yes");

# canvas

- It is not part of the hierarchy for applet or frame windows
- **Canvas** encapsulates a blank window upon which you can draw.
- Canvas creation:  
    Canvas c = new Canvas();  
    Image test = c.createImage(200, 100);
- This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object.  
At this point, the image is blank.

# scrollbars

- Scrollbar generates adjustment events when the scroll bar is manipulated.
- Scrollbar creates a scroll bar control.
- *Scroll bars* are used to select continuous values between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts.
- Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar.
- The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value.

- **Scrollbar** defines the following constructors:
  - Scrollbar( )
  - Scrollbar(int style)
  - Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
- The first form creates a vertical scroll bar.
- The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal.
- In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*.
- The number of units represented by the height of the thumb is passed in *thumbSize*.
- The minimum and maximum values for the scroll bar are specified by *min* and *max*.
- `vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);`
- `horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);`



# text

- **Text is created by Using a TextField class**
- The **TextField** class implements a single-line text-entry area, usually called an *edit control*.
- Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:
  - TextField( )
  - TextField(int numChars)
  - TextField(String str)
  - TextField(String str, int numChars)

- The first version creates a default text field.
- The second form creates a text field that is *numChars* characters wide.
- The third form initializes the text field with the string contained in *str*.
- The fourth form initializes a text field and sets its width.
- **TextField** (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field.
- To obtain the string currently contained in the text field, call **getText()**.
- To set the text, call **setText( )**. These methods are as follows:  
    String getText( )  
    void setText(String str)  
    Here, *str* is the new string.

# components

- At the top of the AWT hierarchy is the **Component** class.
- **Component** is an abstract class that encapsulates all of the attributes of a visual component.
- All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**.
- It defines public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

- To add components

Component add(Component compObj)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned.

Once a control has been added, it will automatically be visible whenever its parent window is displayed.

- To remove a control from a window when the control is no longer needed call **remove( )**.
- This method is also defined by **Container**. It has this general form:

void remove(Component obj)

Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

# check box,

- A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- You can change the state of a check box by clicking on it.
- Check boxes can be used individually or as part of a group.
- Checkboxes are objects of the **Checkbox** class.

- **Checkbox** supports these constructors:
  1. `Checkbox( )`
  2. `Checkbox(String str)`
  3. `Checkbox(String str, boolean on)`
  4. `Checkbox(String str, boolean on, CheckboxGroup cbGroup)`
  5. `Checkbox(String str, CheckboxGroup cbGroup, boolean on)`
- The first form creates a check box whose label is initially blank. The state of the check box is unchecked.
- The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.
- The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared.
- The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

- To retrieve the current state of a check box, call **getState( )**.
- To set its state, call **setState( )**.
- To obtain the current label associated with a check box by calling **getLabel( )**.
- To set the label, call **setLabel( )**.
- These methods are as follows:

boolean getState( )

void setState(boolean on)

String getLabel( )

void setLabel(String str)

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared.

Checkbox creation:

```
CheckBox Win98 = new Checkbox("Windows 98", null, true);
```

# check box groups

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called *radio buttons*.
- To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes.
- Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.
- To determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**.
- To set a check box by calling **setSelectedCheckbox( )**.
- These methods are as follows:  
    Checkbox getSelectedCheckbox( )  
    void setSelectedCheckbox(Checkbox which)  
    Here, *which* is the check box that you want to be selected. The previously selected checkbox will be turned off.
  - CheckboxGroup cbg = new CheckboxGroup();
  - Win98 = new Checkbox("Windows 98", cbg, true);
  - winNT = new Checkbox("Windows NT", cbg, false);



# choices

- The **Choice** class is used to create a *pop-up list* of items from which the user may choose.
- A **Choice** control is a form of menu.
- **Choice** only defines the default constructor, which creates an empty list.
- To add a selection to the list, call **addItem( )** or **add( )**.  
    void addItem(String name)  
    void add(String name)
- Here, *name* is the name of the item being added.
- Items are added to the list in the order to determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**.  
    String getItemSelected()  
    int getSelectedIndex( )

# lists

- The **List** class provides a compact, multiple-choice, scrolling selection list.
- **List** object can be constructed to show any number of choices in the visible window.
- It can also be created to allow multiple selections. **List** provides these constructors:

List( )

List(int numRows)

List(int numRows, boolean multipleSelect)

- To add a selection to the list, call **add( )**. It has the following two forms:

void add(String name)

void add(String name, int index)

- Ex: List os = new List(4, true);

# panels

- The **Panel** class is a concrete subclass of **Container**.
- It doesn't add any new methods; it simply implements **Container**.
- A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**.
- When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.
- **Panel** is a window that does not contain a title bar, menu bar, or border.
- Components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.
- Ex: 

```
Panel osCards = new Panel();
 CardLayout cardLO = new CardLayout();
 osCards.setLayout(cardLO);
```

# scrollpane

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed.
- Horizontal and/or vertical scroll bars may be provided if necessary.
- constants are defined by the **ScrollPaneConstants** interface.
  1. HORIZONTAL\_SCROLLBAR\_ALWAYS
  2. HORIZONTAL\_SCROLLBAR\_AS\_NEEDED
  3. VERTICAL\_SCROLLBAR\_ALWAYS
  4. VERTICAL\_SCROLLBAR\_AS\_NEEDED

# dialogs

- Dialog class creates a dialog window.
- constructors are :
  - Dialog(Frame parentWindow, boolean mode)
  - Dialog(Frame parentWindow, String title, boolean mode)
- The dialog box allows you to choose a method that should be invoked when the button is clicked.
- Ex:

```
Font f = new Font("Dialog", Font.PLAIN, 12);
```

# menubar

- MenuBar class creates a menu bar.
- A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu.
- To create a menu bar, first create an instance of **MenuBar**.
- This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar.
- Following are the constructors for **Menu**:
  - Menu( )
  - Menu(String optionName)
  - Menu(String optionName, boolean removable)

- Once you have created a menu item, you must add the item to a **Menu** object by using `MenuItem add(MenuItem item)`
- Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place.
- Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:
- `Menu add(Menu menu)`

# Graphics

- The AWT supports a rich assortment of graphics methods.
- All graphics are drawn relative to a window.
- A *graphics context* is encapsulated by the **Graphics** class
- It is passed to an applet when one of its various methods, such as `paint( )` or `update( )`, is called.
- It is returned by the `getGraphics( )` method of `Component`.
- The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled.
- Objects are drawn and filled in the currently selected graphics color, which is black by default.
- When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped
- Ex:

```
Public void paint(Graphics g)
{
 G.drawString("welcome",20,20);
}
```



# Layout manager

- A layout manager automatically arranges your controls within a window by using some type of algorithm.
- 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

- Border Layout
- Grid Layout
- Flow Layout
- Card Layout
- GridBag Layout

# 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**:
  - `BorderLayout( )`
  - `BorderLayout(int horz, int vert)`
- **BorderLayout** defines the following constants that specify the regions:
  - `BorderLayout.CENTER`
  - `BorderLayout.SOUTH`
  - `BorderLayout.EAST`
  - `BorderLayout.WEST`
  - `BorderLayout.NORTH`
- Components can be added by
  - `void add(Component compObj, Object region);`

# Grid layout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a
- **GridLayout**, you define the number of rows and columns. The constructors are
  - `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-length rows.

# 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

# 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.
- **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)

# GridBag Layout

- The Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
- **GridLayout** lays out components in a two-dimensional grid.
- The constructors are
  - GridLayout( )
  - GridLayout(int numRows, int numColumns )
  - GridLayout(int numRows, int numColumns, int horz, int vert)

# 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

- Java can be used to create two types of programs: applications and applets.
- An *application* is a program that runs on your computer, under the operating system of that Computer(i.e an application created by Java is more or less like one created using C or C++).
- When used to create applications, Java is not much different from any other computer language.
- An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.
- An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip.

- The important difference is that an applet is an *intelligent program*, not just an animation or media file(i.e an applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over.
- Applications require main method to execute.
- Applets do not require main method.
- Java's console input is quite limited
- Applets are graphical and window-based.

# 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( )**

- **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.

- **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

- 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);
}
}
```

# Introduction to swings

- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
- Even familiar components such as buttons have more capabilities in Swing.
- For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- Unlike AWT components, Swing components are not implemented by platform-specific code.
- Instead, they are written entirely in Java and, therefore, are platform-independent.
- The term *lightweight* is used to describe such elements.

- The Swing component are defined in **javax.swing**
  1. AbstractButton: Abstract superclass for Swing buttons.
  2. ButtonGroup: Encapsulates a mutually exclusive set of buttons.
  3. ImageIcon: Encapsulates an icon.
  4. JApplet: The Swing version of Applet.
  5. JButton: The Swing push button class.
  6. JCheckBox: The Swing check box class.
  7. JComboBox : Encapsulates a combo box (an combination of a drop-down list and text field).
  8. JLabel: The Swing version of a label.
  9. JRadioButton: The Swing version of a radio button.
  - 10.JScrollPane: Encapsulates a scrollable window.
  - 11.JTabbedPane: Encapsulates a tabbed window.
  - 12.JTable: Encapsulates a table-based control.
  - 13.JTextField: The Swing version of a text field.
  - 14.JTree: Encapsulates a tree-based control.

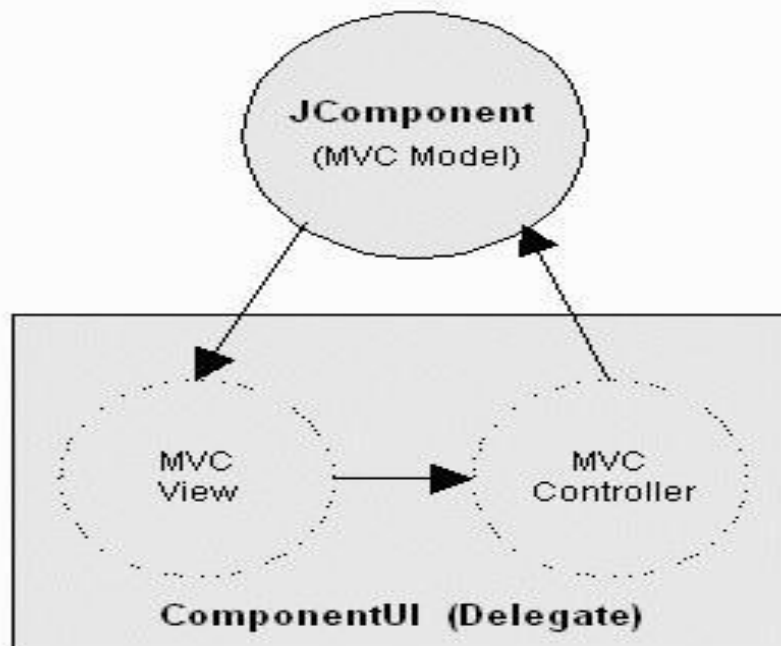
# Limitations of AWT

- 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.

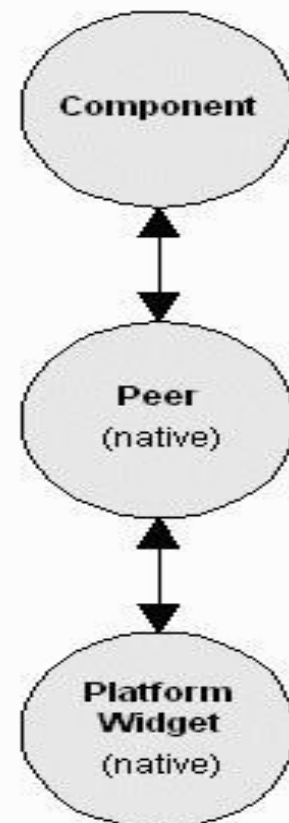
- Lowest Common Denominator
  - If not available natively on one Java platform, not available on any Java platform
- Simple Component Set
- Components Peer-Based
  - Platform controls component appearance
  - Inconsistencies in implementations
    - Interfacing to native platform error-prone



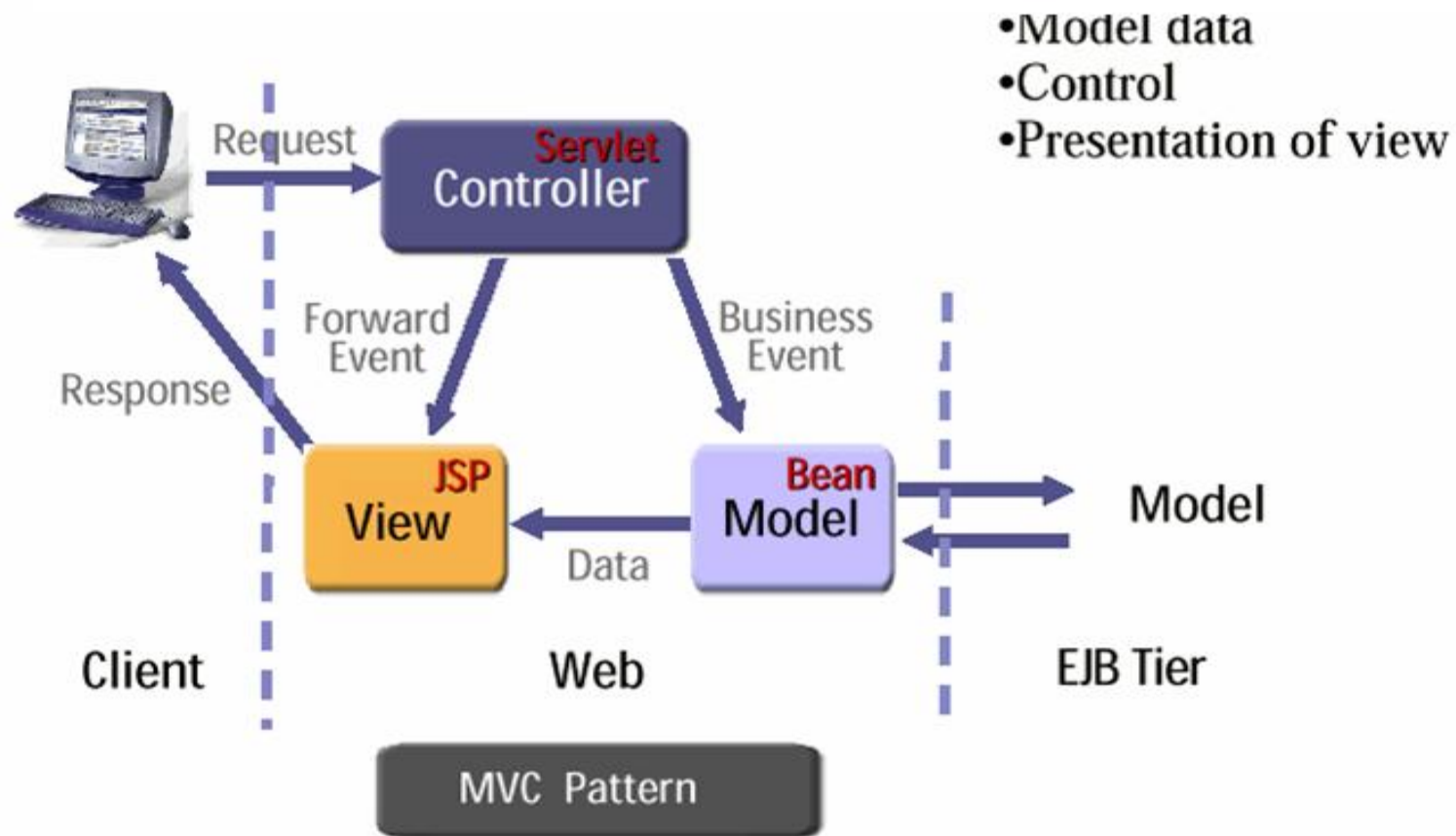
## Swing Look & Feel



## AWT Look & Feel



# MODEL VIEW CONTROLLER ARCHITECTURE



# Model

- Model consists of data and the functions that operate on data
- Java bean that we use to store data is a model component
- EJB can also be used as a model component

# view

- View is the front end that user interact.
- View can be a

HTML

JSP

Struts ActionForm

# Controller

- Controller component responsibilities
  1. Receive request from client
  2. Map request to specific business operation
  3. Determine the view to display based on the result of the business operation

# components

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

# Components (contd...)

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

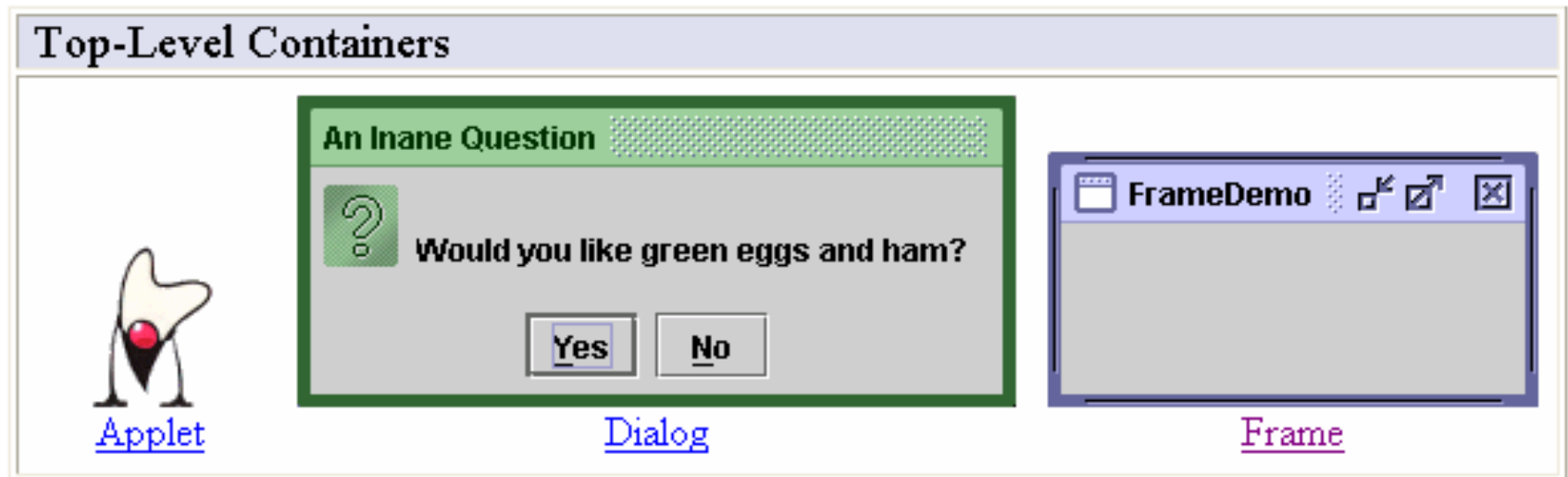
# Components (contd...)

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



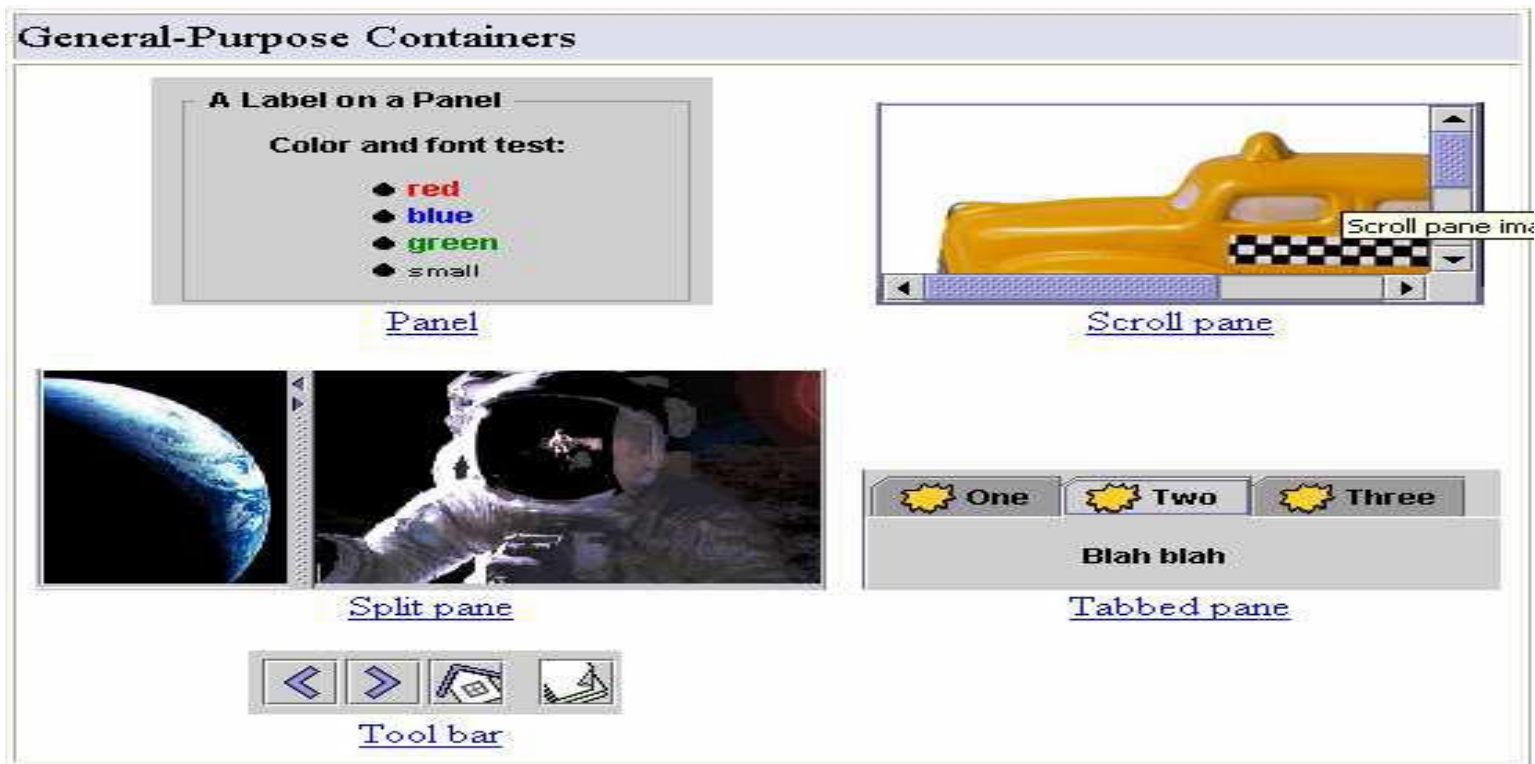
# Containers

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



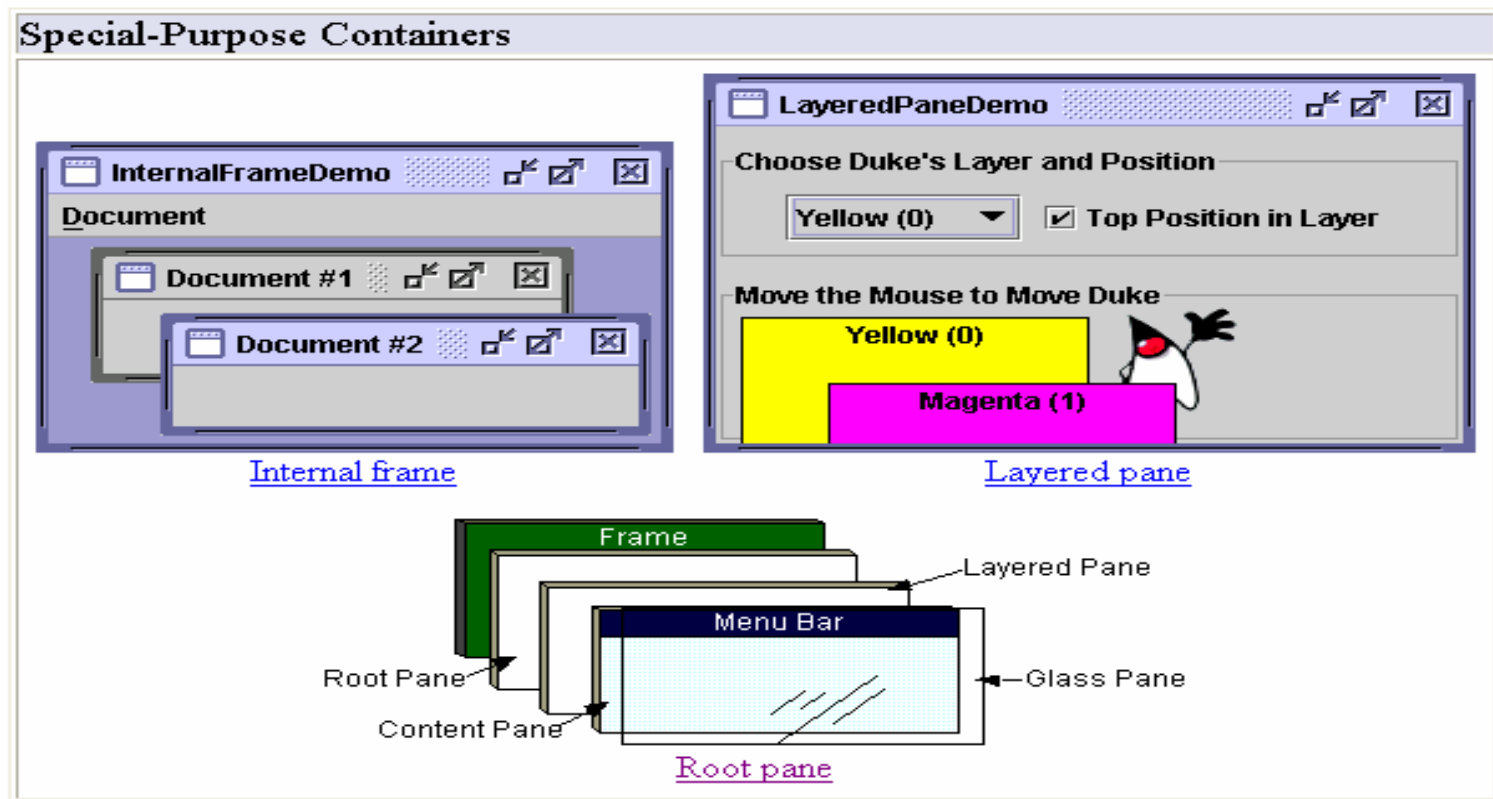
# General Purpose Containers

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



# Special Purpose Container

- Intermediate containers that play specific roles in the UI.

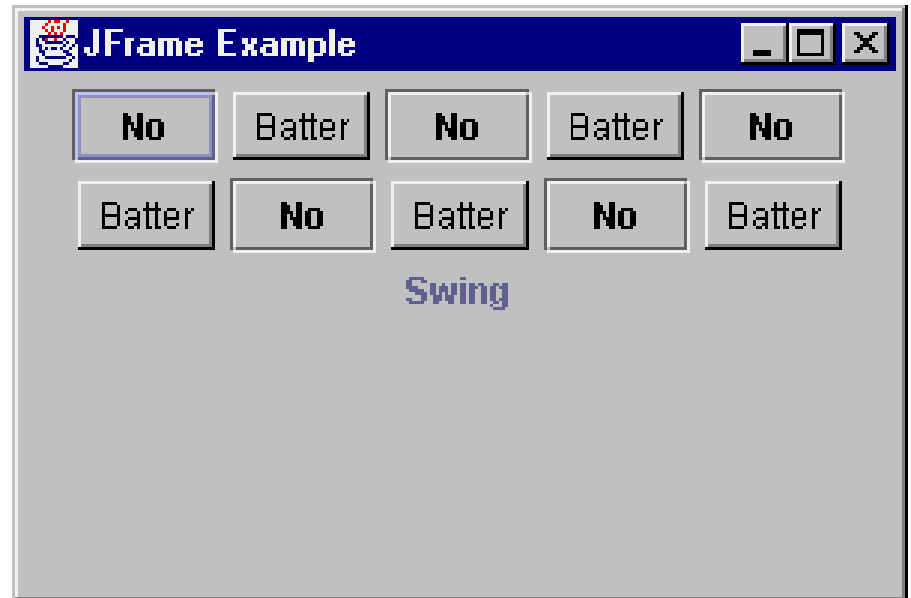


# Exploring swing- JApplet,

- 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*

# 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

# Icons and Labels

- In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image.
- constructors are:
  - `ImageIcon(String filename)`
  - `ImageIcon(URL url)`
- The **ImageIcon** class implements the **Icon** interface that declares the methods
  1. `int getIconHeight( )`
  2. `int getIconWidth( )`
  3. `void paintIcon(Component comp,Graphics g,int x,int y)`

- Swing labels are instances of the **JLabel** class, which extends **JComponent**.
- It can display text and/or an icon.
- Constructors are:
  - JLabel(Icon i)
  - Label(String s)
  - 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.



# Text fields

- The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**.
- It provides functionality that is common to Swing text components.
- One of its subclasses is **JTextField**, which allows you to edit one line of text.
- Constructors are:
  - `JTextField( )`
  - `JTextField(int cols)`
  - `JTextField(String s, int cols)`
  - `JTextField(String s)`
- Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

# 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 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.

# Check boxes

- The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**.
- Some of its constructors are shown here:
  - JCheckBox(Icon i)
  - JCheckBox(Icon i, boolean state)
  - JCheckBox(String s)
  - JCheckBox(String s, boolean state)
  - JCheckBox(String s, Icon i)
  - JCheckBox(String s, Icon i, boolean state)
- Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not.
- The state of the check box can be changed via the following method:
  - void setSelected(boolean state)
- Here, *state* is **true** if the check box should be checked.

# Combo boxes

- Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.
- A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.
- Two of **JComboBox**'s constructors are :
  - JComboBox( )
  - JComboBox(Vector v)
- Here, *v* is a vector that initializes the combo box.
- Items are added to the list of choices via the **addItem( )** method, whose signature is:
  - void addItem(Object obj)
- Here, *obj* is the object to be added to the combo box.

# Radio Buttons

- Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**.
- Some of its constructors are :
  - JRadioButton(Icon i)
  - JRadioButton(Icon i, boolean state)
  - JRadioButton(String s)
  - JRadioButton(String s, boolean state)
  - JRadioButton(String s, Icon i)
  - JRadioButton(String s, Icon i, boolean state)
- Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.
- Elements are then added to the button group via the following method:
  - void add(AbstractButton ab)
- Here, *ab* is a reference to the button to be added to the group.

# Tabbed Panes

- A *tabbed pane* is a component that appears as a group of folders in a file cabinet.
- Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time.
- Tabbed panes are commonly used for setting configuration options.
- Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

void addTab(String str, Component comp)

- Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.
- The general procedure to use a tabbed pane in an applet is outlined here:
  1. Create a **JTabbedPane** object.
  2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the  
title of the tab and the component it contains.)
  3. Repeat step 2 for each tab.
  4. Add the tabbed pane to the content pane of the applet.

# Scroll Panes

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
- Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are :
  - JScrollPane(Component comp)
  - JScrollPane(int vsb, int hsb)
  - JScrollPane(Component comp, int vsb, int hsb)
- Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are **int** constants that define when vertical and horizontal scroll bars for this scroll pane are shown.
- These constants are defined by the **ScrollPaneConstants** interface.
  1. HORIZONTAL\_SCROLLBAR\_ALWAYS
  2. HORIZONTAL\_SCROLLBAR\_AS\_NEEDED
  3. VERTICAL\_SCROLLBAR\_ALWAYS
  4. VERTICAL\_SCROLLBAR\_AS\_NEEDED
- Here are the steps to follow to use a scroll pane in an applet:
  1. Create a **JComponent** object.
  2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
  3. Add the scroll pane to the content pane of the applet.



# Trees

- Data Model - `TreeModel`
  - default: `DefaultTreeModel`
  - `getChild`, `getChildCount`, `getIndexOfChild`, `getRoot`, `isLeaf`
- Selection Model - `TreeSelectionModel`
- View - `TreeCellRenderer`
  - `getTreeCellRendererComponent`
- Node - `DefaultMutableTreeNode`

# Tables

- A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position.
- Tables are implemented by the **JTable** class, which extends **JComponent**.
- One of its constructors is :  
    JTable(Object data[ ][ ], Object colHeads[ ])
- Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- Here are the steps for using a table in an applet:
  1. Create a **JTable** object.
  2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
  3. Add the table to the scroll pane.
  4. Add the scroll pane to the content pane of the applet.