

*Welcome to:*  
**Core Java**

# **Course Description**

---

- This course will teach fundamental of java allowing programmers to gain detailed java knowledge

*Welcome to*

# **Basic Java Language**

# Unit Objective

---

- After completing this unit you should be able to:
  - Object oriented concepts
  - Describe the role of Java Virtual Machine
  - Describe Java Development Cycle.
  - Describe components of class in Java
  - Describe encapsulation and class instantiation and their benefits to java programming
  - Identify keywords, operators, and primitive data types
  - Differentiate between implicit and explicit casting.
  - Use conditional and iteration statement
  - Create and use arrays.

# Basic Principles of Object Orientation

---

- **Abstraction:**

- A model that includes most important aspects of a given problem while ignoring less important details

- **Encapsulation:-**

- Hide implementation from clients

# **Contn...**

---

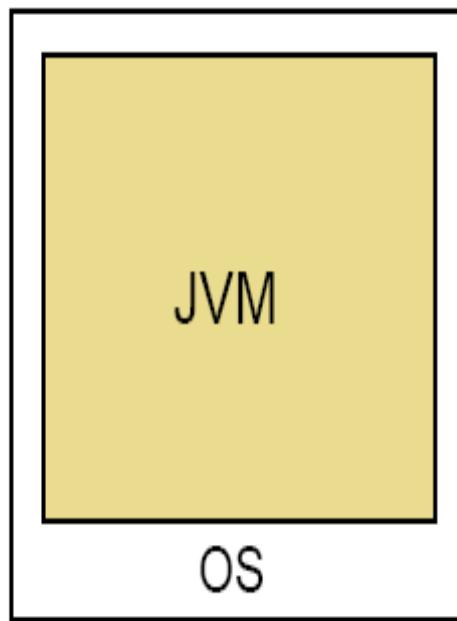
- Inheritance : -**

- A process by which one object acquires the properties of parent.

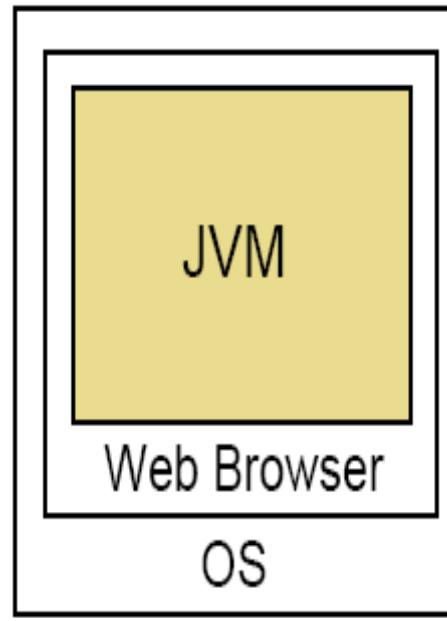
- Polymorphism:-**

- The ability to hide many different implementations behind a single interface.

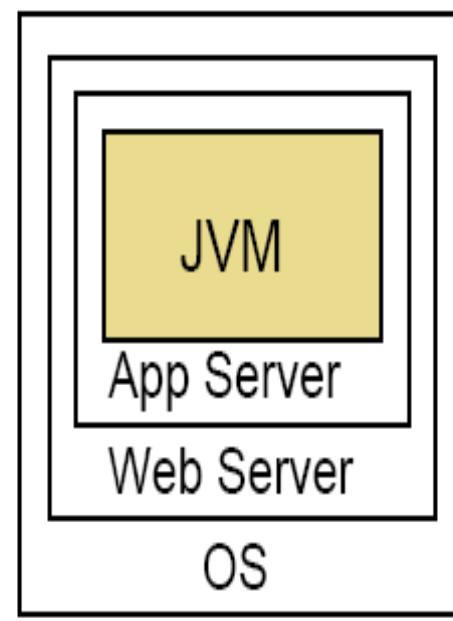
# Java Virtual Machine



Application:  
Stand-alone JVM

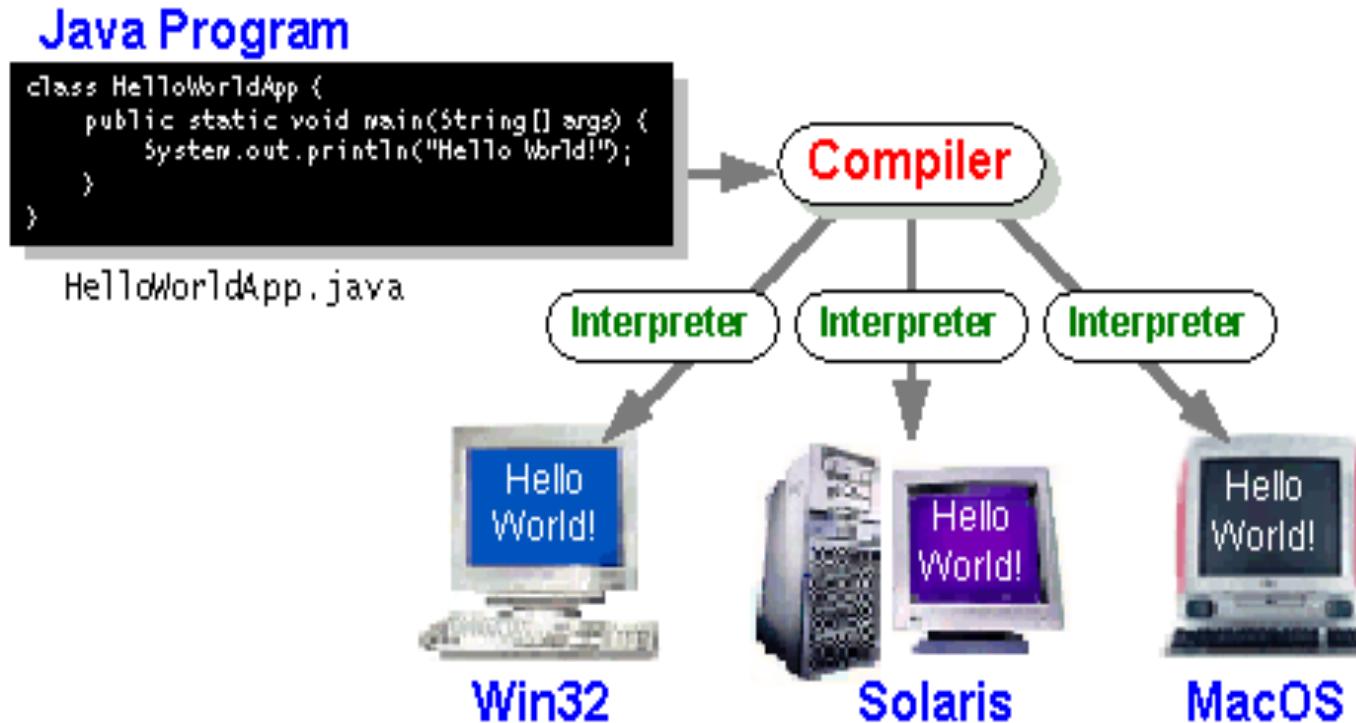


Applet:  
JVM Embedded  
In Web Browser

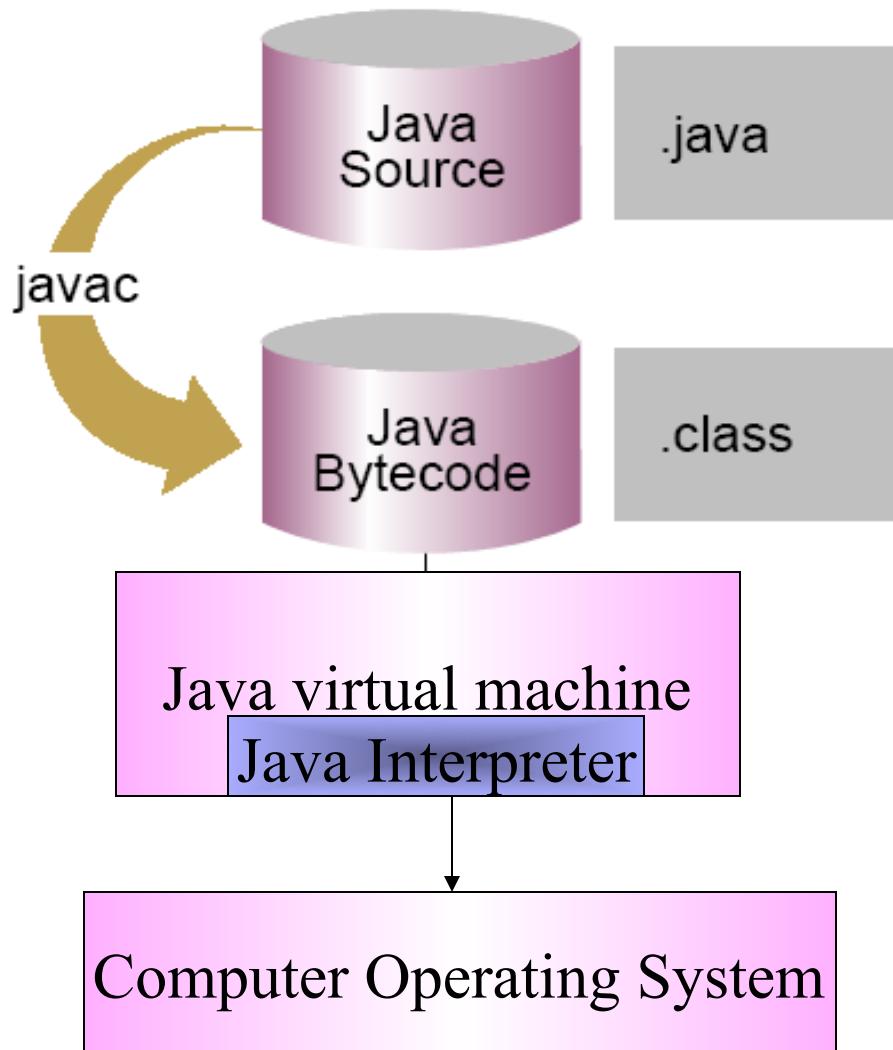


Servlet:  
JVM Embedded  
In App Server

# Java Life Cycle



# Java Development Cycle



# **Java Timeline - 10 years of Java.**

---

**1995**

- **March 23, 1995**
  - Java is born. Father: James Goslings, Mother: Sun Microsystems.
- **May 23, 1995**
  - Java Technology officially announced in SunWorld.

# **Java Timeline - 10 years of Java.**

---

**1996**

**January 23, 1996**

- JDK 1.0 release day May 29, 1996
- First JavaOne developer conference, JavaBeans, Servlets and other technologies announced. October 25, 1996
- First Just-In-Time (JIT) compiler announced. October 29, 1996
- Java Card API announced. December 9, 1996
- JDK 1.1 beta software released

# Java Timeline - 10 years of Java.

---

## January 11, 1997

- JavaBeans Development Kit released.February 18, 1997
- JDK 1.1 released.March 4, 1997
- Java Servlet Developers Kit released March 10, 1997
- JNDI API introduced.April 2, 1997
- Enterprise JavaBeans (EJB) technology announced Java Foundation Classes (JFC) technology included in Java platform

# Java Timeline - 10 years of Java.

---

## 1998

- **March, 1998**

- JFC/"Project Swing" shipped March 24, 1998
- Java Jumpstart product announced Embedded Java spec posted for public review April 20, 1998
- Java Plug-in™ product shipped June 3, 1998
- Visa launches world's first smart card based on Java Card technology July 22, 1998
- Java Blend 1.0 software shipped November 5, 1998
- Java 2 software port to Linux Embedded Java spec complete December 8, 1998
- Java 2 platform shipped Java Community Process (JCP) program formalized

# Java Timeline - 10 years of Java.

---

**1999**

- **January 25, 1999**
  - Jini technology announced< h4>Personal Java 3.0 platform ships **February 24, 1999**
  - Java 2 platform source code released **March 4, 1999**
  - XML support for Java platform unveiled **March 27, 1999**
  - Java HotSpot performance engine unveiled **June 2, 1999**
  - JavaServer Pages technology unveiled **June 15, 1999**
  - Three editions of Java platform: J2SE, J2EE, J2ME announced **December 8, 1999**
  - J2EE platform shipped J2SE platform on Linux ships

# Java Timeline - 10 years of Java.

---

## 2000

- February 29, 2000
- Java API for XML optional package ships May 8, 2000
- J2SE v. 1.3 platform released May 17, 2000
- J2SE v 1.3 platform for Apple Mac OS X

## 2001

- February 26, 2001
- J2EE Connector Architecture announced March 8, 2001
- [J2EE Patterns Catalog](#) released March 14, 2001
- Java Web Start 1.0 released April 2001
- Java 2 Platform, Enterprise Edition (J2EE) SDK 1.3 beta released (EJB 2.0, JSP 1.2, servlet 2.3)

# **Java Timeline - 10 years of Java.**

---

## **2002**

- January 28, 2002
- Java Web Services Developer Pack (WSDP), Early Access Release 1 December 2002
- J2EE 1.4 Beta released

## **2004**

- October 1, 2004
- Tiger (Java 5.0) released!

## **2005**

- March 23, 2005
- Java 10th Anniversary.

# Java Keywords

---

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

# Primitive types

---

- There are exactly eight primitive data types in Java
- Four of them represent integers:
  - byte, short, int, long
- Two of them represent floating point numbers:
  - float, double
- One of them represents characters:
  - char
- And one of them represents boolean values:
  - boolean

# Primitive Data Types

- Each type has a default value
- Size is platform independent

| Type Name        | Size (in bits) |
|------------------|----------------|
| Integrals:       |                |
| byte             | 8              |
| short            | 16             |
| int              | 32             |
| long             | 64             |
| Floating Points: |                |
| float            | 32             |
| double           | 64             |
| Characters:      |                |
| char             | 16             |
| Booleans:        |                |
| boolean          | N/A            |

# Primitive Data Types

| Type Name              | Size    | Range  | Default Values |
|------------------------|---------|--|----------------|
| Integers               |         |  |                |
| <i>byte</i>            | 8 bits  | -128 to 127 (-2 <sup>7</sup> to 2 <sup>7</sup> -1)   | 0              |
| <i>short</i>           | 16 bits | -32,768 to 32,767 (-2 <sup>15</sup> to 2 <sup>15</sup> -1)                                       | 0              |
| <i>int</i>             | 32 bits | -2,147,483,648 to 2,147,483,647 (-2 <sup>31</sup> to 2 <sup>31</sup> -1)                         | 0              |
| <i>long</i>            | 64 bits | $\sim -9.2 \times 10^{18}$ to $\sim 9.2 \times 10^{18}$ (-2 <sup>63</sup> to 2 <sup>63</sup> -1) | 0              |
| Floating-Point Numbers |         |  |                |
| <i>float</i>           | 32 bits | $\sim -3.4 \times 10^{38}$ to $\sim 3.4 \times 10^{38}$  | 0.0f           |
| <i>double</i>          | 64 bits | $\sim -1.8 \times 10^{308}$ to $\sim 1.8 \times 10^{308}$  | 0.0d           |
| Character              |         |  |                |
| <i>char</i>            | 16 bits | 0 to 65,535  | '\u0000'       |
| Boolean                |         |  |                |
| <i>boolean</i>         | N/A     | false or true  | false          |

# Literal Values

---

| Literal | Data Type |
|---------|-----------|
| 178     | int       |
| 8864L   | long      |
| 37.266  | double    |
| 37.266D | double    |
| 87.363F | float     |
| 26.77e3 | double    |
| 'c'     | char      |
| true    | Boolean   |
| false   | boolean   |

# Variables

- Variable is a name for a location in memory
- 2 types of variables
  - Primitive
  - Reference
- Must have a type
- Must have a name (Identifier)
  - Starts with a letter, underscore (\_), or dollar sign (\$)
  - Cannot be a reserved word (public, void, static, int, ...)

The diagram illustrates the structure of a variable declaration. It shows the code `int total = 100;` with three annotations pointing to its parts:

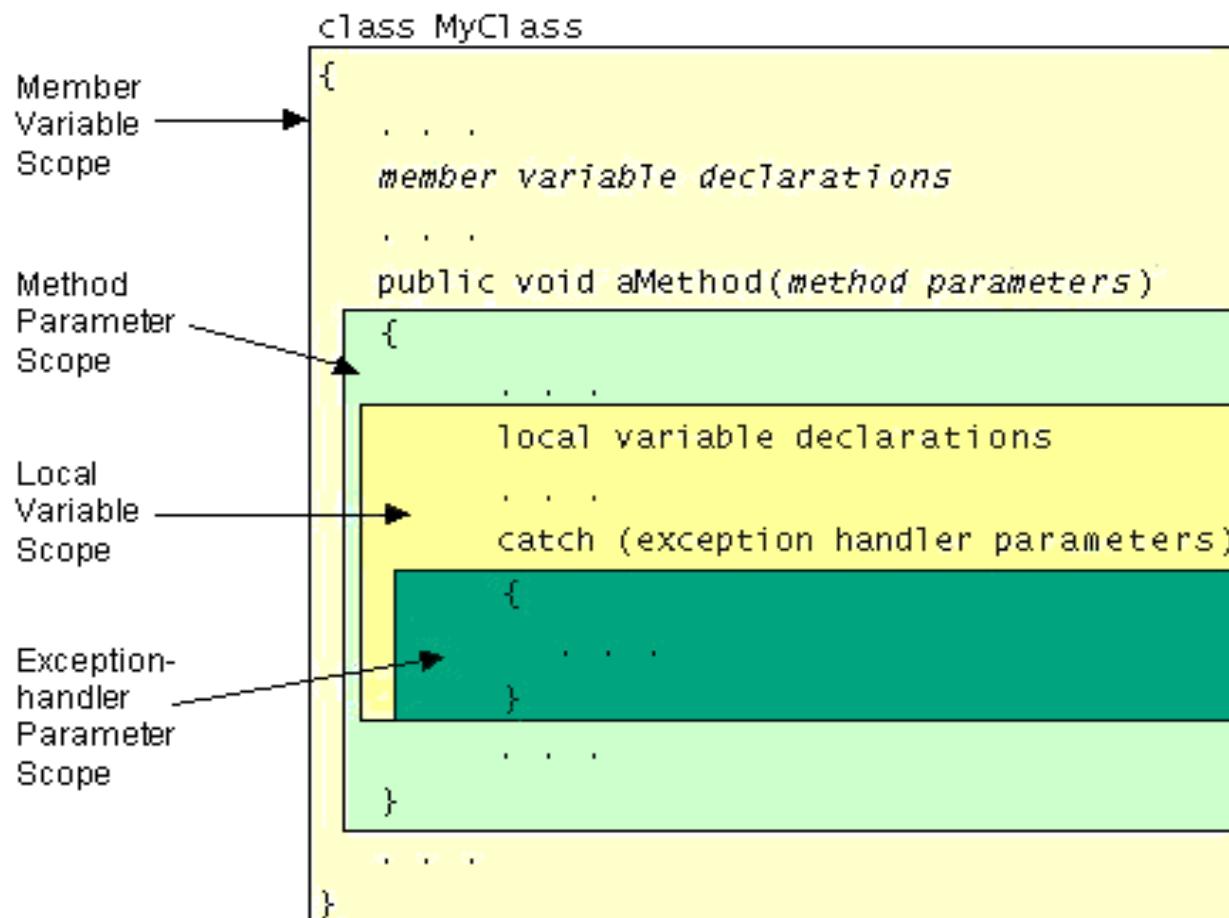
- A blue arrow points from the text "data type" to the word `int`.
- A blue arrow points from the text "variable name" to the word `total`.
- A blue arrow points from the text "Initial Value (Optional)" to the number `100`.

# Variable Names

---

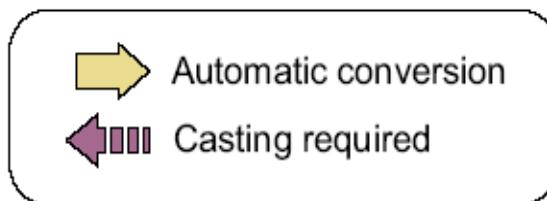
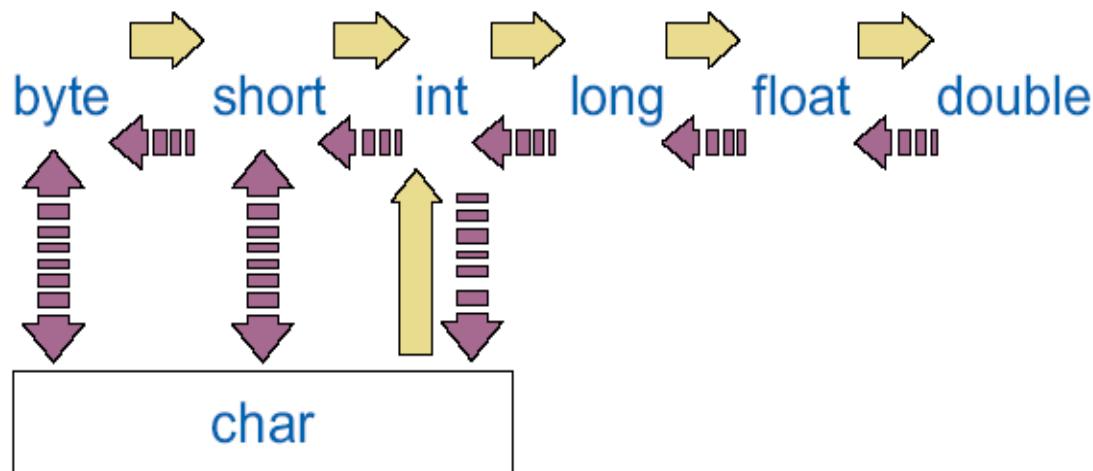
- Must be legal java identifier
- It must not be a keyword, boolean literal or reserved word null.
- It must be unique within its scope.

# Scope



# Type conversion of primitive data types

- Java implicitly cast to longer data types
- When placing larger to smaller types, you must use explicit casting to mention type name to which you are converting.



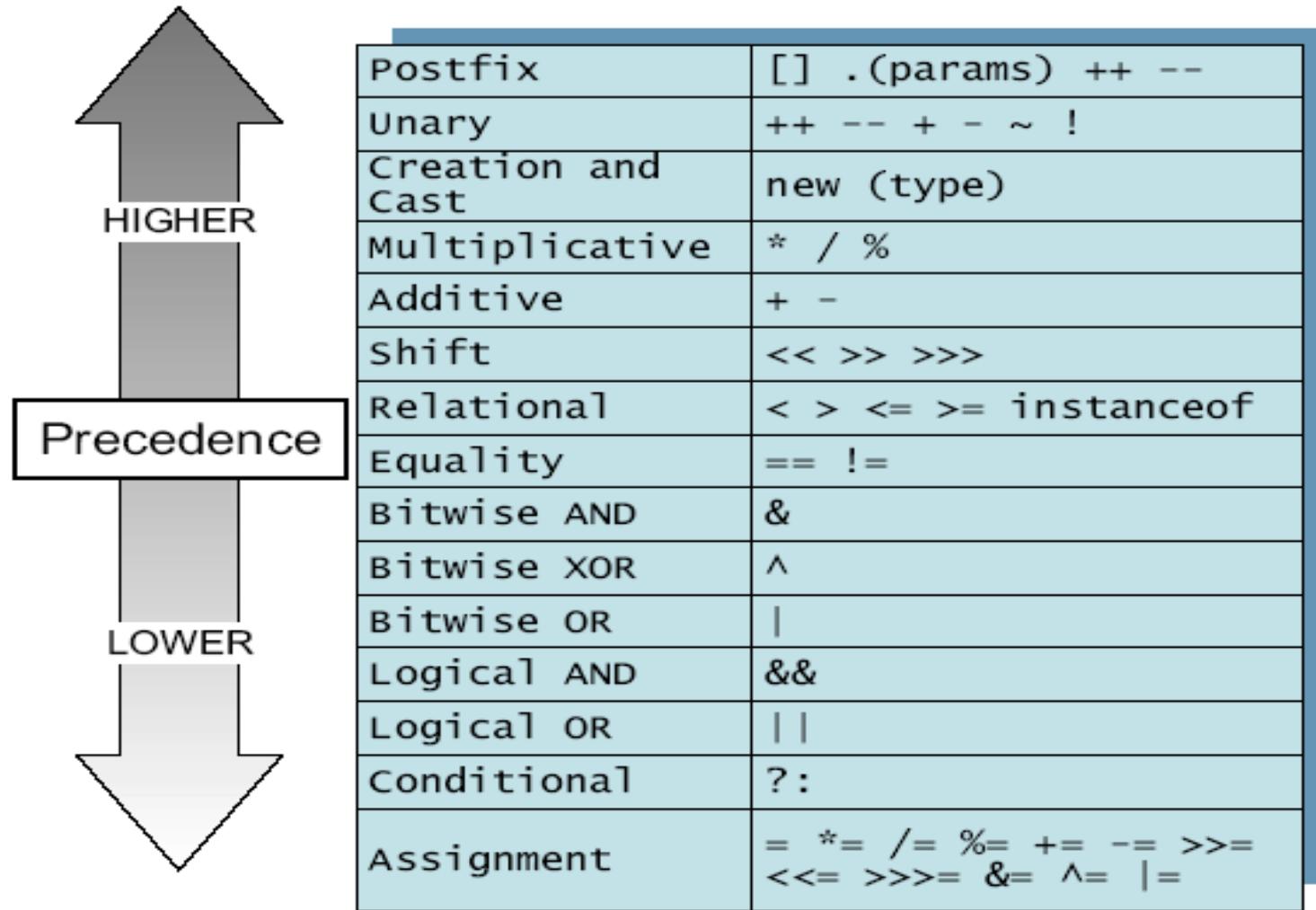
# Automatic casting and conversion of primitives

- The rules governing the automatic casting by the Java compiler are as follows: when considering two operands in arithmetic expression:
  - If either type is double, other is cast to a double
  - If either type is float, other is cast to float.
  - If either type is long, other is cast to long.
  - Otherwise both the operands are converted to int.

# Conversion Example

```
int i, j;  
long x;  
float f;  
double d;  
// ...  
i = j; // exact match, no conversion needed  
x = i; // OK  
i = x; // NO, information could be lost  
i = (int) x; // allowed, explicit cast  
d = i; // OK  
f = d; // NO
```

# Operators and Precedence



# The increment operator

---

- `++` adds 1 to a variable
  - It can be used as a statement by itself, or within an expression
  - It can be put *before* or *after* a variable
  - If before a variable (preincrement), it means to add one to the variable, then use the result
  - If put after a variable (postincrement), it means to use the current value of the variable, then add one to the variable

# Examples of `++`

---

```
int a = 5;
```

```
a++;
```

*// a is now 6*

```
int b = 5;
```

```
++b;
```

*// b is now 6*

```
int c = 5;
```

```
int d = ++c;
```

*// c is 6, d is 6*

```
int e = 5;
```

```
int f = e++;
```

*// e is 6, f is 5*

```
int x = 10;
```

```
int y = 100;
```

```
int z = ++x + y++;
```

*// x is 11, y is 101, z is 111*

# The decrement operator

---

- `--` subtracts 1 from a variable
  - It can be used as a statement by itself, or within an expression
  - It can be put *before* or *after* a variable
  - If before a variable (predecrement), it means to subtract one from the variable, then use the result
  - If put after a variable (postdecrement), it means to use the current value of the variable, then subtract one from the variable

# Examples of ..

---

```
int a = 5;
```

```
a--;
```

```
// a is now 4
```

```
int b = 5;
```

```
--b;
```

```
// b is now 4
```

```
int c = 5;
```

```
int d = --c;
```

```
// c is 4, d is 4
```

```
int e = 5;
```

```
int f = e--;
```

```
// e is 4, f is 5
```

```
int x = 10;
```

```
int y = 100;
```

```
int z = --x + y--;
```

```
// x is 9, y is 99, z is 109
```

# Relational operators

---

> greater than

>= greater than or equal to

< less than

<= less than or equal to

== equal to

!= not equal to

# Logical Operators

---

- Boolean expressions can use the following *logical operators*:

!        Logical NOT

& &      Logical AND

||        Logical OR

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

# Short Circuited Operators

---

- Conditions can use logical operators to form complex expressions
- The processing of logical AND and logical OR is “short-circuited”
- If the left operand is sufficient to determine the result, the right operand is not evaluated

```
if (count != 0 && total/count > MAX)
    System.out.println ("Testing...");
```

This type of processing must be used carefully

# Operators Examples

```
int i =0;  
i++; // this increments the value of i by 1  
  
if (i == 0) { //this will test if the value of i is 0  
}  
int a[] = {1, 2, 3, 4, 5};  
a.length; //uses the member of operator .  
  
i += 85; // increments i's value by 85  
  
if ( (i == 0) && (a.length != 5) ) { // && is and  
}  
if ( (i == 0) || (a.length != 5) ) { // || is or  
}
```

# A problem with simple variables

---

- One variable holds one value
  - The value may change over time, but at any given time, a variable holds a single value
- If you want to keep track of many values, you need many variables
- All of these variables need to have names
- What if you need to keep track of hundreds or thousands of values?

# Multiple values

- An array lets you associate one name with a fixed (but possibly large) number of values
- All values must have the same type
- The values are distinguished by a numerical index between 0 and array size minus 1

|         |    |    |   |    |    |     |     |    |   |   |
|---------|----|----|---|----|----|-----|-----|----|---|---|
|         | 0  | 1  | 2 | 3  | 4  | 5   | 6   | 7  | 8 | 9 |
| myArray | 12 | 43 | 6 | 83 | 14 | -57 | 109 | 12 | 0 | 6 |

# Indexing into arrays

---

- To reference a single array element, use
  - *array-name [ index ]*
- Indexed elements can be used just like simple variables
  - You can access their values
  - You can modify their values
- An array index is sometimes called a subscript

# Using array elements

|         | 0  | 1  | 2 | 3  | 4  | 5   | 6   | 7  | 8 | 9 |
|---------|----|----|---|----|----|-----|-----|----|---|---|
| myArray | 12 | 43 | 6 | 83 | 14 | -57 | 109 | 12 | 0 | 6 |

- Examples:
  - `x = myArray[1];` // sets x to 43
  - `myArray[4] = 99;` // replaces 14 with 99
  - `m = 5;`
  - `y = myArray[m];` // sets y to -57
  - `z = myArray[myArray[9]];` // sets z to 109

# Array values

---

- An array may hold *any* type of value
- All values in an array must be the *same* type
  - For example, you can have:
    - an array of integers
    - an array of Strings
    - an array of Person
    - an array of arrays of String
    - an array of Object

# Declaration versus definition

---

- Arrays are objects
- Creating arrays is like creating other objects:
  - the *declaration* only provides type information
  - the `new` *definition* actually allocates space
  - declaration and definition may be separate or combined
- Example for ordinary objects:

`Person p;` // declaration

`p = new Person("John");` // definition

`Person p = new Person("John");` // combined

# Declaring and defining arrays

- Example for array objects:

- `int myArray[ ];` // declaration

- This *declares* `myArray` to be an array of integers

- Notice that the size is *not* part of the type

- `myArray = new int[10];` // definition

- `new int[10]` creates the array

- The rest is an ordinary assignment statement

- `int myArray[ ] = new int[10];` // both

# Two ways to declare arrays

---

- You can declare more than one variable in the same declaration:

```
int a[ ], b, c[ ], d; // notice position of brackets
```

- **a** and **c** are **int** arrays
- **b** and **d** are just **ints**

- Another syntax:

```
int [ ] a, b, c, d; // notice position of brackets
```

- **a**, **b**, **c** and **d** are **int** arrays
  - When the brackets come before the first variable, they apply to *all* variables in the list

# Array assignment

---

- Array assignment is object assignment
- Object assignment does *not* copy values

```
Person p1; Person p2;
```

```
p1 = new Person("John");
```

```
p2 = p1; // p1 and p2 refer to the same person
```

- Array assignment does *not* copy values

```
int a1[ ]; int a2[ ];
```

```
a1 = new int[10];
```

```
a2 = a1; // a1 and a2 refer to the same array
```

# An array's size is *not* part of its type

---

- When you declare an array, you declare its type; you *must not* specify its size
  - Example: `String names[ ];`
- When you define the array, you allocate space; you *must* specify its size
  - Example: `names = new String[50];`
- This is true even when the two are combined
  - Example: `String names[ ] = new String[50];`

# Array assignment

---

- When you assign an array value to an array variable, the types must be compatible
- The following is *not* legal:

```
double dub[ ] = new int[10]; // illegal
```

- The following *is* legal:

```
int myArray[ ] = new int[10];
```

- ...and later in the program,

```
myArray = new int[500]; // legal!
```

- Legal because array *size* is not part of its *type*

# Example of array use I

---

- Suppose you want to find the largest value in an array **scores** of 10 integers:

```
int largestScore = 0;  
  
for (int i = 0; i < 10; i++)  
  
    if (scores[i] > largestScore)  
  
        largestScore = scores[i];
```

- By the way, do you see an error in the above program?
  - What if all values in the array are negative?

# Example of array use II

---

- To find the largest value in an array scores of 10 (possibly negative) integers:

```
int largestScore = scores[0];
for (int i = 1; i < 10; i++)
    if (scores[i] > largestScore)
        largestScore = scores[i];
```

# Length of an array

---

- Arrays are objects
- Every array has an instance constant, `length`, that tells how large the array is
- Example:

```
for (int i = 0; i < scores.length; i++)  
    System.out.println(scores[i]);
```

- Use of `length` is always preferred over using a constant such as 10
- Strings have a `length()` method!

# Arrays of objects

---

- Suppose you declare and define an array of objects:
  - `Person[ ] people = new Person[20];`
- There is nothing wrong with this array, but
  - it has 20 *references* to Persons in it
  - all of these references are initially null
  - you have *not yet* defined 20 Persons
  - For example, `people[12].name` will give you a `nullPointerException`

# Initializing arrays I

- Here's one way to initialize an array of objects

```
Person people[] = new Person[20];  
for (int i = 0; i < people.length; i++) {  
    people[i] = new Person("John");  
}
```

- This approach has a slight drawback: all the array elements have similar values

# Initializing arrays II

---

- There is a special syntax for giving initial values to the elements of arrays
  - This syntax can be used in place of `new type[size]`
  - It can *only* be used in an *array declaration*
  - The syntax is: `{ value, value, ..., value }`
- Examples:

```
int primes[ ] = {2, 3, 5, 7, 11, 13, 19};
```

```
String languages[ ] = { "Java", "C", "C++" };
```

# Array literals

---

- You can create an array literal with the following syntax:

*type[ ] { value1, value2, ..., valueN }*

- Examples:

myPrintArray(new int[] {2, 3, 5, 7, 11});

int foo[ ];

foo = new int[]{42, 83};

# Initializing arrays III

---

- To initialize an array of Person:

```
Person people[ ] =  
{ new Person("Alice"),  
  new Person("Bob"),  
  new Person("Carla"),  
  new Person("Don") };
```

- Notice that you do *not* say the size of the array
  - The computer is better at counting than you are!

# Arrays of arrays

---

- The elements of an array can be arrays
- Once again, there is a special syntax
- Declaration: `int table[ ] [ ];`
- Definition: `table = new int[10][15];`
- Combined: `int table[ ] [ ] = new int[10][15];`
- The first index (**10**) is usually called the row index; the second index (**15**) is the column index
- An array like this is called a two-dimensional array

# Example array of arrays

- `int table[ ][ ] = new int[3][2];` or,
- `int table[ ][ ] = { {1, 2}, {3, 6}, {7, 8} };`

|   | 0 | 1 |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 6 |
| 2 | 7 | 8 |

- For example, `table[1][1]` contains 6
- `table[2][1]` contains 8, and
- `table[1][2]` is “array out of bounds”
- To “zero out this table”:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 2; j++)
        table[i][j] = 0;
```

- How could this code be improved?

# Size of 2D arrays

- `int table[ ][ ] = new int[3][2];`
- The length of this array is the number of *rows*:

|   | 0 | 1 |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 3 | 6 |
| 2 | 7 | 8 |

`table.length` is 3

- Each row contains an array
- To get the number of *columns*, pick a row and ask for its length:

`table[0].length` is 2

- But remember, rows may not all be the same length

# Conditional Statements

---

- A *conditional statement* lets us choose which statement will be executed next by using a conditional test
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- Java's conditional statements are
  - the *if statement*
  - the *if-else statement*
  - the *switch statement*

# The if Statement

- The *if statement* has the following syntax:

**if** is a Java  
reserved word

```
if (condition)  
    statement;
```

The *condition* must be a boolean expression.  
It must evaluate to either true or false.

If the *condition* is true, the *statement* is executed.  
If it is false, the *statement* is skipped.

# Example

---

```
public class IfElseDemo{  
    public static void main(String[] args) {  
        int testscore = 76;  
        char grade;  
        if (testscore >= 90) {  
            grade = 'A'; }  
        else if (testscore >= 80) {  
            grade = 'B'; }  
        else if (testscore >= 70) {  
            grade = 'C'; }  
        else if (testscore >= 60) {  
            grade = 'D'; }  
        else { grade = 'F'; }  
        System.out.println("Grade = " + grade);  
    }  
}
```

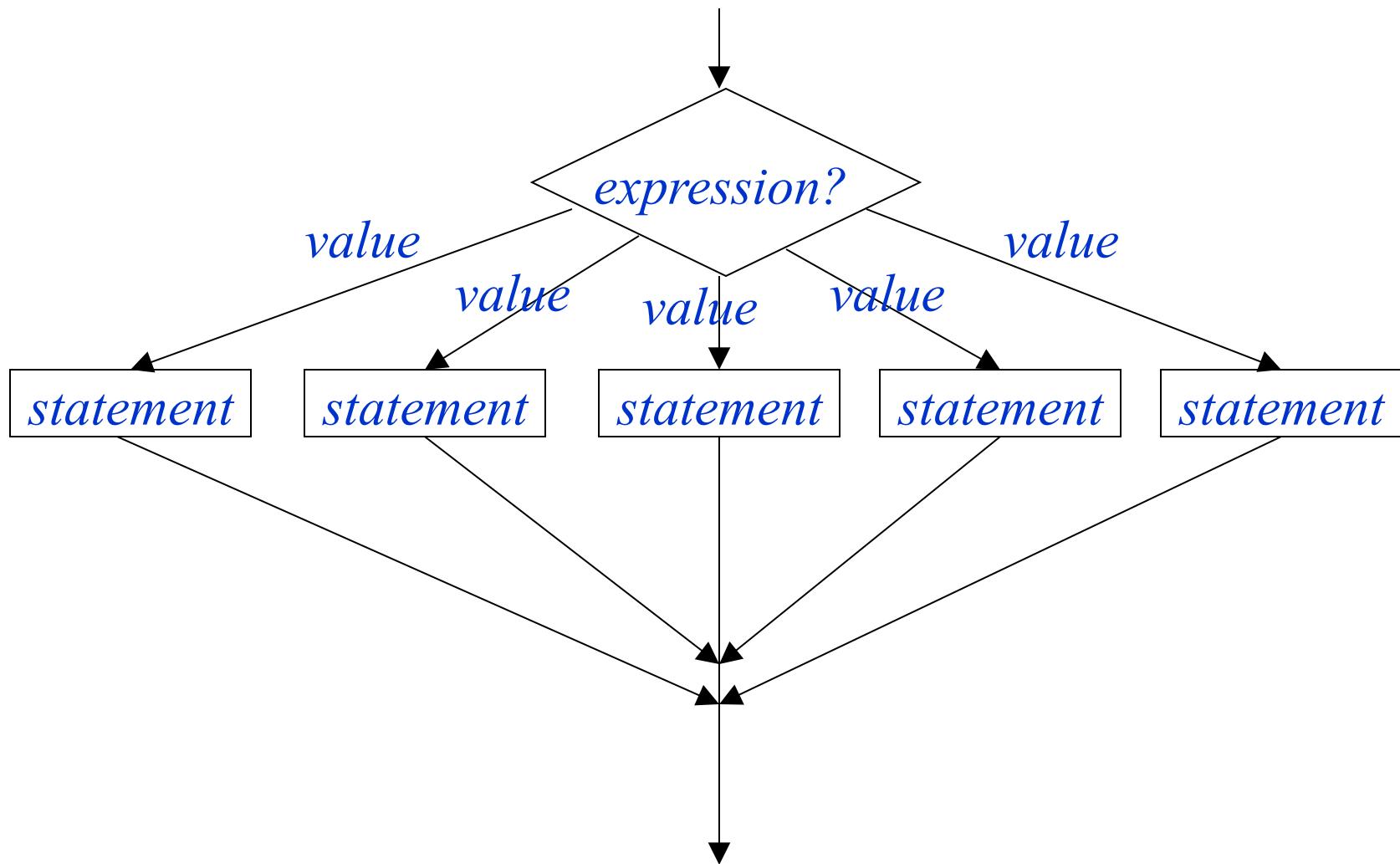
# switch statement

---

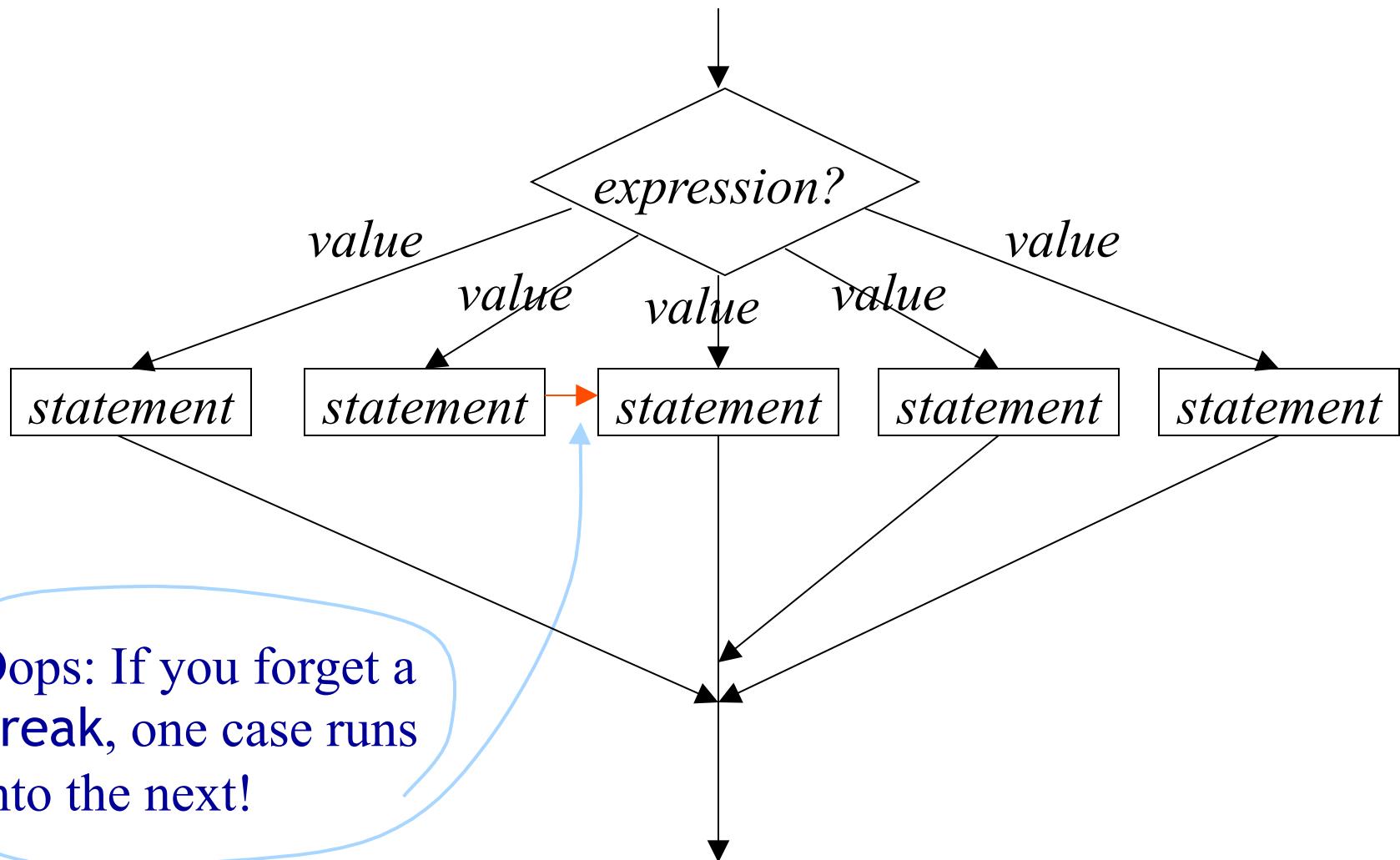
The syntax is:

```
switch (expression) {  
    case value1 :  
        statements ;  
        break ;    case value2 :  
        statements ;  
        break ;  
        ...(more cases)...  
    default :  
        statements ;  
        break ;  
}
```

# Flowchart for switch statement



# Flowchart for switch statement



# Example **switch** statement

---

```
switch (cardValue) {  
    case 1:  
        System.out.print("Ace");  
        break;  
    case 11:  
        System.out.print("Jack");  
        break;  
    case 12:  
        System.out.print("Queen");  
        break;  
    case 13:  
        System.out.print("King");  
        break;  
    default:  
        System.out.print(cardValue);  
        break;
```

# The *while* loop

General forms:

```
while ( booleanExpression ) statement;
```

```
int a[] = { 1, 2, 3, 4, 5};  
int sum = 0;  
  
int i = 0;  
while ( i < a.length ){  
    sum = sum + a[i];  
    i = i + 1;  
}  
  
System.out.println("sum = " + sum);
```

## The *do... while* loop

General forms:

```
do statement;  
while ( booleanExpression );
```

```
int a[] = { 1, 2, 3, 4, 5};  
int sum = 0;  
  
int i = 0;  
do {  
    sum = sum + a[i];  
    i = i + 1;  
} while ( i < a.length );  
  
System.out.println("sum = " + sum);
```

# The *for* loop

```
for ( initializationExp; booleanExp; incrementingExp)  
statement;
```

```
int a[] = { 1, 2, 3, 4, 5};  
int sum = 0;  
  
for (int i=0; i<a.length; i = i + 1){  
    sum = sum + a[i];  
}  
  
System.out.println("sum = " + sum);
```

# Example: Multiplication table

---

```
public static void main(String args[]) {  
    for (int i = 1; i < 11; i++) {  
        for (int j = 1; j < 11; j++) {  
            int product = i * j;  
            if (product < 10)  
                System.out.print(" " + product);  
            else System.out.print(" " + product);  
        }  
        System.out.println();  
    }  
}
```

# Results

---

|    |    |    |    |    |    |    |    |    |     |
|----|----|----|----|----|----|----|----|----|-----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10  |
| 2  | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 | 20  |
| 3  | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 | 30  |
| 4  | 8  | 12 | 16 | 20 | 24 | 28 | 32 | 36 | 40  |
| 5  | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50  |
| 6  | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 | 60  |
| 7  | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 | 70  |
| 8  | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 | 80  |
| 9  | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 | 90  |
| 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |

# The break statement

---

- Inside any loop, the break statement will immediately get you out of the loop
- It doesn't make any sense to break out of a loop unconditionally —you should do it only as the result of an if test
- Even then, it's seldom a good idea
- Example:

```
for (int i = 1; i <= 12; i++) {  
    if (badEgg(i)) break;  
}
```

# Leaving a loop

- Break leaves the loop
- Continue ends the current pass through the loop

```
for(int i=0; i<10; i = i + 1) {  
    for(int k=0; k<20; k = k + 1) {  
        // ...  
        if (...) break;  
        // ...  
        if (...) continue;  
        // ...  
    } // end inner loop  
    System.out.println("After inner loop");  
} // end outer loop
```

● Ends inner loop

● Ends this pass  
of inner loop

# When do you use each loop?

---

- Use the for loop if you know ahead of time how many times you want to go through the loop
  - Example: Print a 12-month calendar
- Use the while loop in almost all other cases
  - Example: Repeat Scott's problem until you get to 1
- Use the do-while loop if you must go through the loop at least once before it makes sense to do the test
  - Example: Ask for the password until user gets it right

# Multiway decisions

---

- The if-else statement chooses one of two statements, based on the value of a **boolean** expression
- The switch statement chooses one of several statements, based on the value on an integer (**int**, **byte**, **short**) or a **char** expression

# Comments

- Three types of comments:

```
// Single line comment
```

```
/* Multiple line comment.  
Comments will continue  
until the following is reached  
*/
```

```
/** Javadoc comment -- multiple line comment that  
can be extracted by the javadoc tool to provide HTML  
documentation.  
*/
```

# Classes

---

A Class describes:

- Fields that hold the data for each object
- Constructors that tell how to create a new object of this class
- Methods that describe the actions the object can perform
- In addition, a class can have data and methods of its own (not part of the objects)
  - For example, it can keep a count of the number of objects it has created

# Defining a class

---

- Here is the simplest syntax for defining a class:

```
class NameOfClass {  
    // the fields (variables) of the object  
    // the constructors for the object  
    // the methods of the object  
}
```

- You can put **public** before the word **class**
- Things in a class can be in any order (I recommend the above order)

# Defining fields

---

- An object's data is stored in fields (also called instance variables)
- The fields describe the state of the object
- Fields are defined with ordinary variable declarations:

String name;

Double health;

int age = 0;

# What is a Constructor?

---

- Constructor is a special method that gets invoked “automatically” at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

# Defining constructors

---

- A constructor is code to create an object
- The syntax for a constructor is:

*ClassName(parameters) {*

*...code...*

*}*

- The *parameters* are a comma-separated list of variable declarations

# Parameters

---

- We usually need to give information to constructors and to methods
- A parameter is a variable used to hold the incoming information
- A parameter must have a name and a type
- You supply values for the parameters when you use the constructor or method
- The parameter name is only meaningful *within* the constructor or method in which it occurs

# Defining a Constructor

- Like any other method

```
public class ClassName {  
  
    // Data Fields...  
  
    // Constructor  
    public ClassName()  
    {  
        // Method Body Statements initialising Data Fields  
    }  
  
    //Methods to manipulate data fields  
}
```

- Invoking:
  - There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

# Defining a Constructor: Example

```
public class Counter {  
    int CounterIndex;  
  
    // Constructor  
    public Counter()  
    {  
        CounterIndex = 0;  
    }  
    //Methods to update or access counter  
    public void increase()  
    {  
        CounterIndex = CounterIndex + 1;  
    }  
    public void decrease()  
    {  
        CounterIndex = CounterIndex - 1;  
    }  
    int getCounterIndex()  
    {  
        return CounterIndex;  
    }  
}
```

# Trace counter value at each statement and What is the output ?

---

```
class MyClass {  
    public static void main(String args[])  
    {  
        Counter counter1 = new Counter();  
        counter1.increase();  
        int a = counter1.getCounterIndex();  
        counter1.increase();  
        int b = counter1.getCounterIndex();  
        if ( a > b )  
            counter1.increase();  
        else  
            counter1.decrease();  
  
        System.out.println(counter1.getCounterIndex());  
    }  
}
```

# A Counter with User Supplied Initial Value ?

```
public class Counter {  
    int CounterIndex;  
  
    // Constructor 1  
    public Counter()  
    {  
        CounterIndex = 0;  
    }  
public Counter(int InitValue )  
{  
    CounterIndex = InitValue;  
}  
}  
  
// A New User Class: Utilising both constructors  
Counter counter1 = new Counter();  
Counter counter2 = new Counter (10);
```

# Adding a Multiple-Parameters Constructor to Circle Class

---

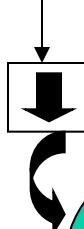
```
public class Circle {  
    public double x,y,r;  
    // Constructor  
    public Circle(double centreX, double centreY,  
                 double radius)  
{  
    x = centreX;  
    y = centreY;  
    r = radius;  
}  
//Methods to return circumference and area  
public double circumference()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

# Constructors initialise Objects

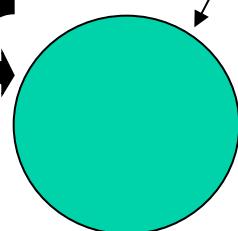
- Recall the following OLD Code Segment:

```
Circle aCircle = new Circle();  
aCircle.x = 10.0; // initialize center and radius  
aCircle.y = 20.0  
aCircle.r = 5.0;
```

aCircle = new Circle();



At creation time the center and radius are not defined.



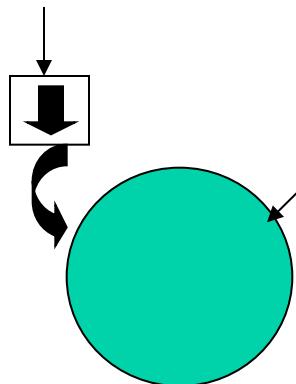
These values are explicitly set later.

# Constructors initialise Objects

- With defined constructor

```
Circle aCircle = new Circle(10.0, 20.0, 5.0);
```

```
aCircle = new Circle(10.0, 20.0, 5.0) ;
```



aCircle is created with center (10, 20)  
and radius 5

# Multiple Constructors

---

- Sometimes want to initialize in a number of different ways, depending on circumstance.
- This can be supported by having multiple constructors having different input arguments.

# Multiple Constructors

```
public class Circle {  
    public double x,y,r; //instance variables  
    // Constructors  
    public Circle(double centreX, double centreY, double radius) {  
        x = centreX; y = centreY; r = radius;  
    }  
    public Circle(double radius) { x=0; y=0; r = radius; }  
    public Circle() { x=0; y=0; r=1.0; }  
  
    //Methods to return circumference and area  
    public double circumference() { return 2*3.14*r; }  
    public double area() { return 3.14 * r * r; }  
}
```

# A problem with names

---

- It would be nice if we could say:

```
- public class Person {  
    String name;  
    boolean male;  
  
    Person (String name, boolean male) {  
  
        name = name ;  
  
        male = male ;  
    }  
}
```

# A problem with names

---

- And have it mean:

```
- public class Person {  
    String name;  
    boolean male;
```

```
Person (String name, boolean male) {
```

```
    name = name ;
```

```
    male = male ;
```

```
}
```

# A problem with names

---

- But this is what it would really mean:

```
- public class Person {
```

```
    String name;
```

```
    boolean male;
```

```
    Person (String name, boolean male) {
```

```
        name = name ;
```

```
        male = male ;
```

```
    }
```

```
}
```

# this to the rescue

---

- A parameter may have the same name as an instance variable
  - The name always refers to the parameter
- The keyword **this** refers to the *current object*
- Putting **this** in front of a name means that the name is a field of *this* object (it isn't a parameter)

# A problem with names—solved!

---

- Here is how we do what we want:

```
- public class Person {  
    String name;  
    boolean male;  
  
    Person (String name, boolean male) {  
  
        this.name = name ;  
  
        this.male = male ;  
    }  
}
```

# A typical use of `this`

---

- If you write a constructor with parameters...
- ...and the parameters are used to set fields that have the same meaning...
- ...then use the same names!

```
Person (String name, boolean male) {
```

```
    this.name = name ;
```

```
    this.male = male ;
```

```
}
```

- In fact, this is the recommended way to do it

# Constructor chaining

```
new Exam() •————→ public Exam() {  
    •————→ this("Java");  
}  
•————→ public Exam(String aName)  
{  
    name = aName;  
}
```

# Defining a method

---

- A method has the syntax:

```
return-type method-name (parameters) {  
    method-variables  
    code  
}
```

- Example:

```
boolean isAdult( ) {  
    int magicAge = 21;  
    return age >= magicAge;  
}
```

# Returning a result from a method

- If a method is to return a result, it must specify the type of the result  
`boolean isAdult ( ...`
- You *must* use a return statement to exit the method with a result of the correct type:  
`return age >= magicAge;`
- This is for *methods only* (constructors automatically return a result of the correct type)

# Returning *no* result from a method

---

- The keyword void is used to indicate that a method doesn't return a value

```
void printAge( ) {  
    System.out.println(name + " is " + age +  
        " years old.");  
  
    return;  
}
```

- The keyword return is not required in a void method

# Sending messages

---

- We send a message to an object by:
  - Naming the object
  - Naming the method we want to use
  - Providing any needed actual parameters
- Example:

```
if (john.isAdult ()) { john.printAge(); }
```

- `john.isAdult()` returns a value (subsequently used by the if statement)
- `john.printAge()` does not return a value

# Putting it all together

```
class Person {  
    // fields  
    String name;  
    int age;  
  
    // constructor  
    Person(String name) {  
        this.name = name;  
        age = 0;  
    }  
  
    // methods  
    String getName() {  
        return name;  
    }  
  
    void birthday() {  
        age = age + 1;  
        System.out.println(  
            "Happy birthday!");  
    }  
}
```

# Using our new class

---

```
Person john;
```

```
john = new Person("John Smith");
```

```
System.out.print (john.getName());
```

```
System.out.println(" is having a birthday!");
```

```
john.birthday();
```

- Of course, this code must *also* be inside a class!

# Class variables and methods

---

- A class describes the variables and methods belonging to objects of that class
  - These are called instance variables and instance methods
- A class may also have *its own* variables and methods
  - These are called class variables and class methods
- Class variables and class methods are denoted by the keyword **static**

# Why have class variables and methods?

---

- Sometimes you want to keep information about the class itself
  - Example: Class **Person** might have a class variable **Population** that counts people
  - This would *not* be appropriate data to keep in each **Person!**
- Sometimes you want to do something relevant to the class as a whole
  - For example, find the average age of a population
- Sometimes you don't have *any* objects
  - For example, you want to start up a program

# Final Variables

- A variable that is declared using the *final* modifier denotes its inability to have its value changed once initialized.

```
public static final int MAX_VALUE=100;  
public static final int MIN_VALUE=10;
```

# Example use of a class variable

```
class Person {  
    // fields  
    String name;  
    int age;  
    static int population;  
  
    // constructor  
    Person(String name) {  
        this.name = name;  
        age = 0;  
        population =  
            population + 1;  
    }  
  
    // methods  
    String getName() {  
        return name;  
    }  
  
    void birthday() {  
        age = age + 1;  
        System.out.println(  
            "Happy birthday!");  
    }  
}
```

# Main Method

```
c:> java Display Hello World
```

JVM

```
public class Display {  
    ...  
    public static void main(String[ ] args) {  
        System.out.println(args[0] + args[1]);  
    }  
    ...  
}
```

# Polymorphism

---

- Allows a single *method or operator* associated with different meaning depending on the type of data passed to it. It can be realized through:
  - Method Overloading
  - Operator Overloading (Supported in C++, but not in Java)
- Defining the same *method* with different argument types (method overloading) - polymorphism.
- The method body can have different logic depending on the date type of arguments.

# Scenario

---

- A Program needs to find a maximum of two numbers or Strings. Write a separate function for each operation.

– In C:

- int max\_int(int a, int b)
- int max\_string(char \*s1, char \*s2)
- max\_int (10, 5) or max\_string (“melbourne”, “sydney”)

– In Java:

- int max(int a, int b)
- int max(String s1, String s2)
- max(10, 5) or max(“melbourne”, “sydney”)

– Which is better ? Readability and intuitive wise ?

# Method Overloading

---

- Constructors all have the same name.
- Methods are distinguished by their signature:
  - name
  - number of arguments
  - type of arguments
  - position of arguments
- That means, a class can also have multiple usual methods with the same name.
- Constructors are frequently overloaded, for example, Student:

Student()

Student(String name)

Student(String name, int age)

Student(int studentNumber)

# A Program with Method Overloading

---

```
// Compare.java: a class comparing different items
class Compare {
    static int max(int a, int b)
    {
        if( a > b)
            return a;
        else
            return b;
    }
    static String max(String a, String b)
    {
        if( a.compareTo (b) > 0)
            return a;
        else
            return b;
    }

    public static void main(String args[])
    {
        String s1 = "Melbourne";
        String s2 = "Sydney";
        String s3  = "Adelaide";

        int a = 10;
        int b = 20;

        System.out.println(max(a, b)); // which number is big
        System.out.println(max(s1, s2)); // which city is big
        System.out.println(max(s1, s3)); // which city is big
    }
}
```

# Static Members

---

- Java supports definition of global methods and variables that can be accessed without creating objects of a class. Such members are called Static members.
- Define a variable by marking with the **static** methods.
- This feature is useful when we want to create a variable common to all instances of a class.
- One of the most common example is to have a variable that could keep a count of how many objects of a class have been created.
- Note: Java creates only one copy for a static variable which can be used even if the class is never instantiated.

# Static Variables

- Define using *static*:

```
public class Circle {  
    // class variable, one for the Circle class, how many circles  
    public static int numCircles;  
  
    //instance variables,one for each instance of a Circle  
    public double x,y,r;  
    // Constructors...  
}
```

- Access with the class name (ClassName.StatVarName):

```
nCircles = Circle.numCircles;
```

# Static Variables - Example

- Using *static variables*:

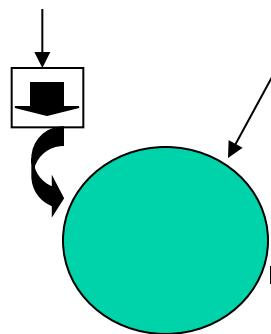
```
public class Circle {  
    // class variable, one for the Circle class, how many circles  
    private static int numCircles = 0;  
    private double x,y,r;  
  
    // Constructors...  
    Circle (double x, double y, double r){  
        this.x = x;  
        this.y = y;  
        this.r = r;  
        numCircles++;  
    }  
}
```

# Class Variables - Example

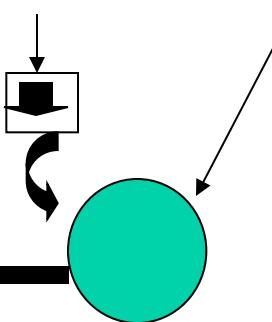
- Using *static variables*:

```
public class CountCircles {  
  
    public static void main(String args[]){  
        Circle circleA = new Circle( 10, 12, 20); // numCircles = 1  
        Circle circleB = new Circle( 5, 3, 10); // numCircles = 2  
    }  
}
```

circleA = new Circle(10, 12, 20)



circleB = new Circle(5, 3, 10)



# Instance Vs Static Variables

---

- **Instance** variables : One copy per **object**. Every object has its own instance variable.
  - E.g. x, y, r (centre and radius in the circle)
- **Static** variables : One copy per **class**.
  - E.g. numCircles (total number of circle objects created)

# Static methods restrictions

---

- They can only call other static methods.
- They can only access static data.
- They cannot refer to “this” or “super” (more later) in anyway.

# Example

```
// Comparator.java: A class with static data items comparision methods
class Comparator {
    public static int max(int a, int b)
    {
        if( a > b)
            return a;
        else
            return b;
    }

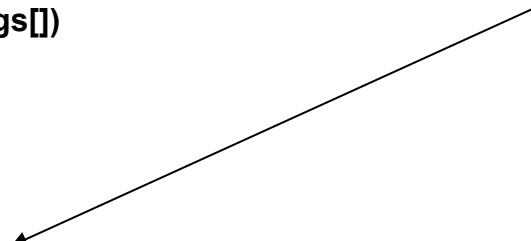
    public static String max(String a, String b)
    {
        if( a.compareTo(b) > 0)
            return a;
        else
            return b;
    }
}

class MyClass {
    public static void main(String args[])
    {
        String s1 = "Melbourne";
        String s2 = "Sydney";
        String s3 = "Adelaide";

        int a = 10;
        int b = 20;

        System.out.println(Comparator.max(a, b)); // which number is big
        System.out.println(Comparator.max(s1, s2)); // which city is big
        System.out.println(Comparator.max(s1, s3)); // which city is big
    }
}
```

Directly accessed using Class Name (NO Objects)



# Unit Summary

---

- Object oriented concepts
- Java Virtual Machine
- Java Development Cycle.
- components of class in Java
- encapsulation and class instantiation
- Java keywords, operators, and primitive data types
- implicit and explicit casting.
- conditional and iteration statement
- Arrays.

*Welcome to :*  
**Inheritance**

# Unit Objective

---

- After completing this unit you should be able to:
  - Apply the concept of inheritance
  - Define sub class and super class
  - Explain overriding methods.
  - Describe the principle of dynamic binding.
  - Final classes
  - Abstract classes

# Inheritance

---

- Another fundamental object-oriented technique is called inheritance, which enhances software design and promotes reuse.
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*

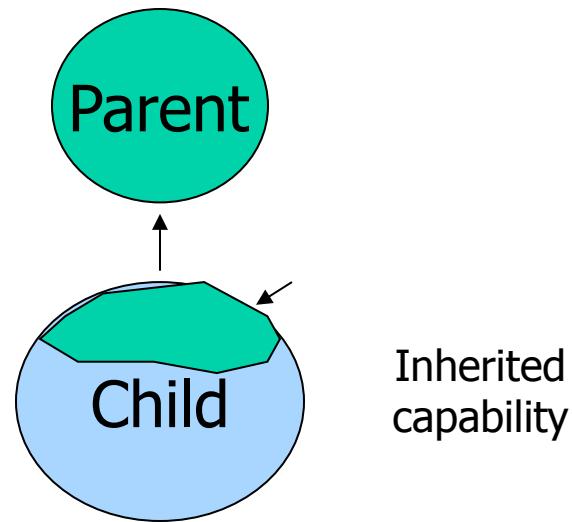
# Inheritance

---

- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined for the parent class

# Inheritance

- Inheritance relationships are often shown graphically with the arrow pointing to the parent class



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

# Deriving Subclasses

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

```
class Car extends Vehicle
{
    Sub Class
    // class contents
}
Super class
```



# Controlling Inheritance

---

- Visibility modifiers determine which class members get inherited and which do not
- Variables and methods declared with **public** visibility are inherited, and those with **private** visibility are not
- But **public** variables violate our goal of encapsulation
- There is a third visibility modifier that helps in inheritance situations: **protected**

# The **protected** Modifier

---

- The **protected** visibility modifier allows a member of a base class to be inherited into the child
- But **protected** visibility provides more encapsulation than **public** does
- However, **protected** visibility is not as tightly encapsulated as **private** visibility

# The `super` Reference

---

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's part of the object
- The `super` reference can be used to refer to the parent class, and is often used to invoke the parent's constructor

# Single vs. Multiple Inheritance

---

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Interface can be used

# Overriding Methods

---

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

# Overriding Methods

---

- Note that a parent method can be explicitly invoked using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data (called *shadowing variables*), there is generally no need for it

# Shadowing – Inheritance example

- Shadowing is resolved at compile time (like overloading), and compiler looks at the static class, not the dynamic class.

```
class Parent {  
    public int x = 12;  
}  
  
class Child extends Parent {  
    public int x = 42;  
}  
  
Parent p = new Parent(), Child c = new Child();  
  
System.out.print(p.x); // prints “12”  
  
System.out.print(c.x); // prints “42” // parent ‘x’ shadowed  
  
p = c; // assign child to superclass variable  
  
System.out.print(p.x); // prints “12”: compile time
```

## Final Members: A way for Preventing Overriding of Members in Subclasses

---

- All methods and variables can be overridden by default in subclasses.
- This can be prevented by declaring them as final using the keyword “final” as a modifier. For example:
  - final int marks = 100;
  - final void display();
- This ensures that functionality defined in this method cannot be altered any. Similarly, the value of a final variable cannot be altered.

# Final Classes: A way for Preventing Classes being extended

---

- We can prevent an inheritance of classes by other classes by declaring them as final classes.
- This is achieved in Java by using the keyword final as follows:

```
final class Marks
```

```
{ // members  
}
```

```
final class Student extends Person
```

```
{ // members  
}
```

- Any attempt to inherit these classes will cause an error.

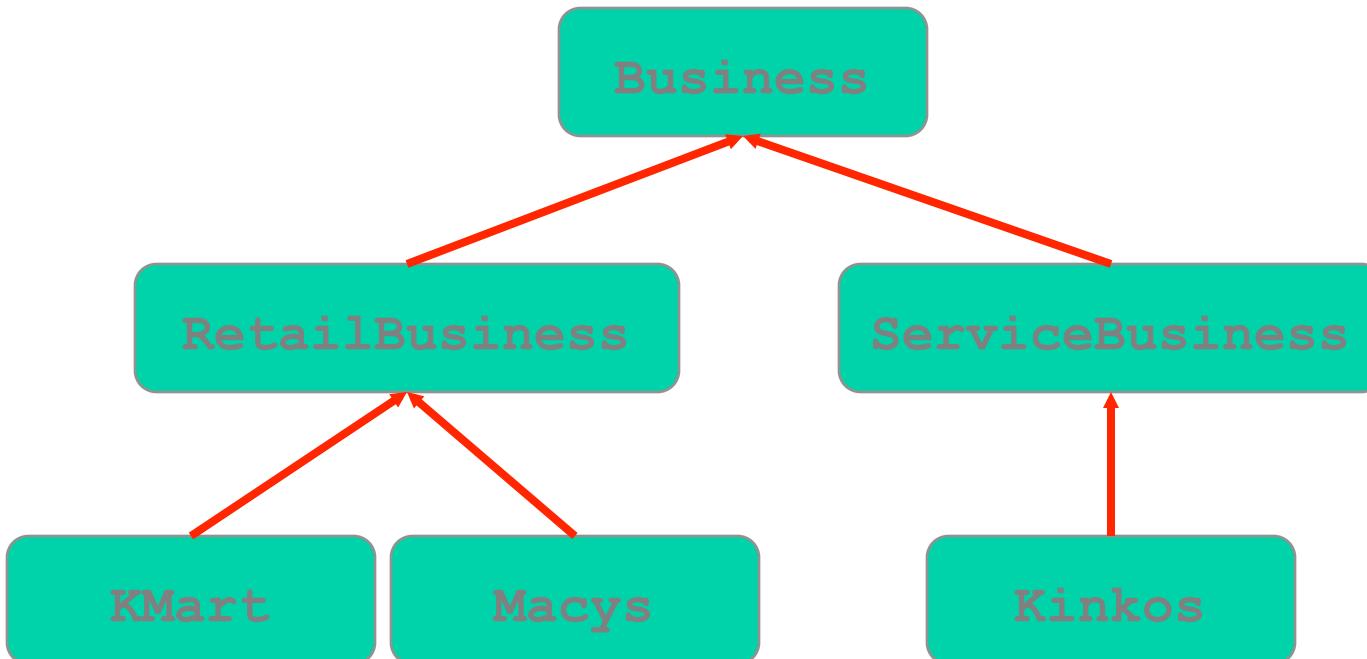
# Overloading vs. Overriding

---

- Overloading deals with multiple methods in the same class with the same name but different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar operation in different ways for different data
- Overriding lets you define a similar operation in different ways for different object types

# Class Hierarchies

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



# The Object Class

---

- A class called `Object` is defined in the `java.lang` package of the Java standard class library
- All classes are derived from the `Object` class
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class
- The `Object` class is therefore the ultimate root of all class hierarchies

# The Object Class

---

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we have defined `toString`, we have actually been overriding it
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class and a hash value

# The Object Class

---

- That's why the `println` method can call `toString` for any object that is passed to it – all objects are guaranteed to have a `toString` method via inheritance
- The `equals` method of the `Object` class determines if two references are aliases
- You may choose to override `equals` to define equality in some other way

# Abstract Classes

---

- An abstract class is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated

```
abstract class ClassName
{
    ...
    ...
    abstract Type MethodName1();
    ...
    ...
    Type Method2()
    {
        // method body
    }
}
```

# Abstract Classes

---

- The child of an abstract class must override the abstract methods of the parent, or it too will be considered abstract
- An abstract method cannot be defined as final or static
- The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

# Example

---

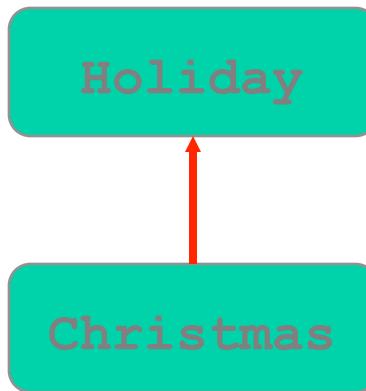
```
public abstract class Exam {  
    \ different data  
    public abstract void testPaper();  
    other methods may be defined }
```

```
class PracticeExam extends Exam{  
    void testPaper() { //implementation }}
```

```
class Quiz extends Exam{  
    void testPaper() { // implementation }}
```

# References and Inheritance

- An object reference can refer to an object of its class, or to an object of any class related to it by inheritance



```
Holiday day;  
day = new Christmas();
```

# References and Inheritance

---

- Assigning a predecessor object to an ancestor reference is considered to be a widening conversion, and can be performed by simple assignment
- Assigning an ancestor object to a predecessor reference can also be done, but it is considered to be a narrowing conversion and must be done with a cast
- The widening conversion is the most useful

# Polymorphism via Inheritance

---

- A polymorphic reference is one which can refer to different types of objects at different times
- Inheritance can also be used as a basis of polymorphism
- An object reference can refer to one object at one time, then it can be changed to refer to another object (related by inheritance) at another time

# Polymorphism via Inheritance

---

- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrode it
- Now consider the following invocation:

```
day.celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

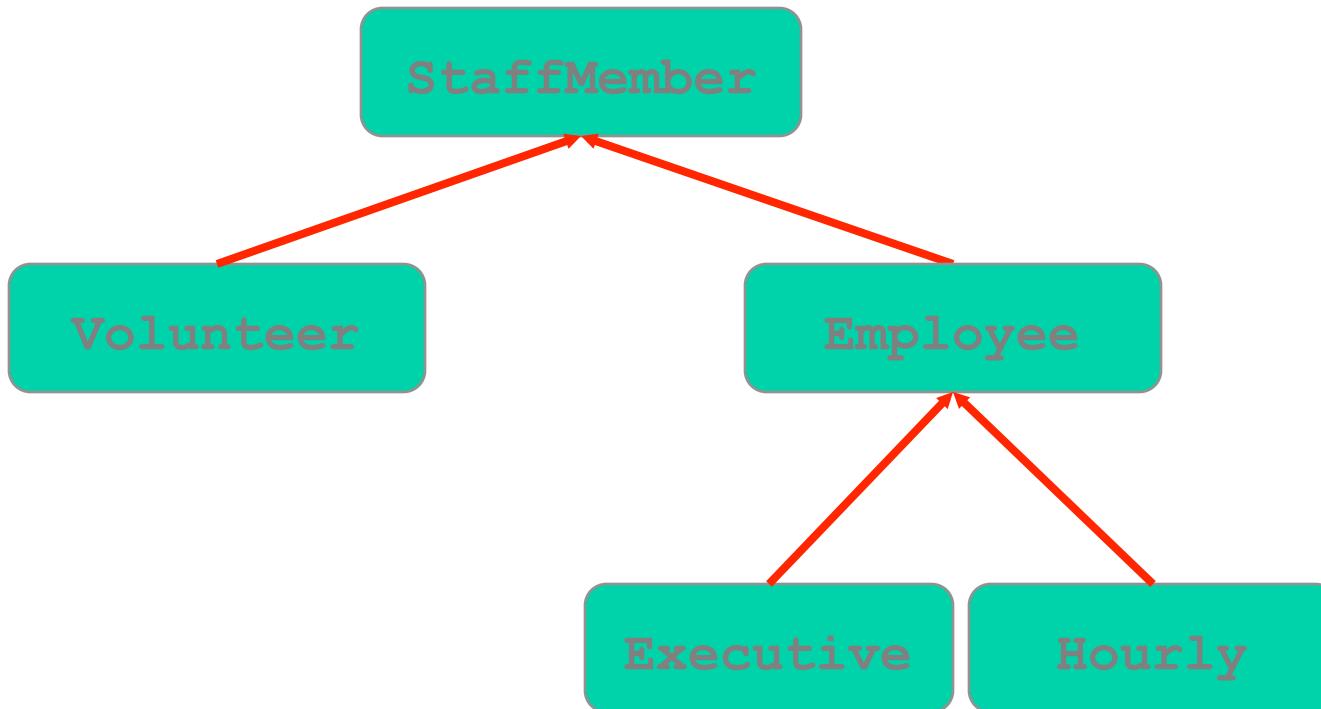
# **Polymorphism via Inheritance**

---

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Polymorphic references are resolved at run-time, not during compilation

# Polymorphism via Inheritance

- Consider the following class hierarchy:



# Indirect Access

---

- An inherited member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is not inherited by a child, it can still be accessed indirectly through parent methods

# Interface Hierarchies

---

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces

# Unit Summary

---

- Concept of inheritance
- Sub class and super class
- Difference between overriding and overloading methods.
- Dynamic binding.
- Final classes
- Abstract classes

*Welcome to*  
**Packages**

# **Unit Objective**

---

- After completing this unit you should be able to:
  - What are packages
  - Creation of packages
  - Using packages
  - Java packages.

# Packages

---

- Way to group a related class and interface into one unit
- To resolve the name conflicts between class names
- You have seen packages before. Remember the `import` keyword?
  - e.g. `import java.awt.Graphics;`
  - This tells the compiler that some of the classes and interfaces in your class come from the `Graphics` package.

# Why use packages?

---

- To easily distinguish related classes and interfaces based on the package it is located
- To determine related classes and interfaces and know where to find classes and interfaces that provide certain functions
  - e.g. `java.awt.Graphics` or `java.lang.Math`

# Why use packages?

---

- The names of your classes won't conflict with class names in other packages, because the package creates a new namespace.
- You can allow classes within the package to have unrestricted access to one another yet still restrict access for classes outside the package.

# Using packages

---

- To use a class or an interface that's in a different package, you have three choices:
  1. Use the fully qualified name of the class or the interface.
  2. Import the class or the interface.
  3. Import the entire package of which the class or the interface is a member.
- You might have to set your class path so that the compiler and the interpreter can find the source and class files for your classes and interfaces.

# Creating a package

---

Add this to the first line of the java file:

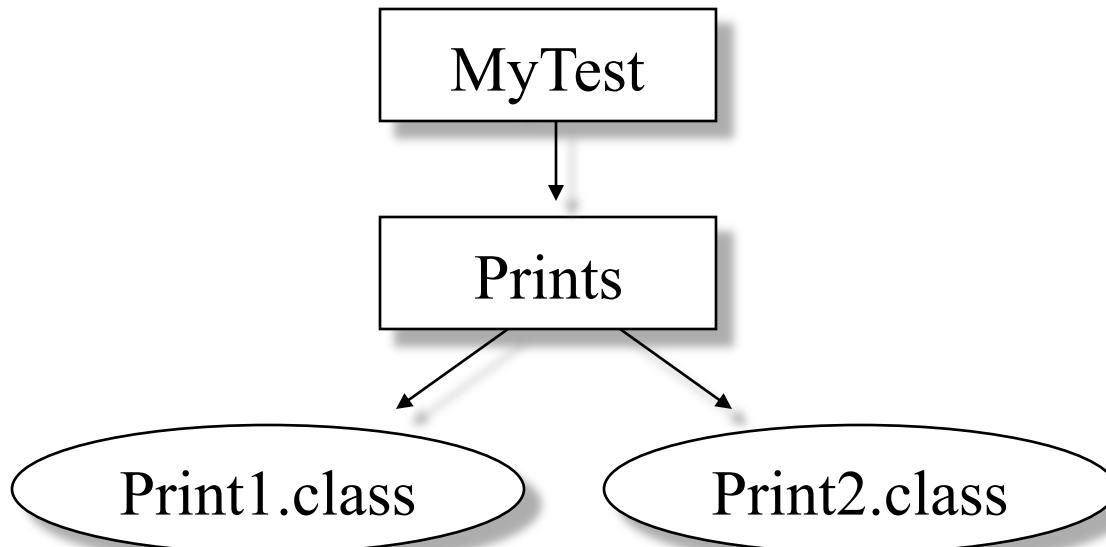
- package <foldername>;
- e.g. package world;
- If a class does not have this on the first line, it belongs to the default package

# Declaration of Package

```
package PackageName ;
```

- Declare at the beginning of source file

[Print1.java], [Print2.java]



# Creating a package

---

Compile the java file by typing this:

- javac -d .<classname>.java
- e.g. javac -d .HelloWorld.java

Run the program by typing the fully-qualified class name:

- java packagename.classname
- e.g. java world.HelloWorld

# Usage of Package

- Use of absolute path name
  - Declare to describe the full package name
  - To resolve conflicts between class names

```
MyTest.Prints.Print1 m1;  
MyTest.Prints.Print2 m2;
```

# Use of import Statement

```
import packageName.className ;
```

- Include all the classes in package
  - import packageName.\*;

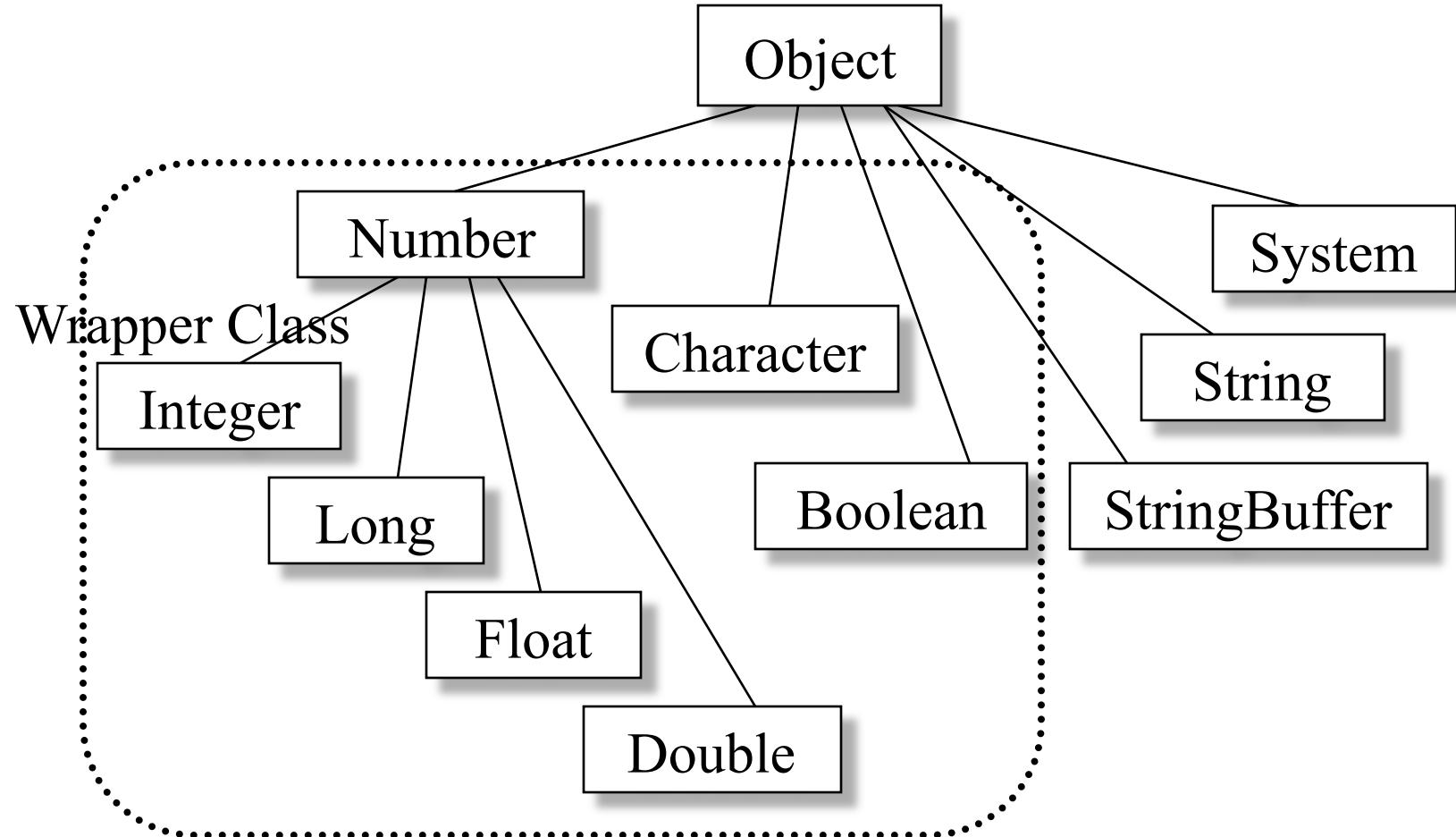
```
import MyTest.Prints.*;
```

# **java.lang Package**

---

- Basic class to enlarge the function of Java
- Provide classes according to primitive type (wrapper class)
  - Integer class, Double class, . . .

# java.lang Class



# Unit Summary

---

- What are packages
- Creation of packages
- Using packages
- Java packages.

*Welcome to*  
**Access modifiers**

# Unit Objective

---

- After completing this unit you should be able to:
  - Access modifiers for classes
  - Data types and methods
  - Using in same or different package

# Access modifiers

---

- Describe the access level or visibility of certain members (class, fields, methods, etc.)
- This allow for better use of encapsulation, especially when you want to protect against access for certain members.

# Access rights for different elements

| Class/have access to              | private | default | protected | public |
|-----------------------------------|---------|---------|-----------|--------|
| own class (Base)                  | yes     | yes     | yes       | yes    |
| subclass, same package (SubA)     | no      | yes     | yes       | Yes    |
| class, same package (AnotherA)    | no      | yes     | yes       | yes    |
| subclass, another package (SubB)  | no      | no      | yes       | yes    |
| class, another package (AnotherB) | no      | no      | no        | yes    |

# Access modifiers

---

- public
  - A member (variable or method) with this access modifier can be accessed by *any object*
- private
  - A member with this access modifier can be accessed from *the same class*
  - But cannot be accessed by *different classes*

# Access modifiers

---

- protected
  - A member with this access modifier can be accessed from:
    - the same class
    - a sub-class
    - the same package
  - But cannot be accessed from *different packages* (is not from a sub-class)

# Access modifiers

---

- default
  - A member that does not have an access modifier can be accessed from
    - the same class
    - the same package
  - But cannot be accessed from a different package

# Attributes (Data) Modifiers

---

- Data with no modifier is visible to sub-class within the same package but not to sub-class outside the package.
- Private data is available only within the class in which it is defined.
- Protected is available to the class and all its sub-classes.
- Public is available to all class.

*Welcome to*  
**Garbage Collection**

# Unit Objective

---

- After completing this unit you should be able to:
  - Life cycle of an object
  - Java garbage collection
  - Writing finalization methods

# Garbage Collection (GC)

- As a program is executed, data is dynamically allocated and assigned to structures and objects
- Since memory is finite, we need to conserve memory whenever possible
- A good practice is to reclaim any previously allocated memory when no longer in use
- Some programming languages leave this up to the programmer (C, C++ etc.), while others alleviate this problem by providing a garbage collection facility
- The JVM runs a garbage collector service in a separate thread of execution when a program is run on the JVM
- The garbage collector periodically checks for unused allocated memory, and reclaims it for re-allocation later on in the program

# Lifetime of Objects

---

- An object is considered alive if:
  - there is a reference to the object on the JVM stack
  - there is a reference to the object in a local variable, on the stack, or in a static field
  - if a field of an alive object contains a reference to the object
  - if the JVM keeps an internal reference to the object (e.g. for supporting native methods)
- If an object is not deemed alive by these rules, it is considered dead, and is ready to be garbage collected

# Pros/Cons of GC

---

Pros:

- Manual memory management is (programmer) time consuming, and error prone. Most programs still contain leaks. This is all doubly true with programs using exception-handling and/or threads.
- Simplifies the interfaces between components (subroutines, libraries, modules, classes) that no longer need expose memory management details

Cons:

- Slower: adds additional overhead on execution of program (Java slower than c++).

# Finalization

---

- OO programming introduces the notion of constructors and destructors for objects:
  - constructors are invoked upon object creation
  - destructors are invoked upon object destruction
- Destructors are used to perform last minute cleaning up on an object:
  - releasing a database connection, closing a network socket, etc.
- In Java, the finalize method is used in conjunction with the garbage collector to perform any housekeeping tasks before an object is garbage collected

# Unit Summary

---

- Life cycle of an object
- Java garbage collection
- Finalization methods

*Welcome to :*

**Interface**

# Unit Objective

---

- After completing this unit you should be able to:
  - Multiple inheritance
  - Interface in java
  - Implementing and extending interfaces

# Interfaces

---

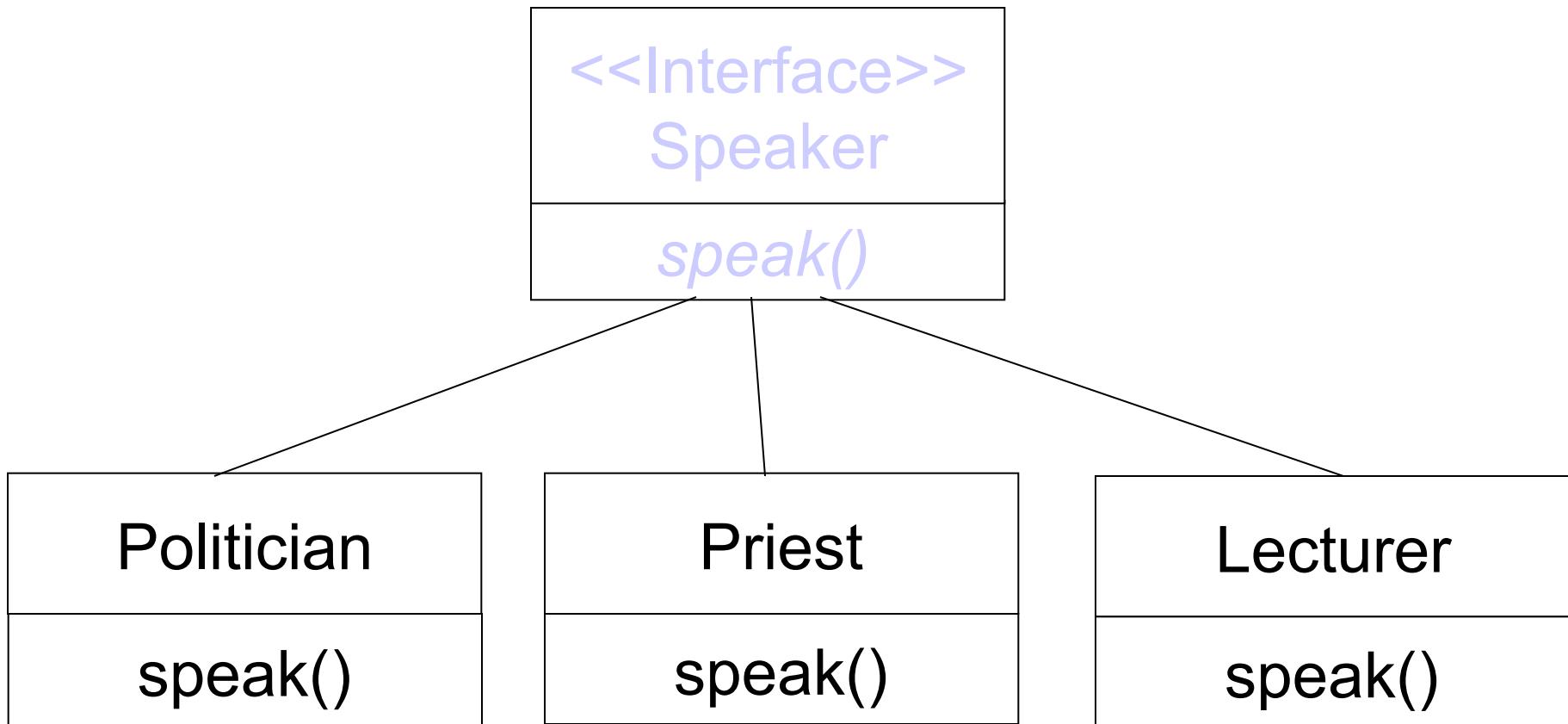
- *Interface* is a conceptual entity similar to a Abstract class.
- Can contain only **constants (final variables)** and **abstract method** (no implementation) - Different from Abstract classes.
- Use when a number of classes share a common interface.
- Each class should implement the interface.

# Interfaces: An informal way of realizing multiple inheritance

---

- An interface is basically a kind of class—it contains methods and variables, but they have to be only abstract classes and final fields/variables.
- Therefore, it is the responsibility of the class that implements an interface to supply the code for methods.
- A class can implement any number of interfaces, but cannot extend more than one class at a time.
- Therefore, interfaces are considered as an informal way of realizing multiple inheritance in Java.

# Interface - Example



# Interfaces Definition

- Syntax (appears like abstract class):

```
interface InterfaceName {  
    // Constant/Final Variable Declaration  
    // Methods Declaration – only method signature  
}
```

- Example:

```
interface Speaker {  
    public void speak( );  
}
```

# Implementing Interfaces

- Interfaces are used like super-classes who properties are inherited by classes. This is achieved by creating a class that implements the given interface as follows:

```
class ClassName implements InterfaceName [, InterfaceName2, ...]
{
    // Body of Class
}
```

# Example

```
class Politician implements Speaker {  
    public void speak(){  
        System.out.println("Talk politics");  
    }  
}
```

```
class Priest implements Speaker {  
    public void speak(){  
        System.out.println("Religious Talks");  
    }  
}
```

```
class Lecturer implements Speaker {  
    public void speak(){  
        System.out.println("Talks Object Oriented Design and Programming!");  
    }  
}
```

# Extending Interfaces

- Like classes, interfaces can also be extended. The new sub-interface will inherit all the members of the superinterface in the manner similar to classes. This is achieved by using the keyword **extends** as follows:

```
interface InterfaceName2 extends InterfaceName1
{
    // Body of InterfaceName2
}
```

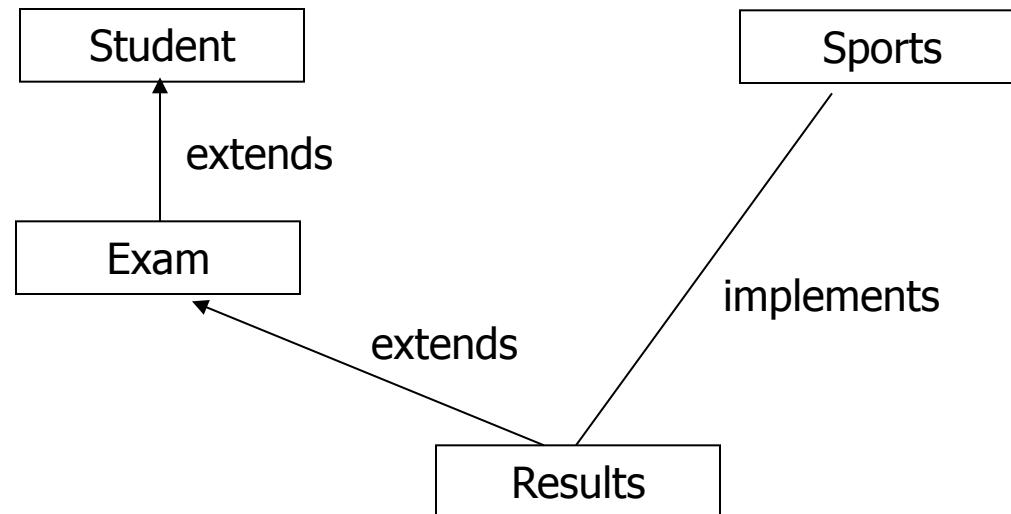
# Inheritance and Interface Implementation

- A general form of interface implementation:
- This shows a class can extend another class while implementing one or more interfaces. It appears like a multiple inheritance (if we consider interfaces as special kind of classes with certain restrictions or special features).

```
class ClassName extends SuperClass implements InterfaceName [,  
InterfaceName2, ...]  
{  
    // Body of Class  
}
```

# Student Assessment Example

- Consider a university where students who participate in the national games or Olympics are given some grace marks. Therefore, the final marks awarded = Exam\_Marks + Sports\_Grace\_Marks. A class diagram representing this scenario is as follow:



# Software Implementation

---

```
class Student
{
    // student no and access methods
}

interface Sport
{
    // sports grace marks (say 5 marks) and abstract methods
}

class Exam extends Student
{
    // example marks (test1 and test 2 marks) and access methods
}

class Results extends Exam implements Sport
{
    // implementation of abstract methods of Sport interface
    // other methods to compute total marks = test1+test2+sports_grace_marks;
    // other display or final results access methods
}
```

# Interfaces and Software Engineering

---

- *Interfaces*, like abstract classes and methods, provide templates of behaviour that other classes are expected to implement.
- Separates out a design hierarchy from implementation hierarchy. This allows software designers to enforce/pass common/standard syntax for programmers implementing different classes.
- Pass method descriptions, not implementation
- Java allows for inheritance from only a single superclass. *Interfaces* allow for *class mixing*.
- Classes *implement* interfaces.

# Unit Summary

---

- Multiple inheritance achieved by interfaces
- Implementing and extending interfaces
- Difference between abstract class and interface.

*Welcome to :*

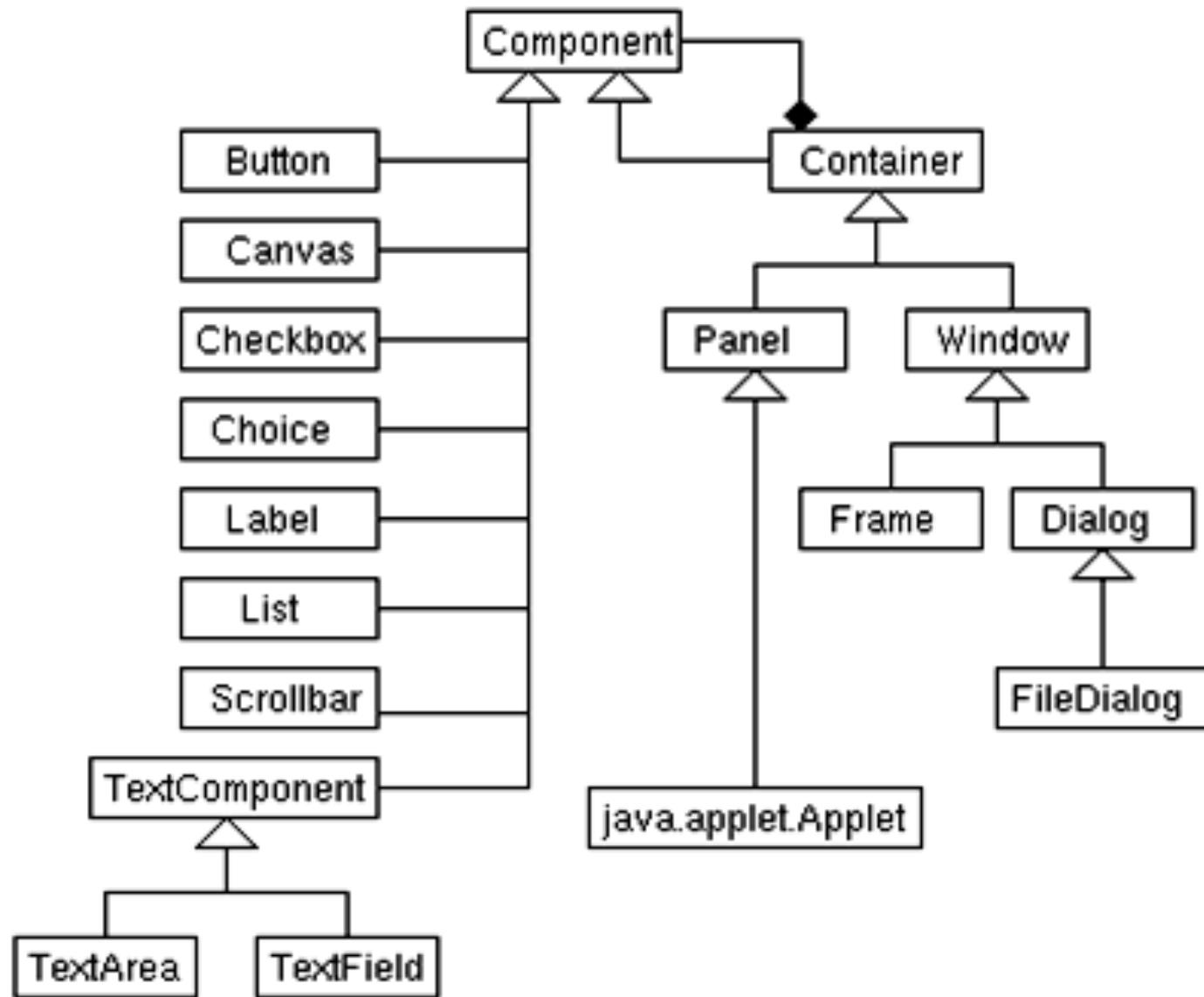
# **Abstract Window Toolkit**

# Unit Objective

---

- After completing this unit you should be able to:
  - Define AWT
  - Identify general organization of AWT
  - Describe component class hierarchy.
  - Handling events
  - Define java's layout Manager.
  - Describe a process of setting components layout manager
  - Efficiently describe layout manager used to achieve proper GUI Layout.
  - Using inner classes.
  - Graphics in Java

# AWT Class Hierarchy



# AWT Components

---

- *Basic components*: Present information to or get information from user.
  - Examples: Button, Label, TextField, TextArea, MenuBar.
- *Containers*: Hold other components.
  - Examples: Frame, Panel.
- *Intermediate container*: For organizing other components

Example: Frame

- *Top-level container*: contains all of the visual components of a GUI
  - Examples: Panel, Window.

# Common AWT Components

---

- A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.  
Examples: buttons, checkboxes, and scrollbars of a typical graphical user interface.
- The Component class is the abstract superclass of the Abstract Window Toolkit components.
- The basic steps involved in using a GUI component are:
  1. The component object must be created with a constructor.
  2. It must be added to a container.

# Frame

---

- A frame is an independent window.
- When frame is constructed it does not have a size and is not visible.
- A frame can be created by

```
class MyFrame extends Frame  
{  
    MyFrame(){  
        super("title");  
    }  
    //  
}
```

# Panel

---

- Panel is invisible component for container.
- Number of components can be placed using panel.

```
Panel panel;
```

```
panel = new Panel();
```

```
button1 = new Button("open");
```

```
button2 = new Button("Exit");
```

```
panel.add(button1);
```

```
panel.add(button2);
```

# Standard Component methods

---

- **comp.setBackground(color) and comp.setForeground(color):**
  - Sets the background and foreground colors for the component. If no colors are set for a component, it inherits the colors of its parent container.
- **comp.setFont(font):**
  - Sets the font for the component.
  - That font is used to display text on the component.
- **comp.getSize():**
  - Returns the size of the component in the form of a Dimension object.  
When a component is first created, its size is zero. The size will be set later, probably by a layout manager.

# Standard Component methods

---

- **comp.getParent():**
  - Returns the parent Container of the component. For a top-level component such as a Window or Applet, the return value will be null.
- **comp.getLocation():**
  - Gets the location of this component in the form of a point specifying the component's top-left corner. The location will be relative to the parent's coordinate space.
- **comp.isEnabled():**
  - Determines whether the component is enabled. An enabled component can respond to user input. Components are enabled initially by default.
- **comp.setVisible(boolean b):**
  - Shows or hides the component depending on the value of parameter b. If true, shows the component; otherwise, hides the component.

# **Button**

---

- A Button component creates a labeled button.
- The constructor of Button class:

**Button()**

**Button(String label)**

- An object of class Button is a push button. The application can cause some action to happen when the button is pushed.
- The gesture of clicking on a button with the mouse is associated with one instance of ActionEvent

# Example: Button

- `Button quitButton = new Button("Quit");`



`setLabel()` and `getLabel()`: Sets and gets the label associated with the button.

# Label

---

- A Label component is a single line of text.
- The text cannot be edited by the user, although it can be changed by your program.
- A Label class has two main constructors:
  1. Label name = new Label("John"); //This Label constructor displays the given text.
  2. Label name = new Label("John", Label.CENTER); //creates a label whose text is centered in the available space.
- getText() and setText(): Sets and gets the text for the label.  
name.setText("Harry");

# Example: Label

---

```
add(new Label("Hi There!"));
```

```
add(new Label("Another Label"));
```



Hi There! Another Label

# Choice

---

- The Choice component presents a pop-up menu of choices. The current choice is displayed as the title of the menu.
- The Choice class has only the default constructor:
  - Choice()

```
Choice ColorChooser = new Choice(); ColorChooser.add("Green");
ColorChooser.add("Red"); ColorChooser.add("Blue");
```



# Choice methods

---

- **getItemCount():**

- Returns an int that gives the number of items in the list.

- **getItem(int index):**

- Returns the string at the specified index in the Choice menu. (Items numbering starts with 0)

- **getSelectedIndex():**

- Returns the index of the currently selected item.

- **getSelectedItem():**

- Returns a string representation of the currently selected item in the Choice menu.

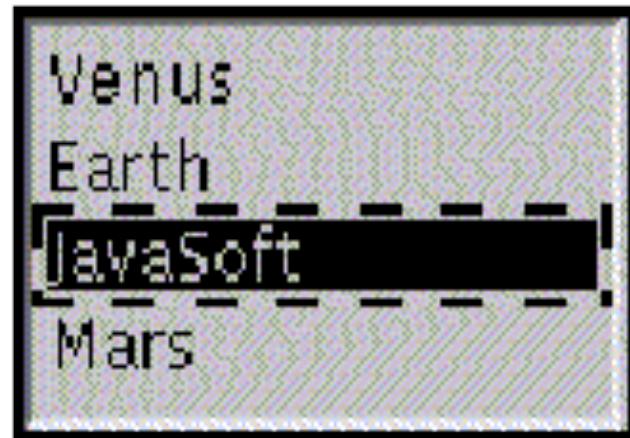
# List

---

- The List component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items.
- The List class has two main constructors:
  1. List () –only one item can be selected at a given time
  2. List (int rows) - Creates a new scrolling list initialized with the specified number of visible lines.
  3. List (int rows, boolean multipleMode) - Creates a new scrolling list initialized to display the specified number of rows.

# Example : List

```
List lst = new List(4, false);
lst.add("Mercury");
lst.add("Venus");
lst.add("Earth");
lst.add("JavaSoft");
lst.add("Mars");
lst.add("Jupiter");
lst.add("Saturn");
lst.add("Uranus");
lst.add("Neptune");
lst.add("Pluto");
cnt.add(lst);
```



# List methods

---

- **add(String):**
  - Adds the specified item to the end of scrolling list.
- **add(String,int):**
  - Adds the specified item to the scrolling list at the position indicated by the index.
- **getItemCount():**
  - Gets the number of items in the list.
- **getItem():**
  - Gets the item associated with the specified index.

# List methods (continued..)

---

- **getItems():**

- Returns a string array containing items of the list.

- **remove(int):**

- Remove the item at the specified position from the scrolling list.

- **remove(String):**

- Removes the first occurrence of an item from the list.

- **removeAll():**

- Removes all items from the list.

- **getSelectedIndex():**

- Returns the index of the selected item, or -1 if no item is selected, or if more than one item is selected.

# List methods (continued..)

---

- **getSelectedIndexes():**

- Returns an array of the selected indexes of this scrolling list. If no items are selected, a zero-length array is returned.

- **getSelectedItem():**

- Returns the selected item on the list, or null if no item is selected.

- **getSelectedItems():**

- Returns the selected item on the list, or null if no item is selected.

# Checkbox and CheckboxGroup Classes

- A Checkbox is a graphical component that can be in either an "on" (true) or "off" (false) state. Clicking on a check box changes its state from "on" to "off," or from "off" to "on."
- A Checkbox class has four main constructors:

## **Checkbox(String label)**

Creates a check box with the specified label.

## **Checkbox(String label, boolean state)**

Creates a check box with the specified label and sets the specified state.

# Checkbox

---

- **Checkbox(String label, boolean state, CheckboxGroup group)**

Creates a check box with the specified label, in the specified check box group, and set to the specified state.

- **Checkbox(String label, CheckboxGroup group, boolean state)**

Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.

# Example: Checkbox

```
add(new Checkbox("one", null, true));
```

```
add(new Checkbox("two"));
```

```
add(new Checkbox("three"));
```



# Checkbox methods

---

- **getLabel() and setLabel():**
  - gets and sets the label for the checkbox
- **getState():**
  - Determines whether the check box is in the "on" or "off" state. The boolean value true indicates the "on" state, and false indicates the "off" state.
- **setState(boolean state):**
  - Sets the state of the check box to the specified state. The boolean value true indicates the "on" state, and false indicates the "off" state.

# Radio Buttons

---

- Radio buttons occur in groups.
- At most one radio button in a group can be checked at any given time.
- A radio button is an object of type Checkbox that is a member of such a group.
- An entire group of radio buttons is represented by an object belonging to the class CheckboxGroup.

# Example: Radio Button

---

- To create a group of radio buttons, you should first create an object of type CheckboxGroup

```
CheckboxGroup colorGroup = new CheckboxGroup();
Checkbox red = new Checkbox("Red", colorGroup, false);
Checkbox blue = new Checkbox("Blue", colorGroup, false);
Checkbox green = new Checkbox("Green", colorGroup,
true);
Checkbox black = new Checkbox("Black", colorGroup, false);
```

# Class TextField

---

A TextField object is a text component that allows for the editing of a single line of text.

There are three main constructors

TextField([int columns](#))

Constructs a new empty text field with the specified number of columns.

TextField([String text](#))

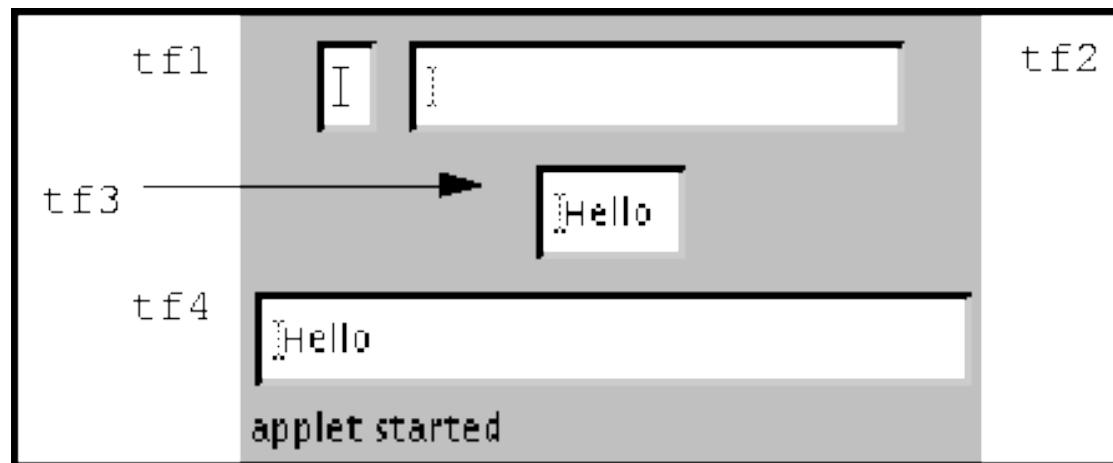
Constructs a new text field initialized with the specified text.

TextField([String text, int columns](#))

[Constructs](#) a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

# Example: TextField

```
TextField tf1, tf2, tf3, tf4; // a blank text field  
tf1 = new TextField(); // blank field of 20 columns  
tf2 = new TextField("", 20); // predefined text displayed  
tf3 = new TextField("Hello!");  
tf4 = new TextField("Hello", 30); // predefined text in 30 columns
```



# **TextComponent methods**

---

- **setText(String text):**
  - Sets the text that is presented by the text component to be the specified text.
- **getText():**
  - Gets the text that is presented by the text component
- **getSelectedText():**
  - Returns the selected text of the text component.
- **select(int start, int end):**
  - Selects the text between the specified start and end positions.  
Characters in the range start  $\leq$  pos  $<$  end are selected; characters are numbered starting from zero.
-

# (continued...)

---

- **selectAll():**
  - Selects all the text in the text component.
- **getSelectionStart():**
  - Gets the start position of the selected text in the text component.
- **getSelectionEnd():**
  - Gets the end position of the selected text in the text component.
- **isEditable():**
  - Returns true if the text component is editable; false otherwise.
  -

# Class TextArea

---

A TextArea object is a multi-line region that displays text. It can be set to allow editing or to be read-only.

A TextArea has four main constructors:

1. `TextArea(int rows, int columns)`

Constructs a new empty text area with the specified number of rows and columns.

2. `TextArea(String text)`

Constructs a new text area with the specified text.

3. `TextArea(String text, int rows, int columns)`

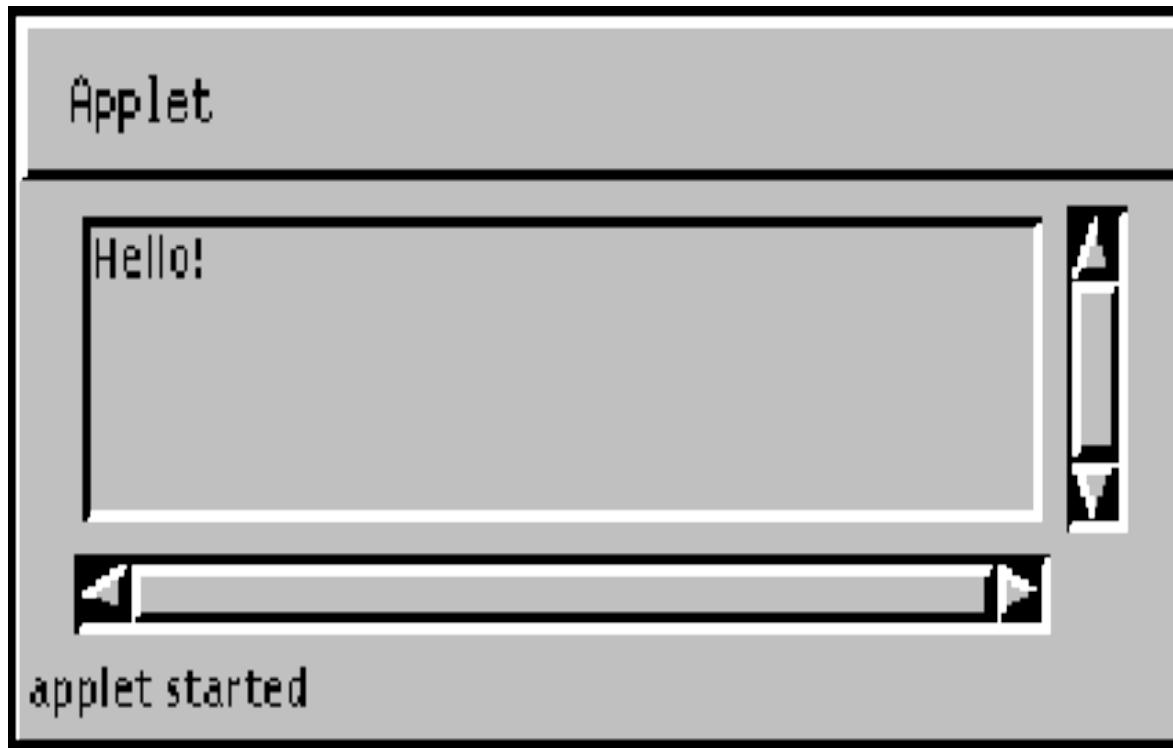
Constructs a new text area with the specified text, and with the specified number of rows and columns.

4. `TextArea(String text, int rows, int columns, int scrollbars)`

Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

# Example: TextArea

```
new TextArea("Hello", 5, 40);
```



# TextArea methods

---

The TextArea class adds a few useful procedures to those inherited from TextComponent:

- **append(String text):**

- Adds the specified text at the end of the current contents; line breaks can be inserted by using the special character \n

- **insert(String text, int pos):**

- Inserts the text, starting at specified position.

- **replaceRange(String text, int start, int end):**

- Deletes the text from position start to position end and then insert the specified text in its place.

# **Layout Managers**

# Layout Managers: Outline

---

- How layout managers simplify interface design?
- Standard layout managers
  - `FlowLayout`, `BorderLayout`, `CardLayout`, `GridLayout`,  
`GridBagLayout`, `BoxLayout`
- Positioning components manually
- Strategies for using layout managers effectively

# Layout Managers

---

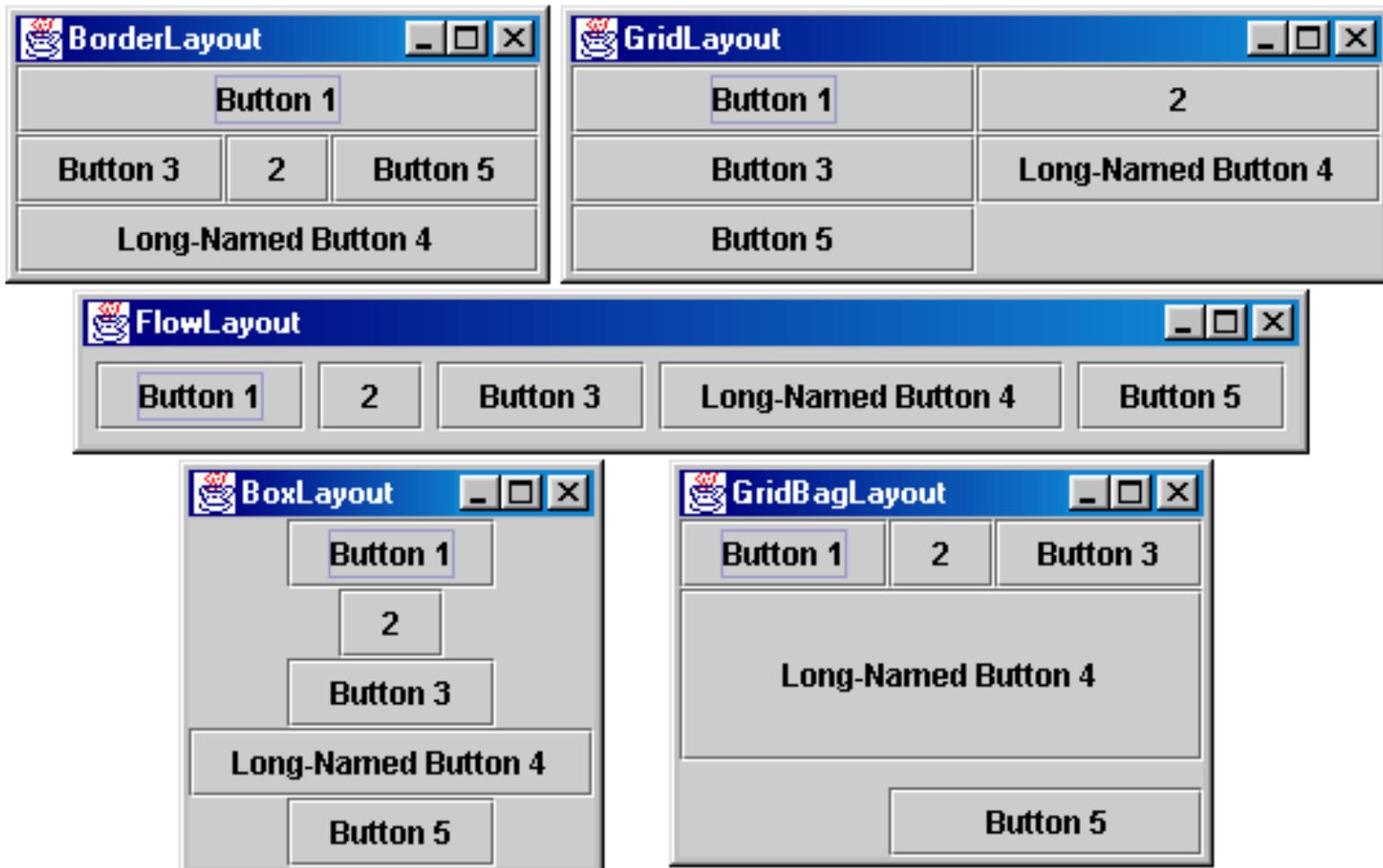
- So far we've just been dumping GUI components into containers. By default components are laid out left-to-right and top-to-bottom
- Layout Manager—an object that decides how components will be arranged in a container
- Used because containers can change size
- Each type of layout manager has rules about how to rearrange components when the size or shape of the container changes

# Layout Managers

---

- Assigned to each Container
  - Give *sizes* and *positions* to components in the window
  - Helpful for windows whose size changes or that display on multiple operating systems
- Relatively easy for simple layouts
  - But, it is surprisingly hard to get complex layouts with a single layout manager
- Controlling complex layouts
  - Use nested containers (each with its own layout manager)
  - Turn layout managers off and arrange things manually

# Layout Managers



# FlowLayout

- Default layout for **Panel** and **Applet**

- Behavior

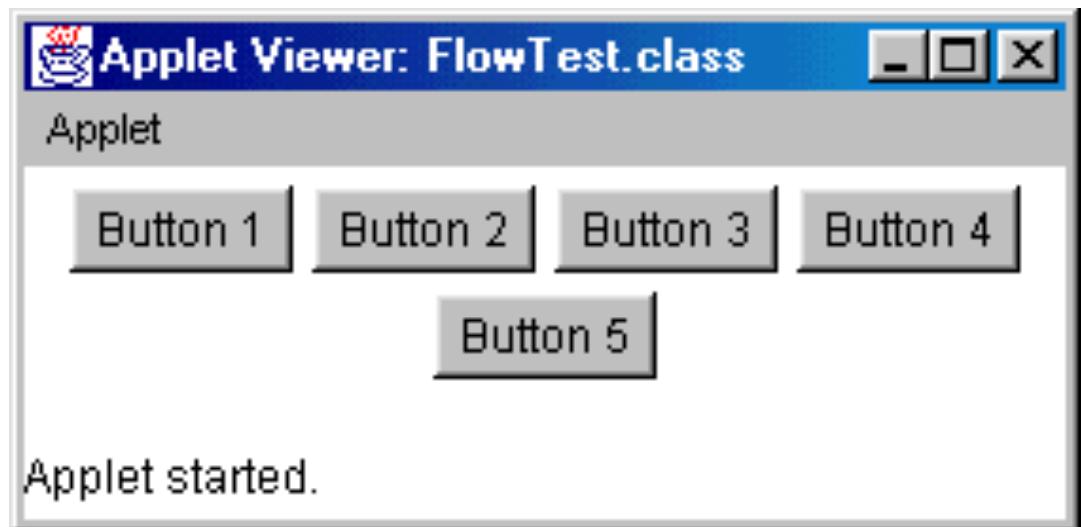
- Resizes components to their **preferred** size
  - Places components in rows **left to right, top to bottom**
  - Rows are **centered** by default

- Constructors

- **FlowLayout()**
    - Centers each row and keeps 5 pixels between entries in a row and between rows
  - **FlowLayout(int alignment)**
    - Same 5 pixels spacing, but changes the alignment of the rows
    - FlowLayout.LEFT, FlowLayout.RIGHT, FlowLayout.CENTER
  - **FlowLayout(int alignment, int hGap, int vGap)**
    - Specify the alignment as well as the horizontal and vertical spacing between components

# FlowLayout: Example

```
public class FlowTest extends Applet {  
    public void init() {  
        // setLayout(new FlowLayout()); [Default]  
        for(int i=1; i<6; i++) {  
            add(new Button("Button " + i));  
        }  
    }  
}
```



# BorderLayout

- Default layout for **Frame** and **Dialog**
- Behavior
  - Divides the Container into **five regions**
  - Each region is identified by a corresponding BorderLayout constant
    - NORTH, SOUTH, EAST, WEST, and CENTER
  - NORTH and SOUTH **respect the preferred height** of the component
  - EAST and WEST **respect the preferred width** of the component
  - CENTER is given the remaining space
- Is allowing a maximum of five components too restrictive? Why not?

# BorderLayout (Continued)

- Constructors

- `BorderLayout()`

- Border layout with no gaps between components

- `BorderLayout(int hGap, int vGap)`

- Border layout with the specified empty pixels between regions

- Adding Components

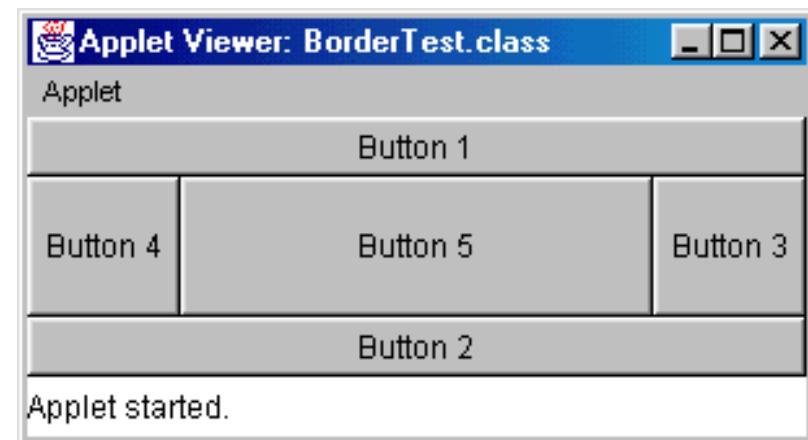
- `add(component, BorderLayout.REGION)`

- Always specify the region in which to add the component

- CENTER is the default, but specify it explicitly to avoid confusion with other layout managers

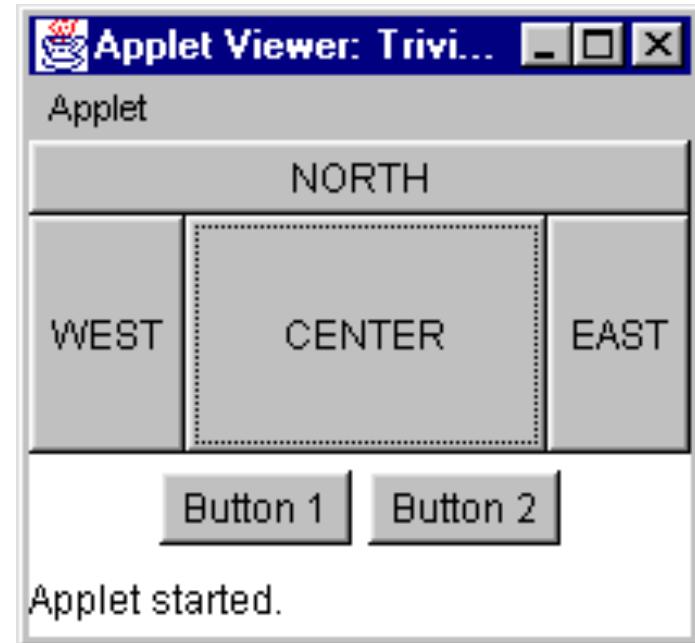
# BorderLayout: Example

```
public class BorderTest extends Applet {  
    public void init() {  
        setLayout(new BorderLayout());  
        add(new Button("Button 1"), BorderLayout.NORTH);  
        add(new Button("Button 2"), BorderLayout.SOUTH);  
        add(new Button("Button 3"), BorderLayout.EAST);  
        add(new Button("Button 4"), BorderLayout.WEST);  
        add(new Button("Button 6"), BorderLayout.CENTER);  
        add(new Button("Button 5"), BorderLayout.CENTER);  
    }  
}
```



# Using a Panel

```
panel p = new Panel();
add (p,BorderLayout.SOUTH);
p.add (new Button ("Button 1"));
p.add (new Button ("Button 2"));
```



# GridLayout

---

- Behavior
  - Divides window into **equal-sized rectangles** based upon the **number of rows and columns specified**
  - Items placed into cells left-to-right, top-to-bottom, based on the order added to the container
  - Ignores the preferred size of the component; each component is **resized to fit into its grid cell**
  - Too few components results in blank cells
  - Too many components results in extra columns

# GridLayout (Continued)

---

- Constructors

- **GridLayout()**

- Creates a single row with one column allocated per component

- **GridLayout(int rows, int cols)**

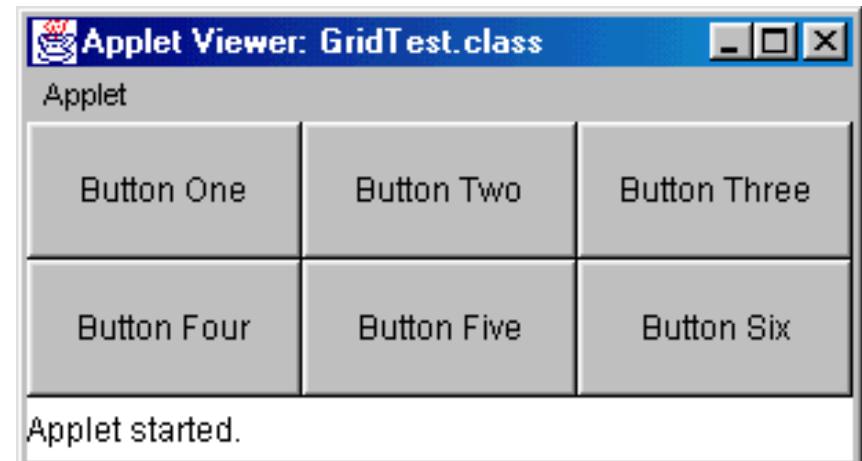
- Divides the window into the specified number of rows and columns
    - Either rows or cols (but not both) can be zero

- **GridLayout(int rows, int cols,  
int hGap, int vGap)**

- Uses the specified gaps between cells

# GridLayout, Example

```
public class GridTest extends Applet {  
    public void init() {  
        setLayout(new GridLayout(2,3)); // 2 rows, 3 cols  
        add(new Button("Button One"));  
        add(new Button("Button Two"));  
        add(new Button("Button Three"));  
        add(new Button("Button Four"));  
        add(new Button("Button Five"));  
        add(new Button("Button Six"));  
    }  
}
```



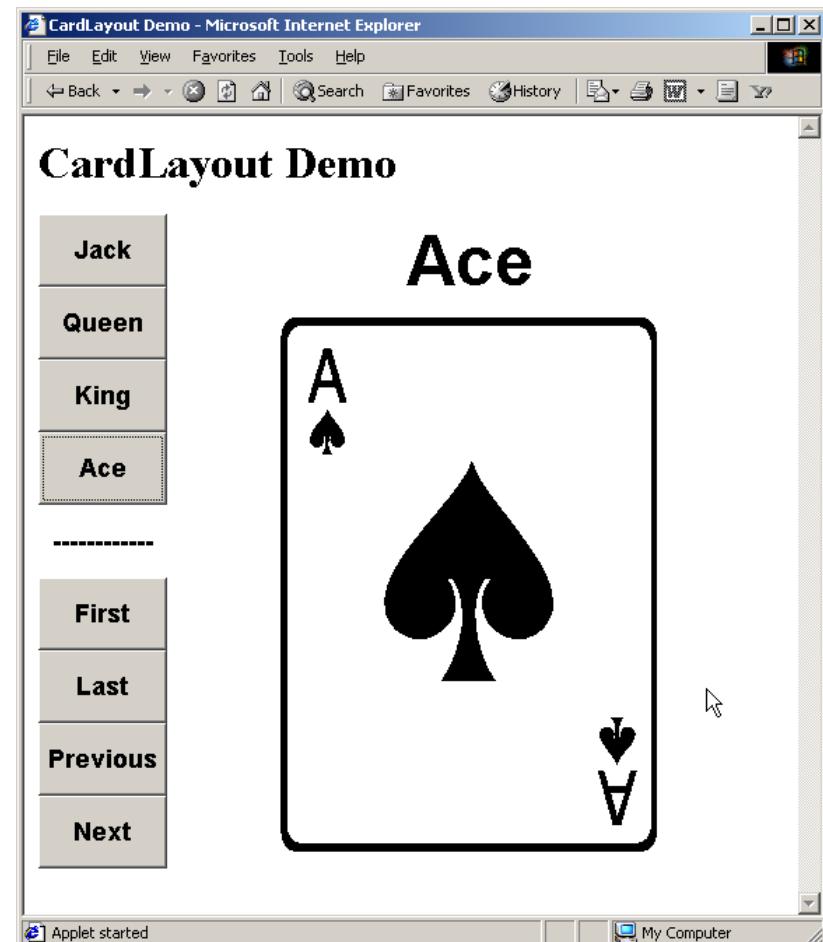
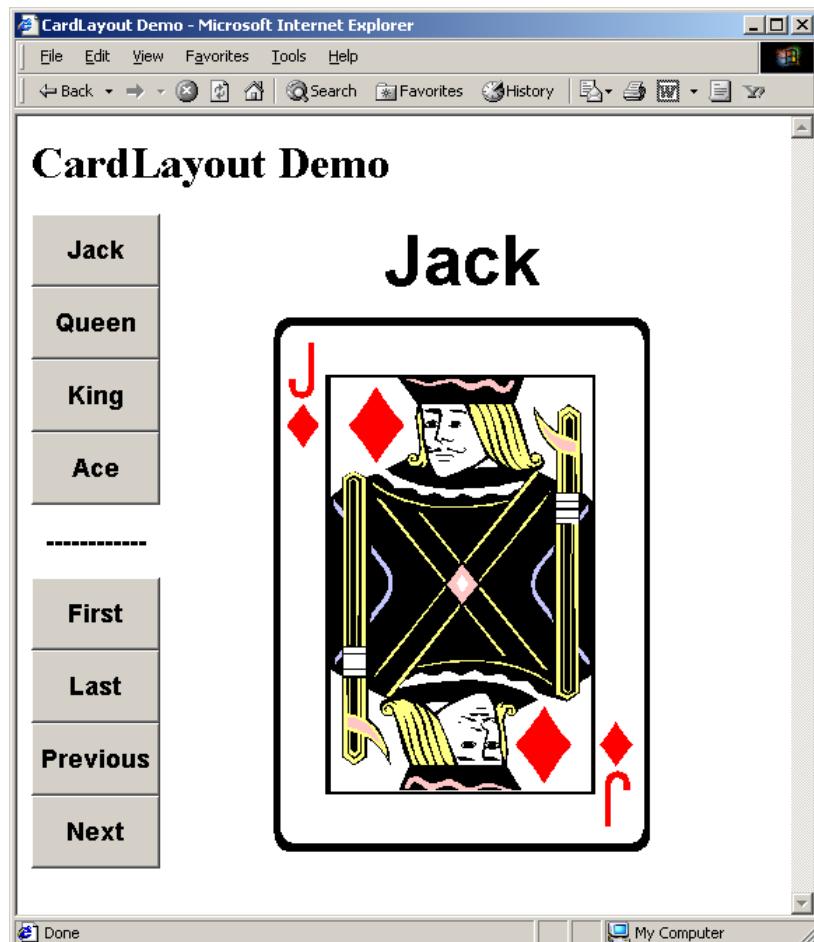
# CardLayout

- Behavior

- Stacks components on top of each other, displaying the top one
- Associates a name with each component in window

```
Panel cardPanel;  
  
CardLayout layout= new CardLayout();  
  
cardPanel.setLayout(layout);  
  
...  
  
cardPanel.add("Card 1", component1);  
cardPanel.add("Card 2", component2);  
  
...  
  
layout.show(cardPanel, "Card 1");  
layout.first(cardPanel);  
layout.next(cardPanel);
```

# CardLayout, Example



# GridBagLayout

---

## – Behavior

- Divides the window into **grids**, without requiring the components to be the same size
- Each component managed by a grid bag layout is associated with an instance of **GridBagConstraints**
  - The GridBagConstraints specifies:
    - » How the component is laid out in the display area
    - » In which cell the component starts and ends
    - » How the component stretches when extra room is available
    - » Alignment in cells

# GridLayout: Basic Steps

- Set the layout, saving a reference to it

```
GridLayout layout = new GridLayout();  
setLayout(layout);
```

- Allocate a GridBagConstraints object

```
GridBagConstraints constraints =  
    new GridBagConstraints();
```

- Set up the GridBagConstraints for component 1

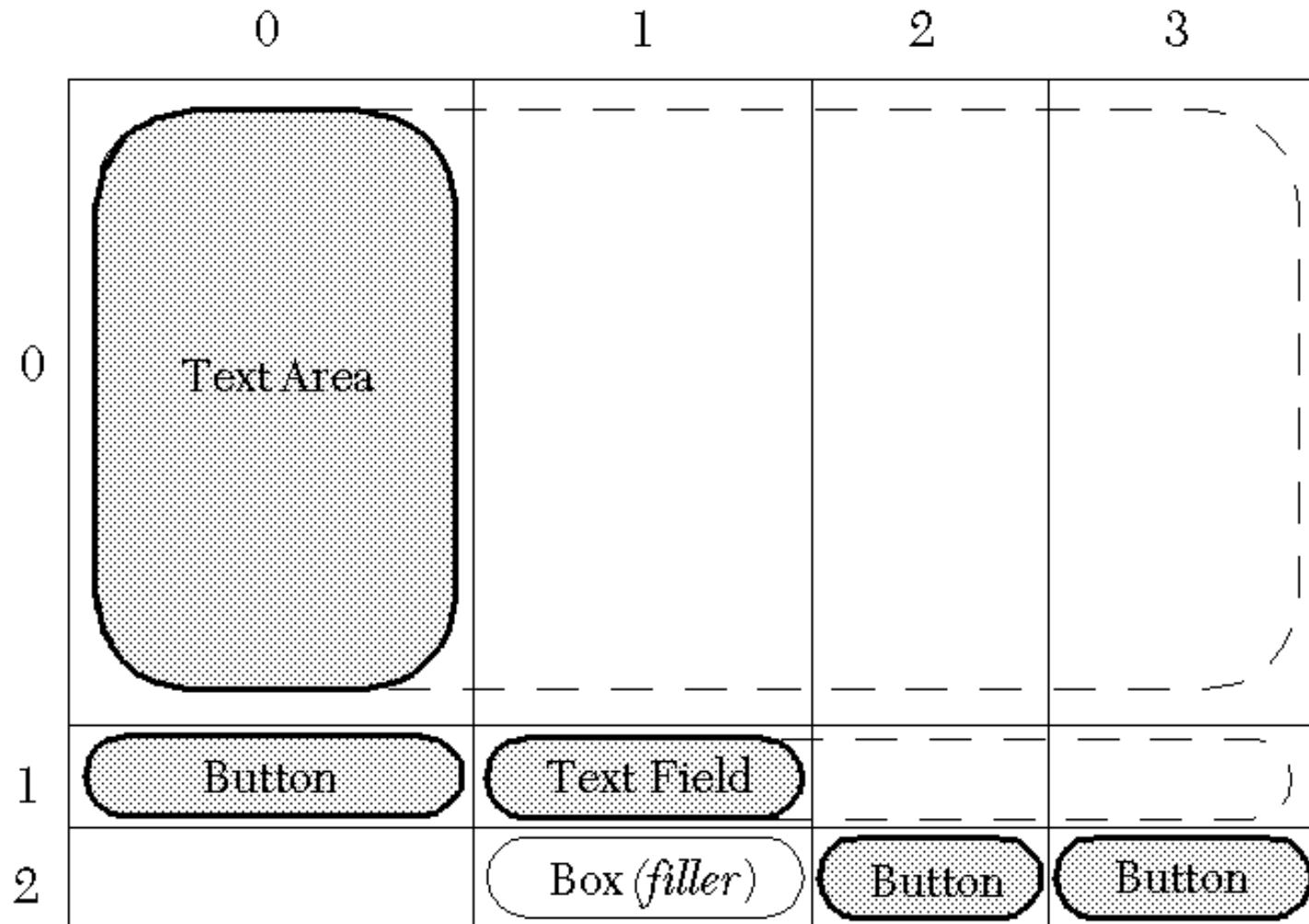
```
constraints.gridx = x1;  
constraints.gridy = y1;  
constraints.gridwidth = width1;  
constraints.gridheight = height1;
```

- Add component 1 to the window, including constraints

```
add(component1, constraints);
```

- Repeat the last two steps for each remaining component

# GridBagLayout: Example

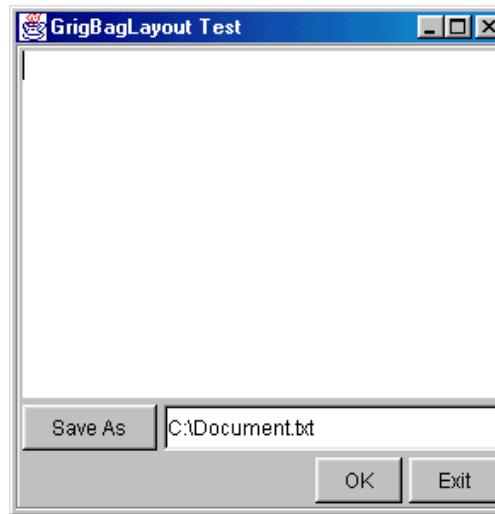


# GridLayout, Example

```
public GridBagTest() {  
    setLayout(new GridLayout());  
    textArea = new JTextArea(12, 40);  
    // 12 rows, 40 cols  
    bSaveAs = new JButton("Save As");  
    fileField = new JTextField("C:\\\\Document.txt");  
    bOk = new JButton("OK");  
    bExit = new JButton("Exit");  
    GridBagConstraints c = new GridBagConstraints();  
    // Text Area.  
    c.gridx      = 0;  
    c.gridy      = 0;  
    c.gridwidth  = GridBagConstraints.REMAINDER;  
    c.gridheight = 1;  
    c.weightx   = 1.0;  
    c.weighty   = 1.0;  
    c.fill       = GridBagConstraints.BOTH;  
    c.insets    = new Insets(2,2,2,2); //t,l,b,r  
    add(textArea, c);  
    ...  
    // Save As Button.  
    c.gridx      = 0;  
    c.gridy      = 1;  
    c.gridwidth  = 1;  
    c.gridheight = 1;  
    c.weightx   = 0.0;  
    c.weighty   = 0.0;  
    c.fill       = GridBagConstraints.VERTICAL;  
    add(bSaveAs,c);  
  
    // Filename Input (Textfield).  
    c.gridx      = 1;  
    c.gridwidth  =  
        GridBagConstraints.REMAINDER;  
    c.gridheight = 1;  
    c.weightx   = 1.0;  
    c.weighty   = 0.0;  
    c.fill       = GridBagConstraints.BOTH;  
    add(fileField,c);  
    ...
```

# GridBagLayout, Example

```
// Exit Button.  
c.gridx = 3;  
c.gridy = 1;  
c.gridheight = 1;  
c.weightx = 0.0;  
c.weighty = 0.0;  
c.fill = GridBagConstraints.NONE;  
add(bExit,c);  
  
// Filler so Column 1 has nonzero width.  
Component filler =  
    Box.createRigidArea(new Dimension(1,1));  
c.gridx = 1;  
c.weightx = 1.0;  
add(filler,c);  
  
...}
```



With / Without Box *filler* at (2,1)



# Disabling the Layout Manager

---

- Behavior
  - If the layout is set to `null`, then components must be **sized** and **positioned** by hand
- Positioning components
  - `component.setSize(width, height)`
  - `component.setLocation(left, top)`
- Or
  - `component.setBounds(left, top, width, height)`

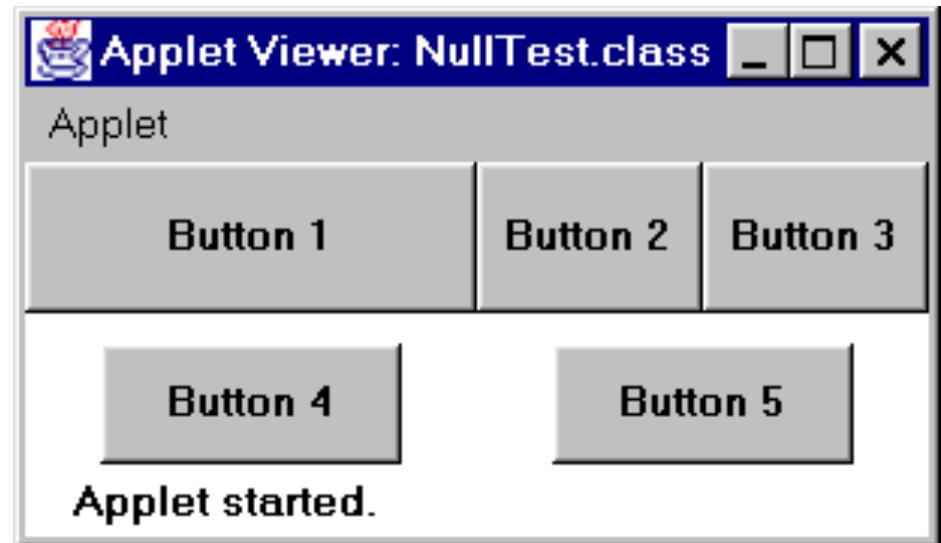
# No Layout Manager, Example

```
setLayout(null);

Button b1 = new Button("Button 1");
Button b2 = new Button("Button 2");

...
b1.setBounds(0, 0, 150, 50);
b2.setBounds(150, 0, 75, 50);
...

add(b1);
add(b2);
...
```



# Using Layout Managers Effectively

---

- Use nested containers
  - Rather than struggling to fit your design in a single layout, try dividing the design into sections
  - Let each section be a panel with its own layout manager
- Turn off the layout manager for some containers
- Adjust the empty space around components
  - Change the space allocated by the layout manager
  - Override insets in the Container

# Nested Containers, Example

```
public NestedLayout() {  
  
    setLayout(new BorderLayout(2,2));  
    textArea = new JTextArea(12,40); // 12 rows, 40 cols  
    bSaveAs = new JButton("Save As");  
    fileField = new JTextField("C:\\\\Document.txt");  
    bOk = new JButton("OK");  
    bExit = new JButton("Exit");  
    add(textArea,BorderLayout.CENTER);  
    // Set up buttons and textfield in bottom panel.  
    JPanel bottomPanel = new JPanel();  
    bottomPanel.setLayout(new GridLayout(2,1));
```

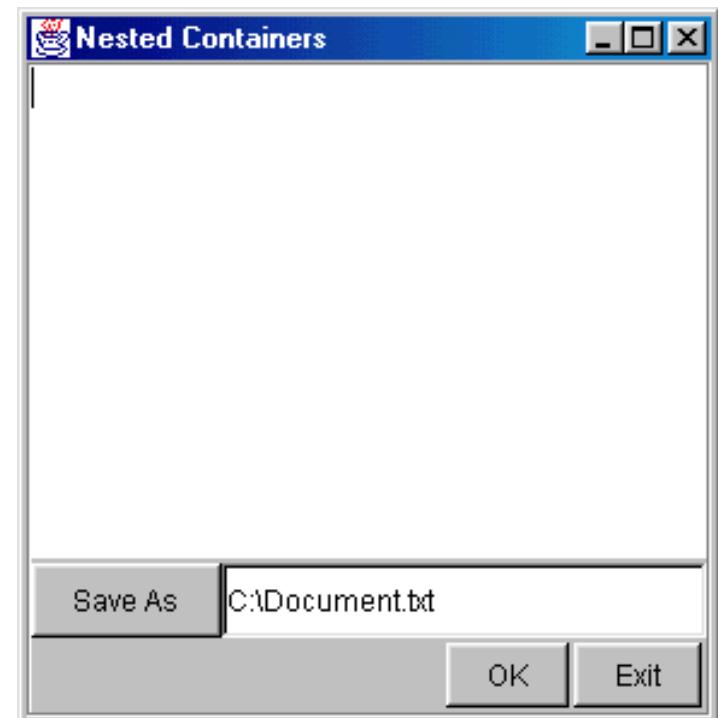
# Nested Containers, Example

```
JPanel subPanel1 = new JPanel();
JPanel subPanel2 = new JPanel();
subPanel1.setLayout(new BorderLayout());
subPanel2.setLayout(new FlowLayout(FlowLayout.RIGHT, 2, 2));

subPanel1.add(bSaveAs, BorderLayout.WEST);
subPanel1.add(fileField, BorderLayout.CENTER);
subPanel2.add(bOk);
subPanel2.add(bExit);

bottomPanel.add(subPanel1);
bottomPanel.add(subPanel2);

add(bottomPanel, BorderLayout.SOUTH);
}
```

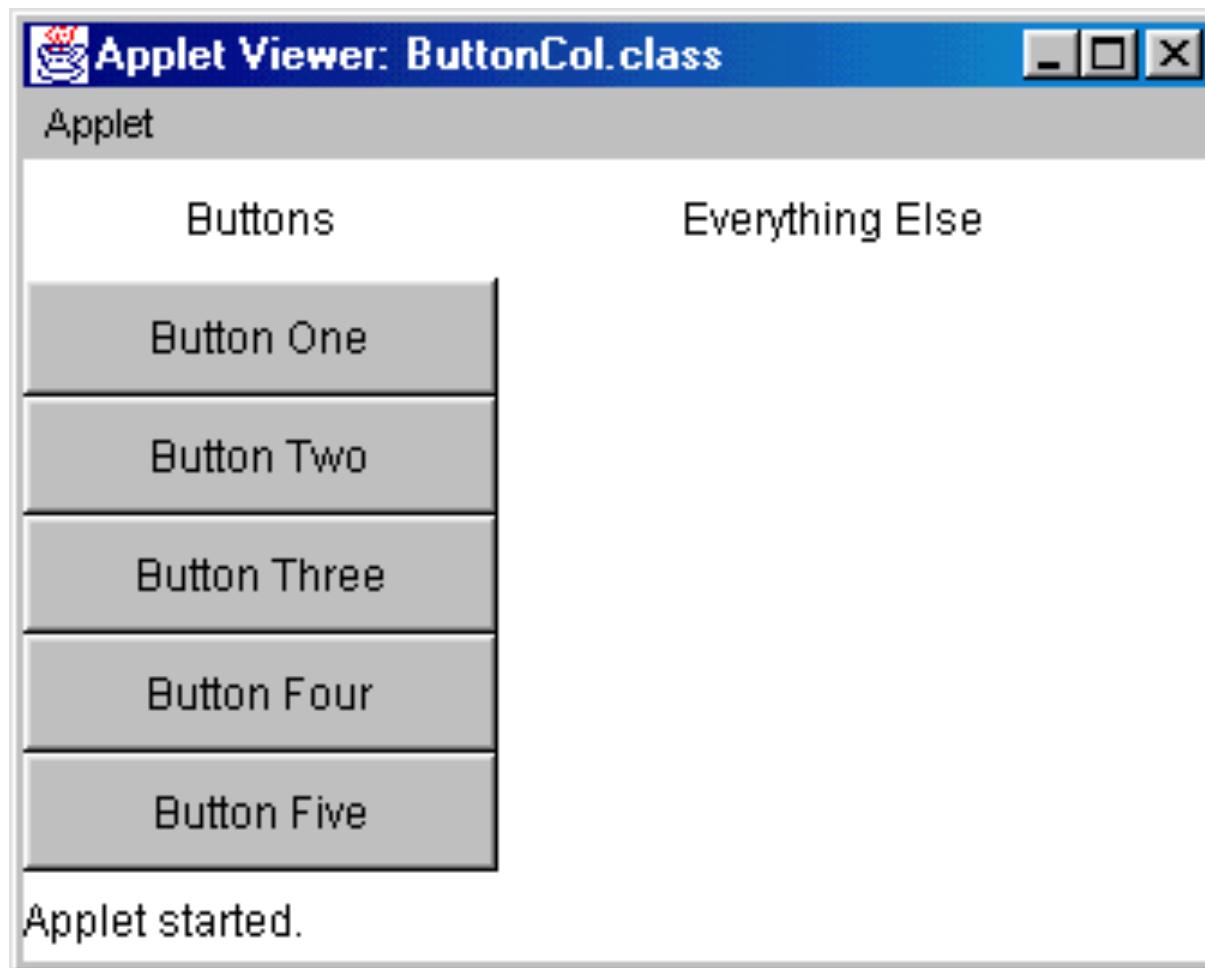


## Turning Off Layout Manager for Some Containers, Example

- Suppose that you wanted to arrange a column of buttons (on the left) that take exactly 40% of the width of the container

```
setLayout(null);  
int width1 = getSize().width*4/10;,  
int height = getSize().height;  
Panel buttonPanel = new Panel();  
buttonPanel.setBounds(0, 0, width1, height);  
buttonPanel.setLayout(new GridLayout(6, 1));  
buttonPanel.add(new Label("Buttons", Label.CENTER));  
buttonPanel.add(new Button("Button One"));  
...  
buttonPanel.add(new Button("Button Five"));  
add(buttonPanel);  
Panel everythingElse = new Panel();  
int width2 = getSize().width - width1,  
everythingElse.setBounds(width1+1, 0, width2, height);
```

# Turning Off Layout Manager for Some Containers: Result



# Adjusting Space Around Components

---

- Change the space allocated by the layout manager
  - Most LayoutManagers accept a horizontal spacing (hGap) and vertical spacing (vGap) argument
  - For GridBagLayout, change the insets
- Use a Canvas or a Box as an invisible spacer
  - For **AWT** layouts, use a **Canvas** that does not draw or handle mouse events as an “empty” component for spacing.
  - For **Swing** layouts, add a **Box** as an invisible spacer to improve positioning of components

# Summary

---

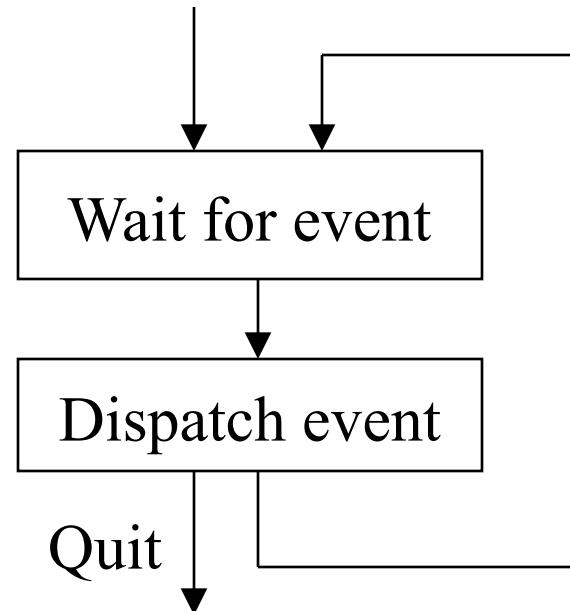
- Default layout managers
  - Applet and Panel: FlowLayout
  - Frame and Dialog: BorderLayout
- Layout managers respect the preferred size of the component differently
- GridBagConstraints is the most complicated but most flexible manager
  - Use GridBagConstraints to specify the layout of each component
- Complex layouts can often be simplified through nested containers
- In AWT use a Canvas as a **spacer**; in Swing use a Box as a spacer

*Welcome to*

# Event Handling

# Modern event-driven programs

- Programs respond to events that are generated *outside* the control of the program.



# Java hides the event loop

---

- The event loop is built into Java GUIs
  - GUI stands for Graphical User Interface
- Interacting with a GUI component (such as a button) causes an event to occur
- An **Event** is an object
- You create **Listener**s for interesting events
- The **Listener** gets the **Event** as a parameter

# Building a GUI

---

- To build a GUI in Java,
  - Create some **Components**
  - Use a layout manager to arrange the **Components** in a window
  - Add **Listener**s, usually one per **Component**
  - Put code in the **Listener**s to do whatever it is you want done
- That's it!
  - Of course, there are a lot of details....

# Vocabulary I

---

- **Event**

- an object representing an external happening that can be observed by the program

- **event-driven programming**

- A style of programming where the main thing the program does is respond to Events

- **event loop**

- a loop that waits for an Event to occur, then dispatches it to the appropriate code

- **GUI**

- a Graphical User Interface (user interacts with the program via things on the screen)

# Vocabulary II

---

- Component
  - an interface element, such as a Button or a TextField
- Layout Manager
  - an object (provided by Java) that arranges your Components in a window
- Listener
  - an object you create to execute some code appropriate when an Event occurs

# Event Handling

---

- With event-driven programming, events are detected by a program and handled appropriately
- Events: Objects that represent user initiated actions
  - moving the mouse
  - clicking the button
  - pressing a key
  - sliding the scrollbar thumb
  - choosing an item from a menu

# Three Steps of Event Handling

---

## Delegation Event Model

- Prepare to accept events
  - import package java.awt.event
- Start listening for events
  - include appropriate methods
- Respond to events
  - implement appropriate abstract method

# 1. Prepare to accept events

---

- Import package `java.awt.event`
- Listeners are *interfaces*, not classes
- Example:
  - “`ActionListener`” for Button events
  - “`AdjustmentListener`”  
for Scrollbar events

## 2. Start listening for events

---

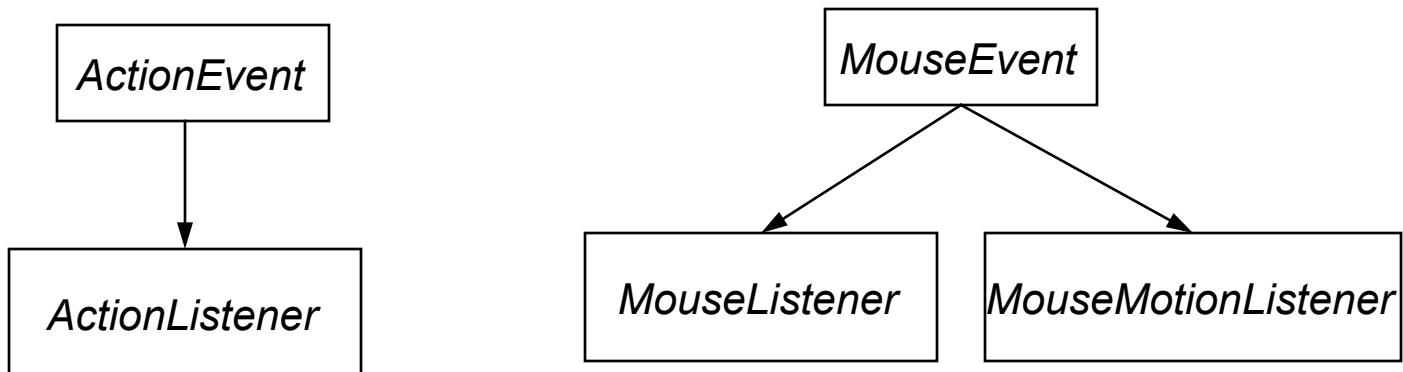
- To make the applet “listen” to a particular event, include the appropriate “addxxxListener”.
- Examples: For a Button, you need an *ActionListener*  
`b1.addActionListener(this)`
  - shows that the applet is interested in listening to events generated by the pushing of a certain button.

## 2. Start listening for events (cont)

- Example

```
addAdjustmentListener(this)
```

- shows that the applet is interested in listening to events generated by the sliding of a certain scroll bar thumb.
- “this” refers to the applet itself - “me” in English



### 3. Respond to events

---

- When you say **implements**, you are *promising* to supply those methods
- The appropriate abstract methods are implemented.
- Example:
  - `actionPerformed()` is automatically called whenever the user clicks the button.
- Thus, implement `actionPerformed()` to respond to the button event.

### **3. Respond to events (cont)**

---

- Example:

**adjustmentValueChanged()** is automatically invoked whenever the user slides the scroll bar thumb.

So **adjustmentValueChanged()** needs to be implemented.

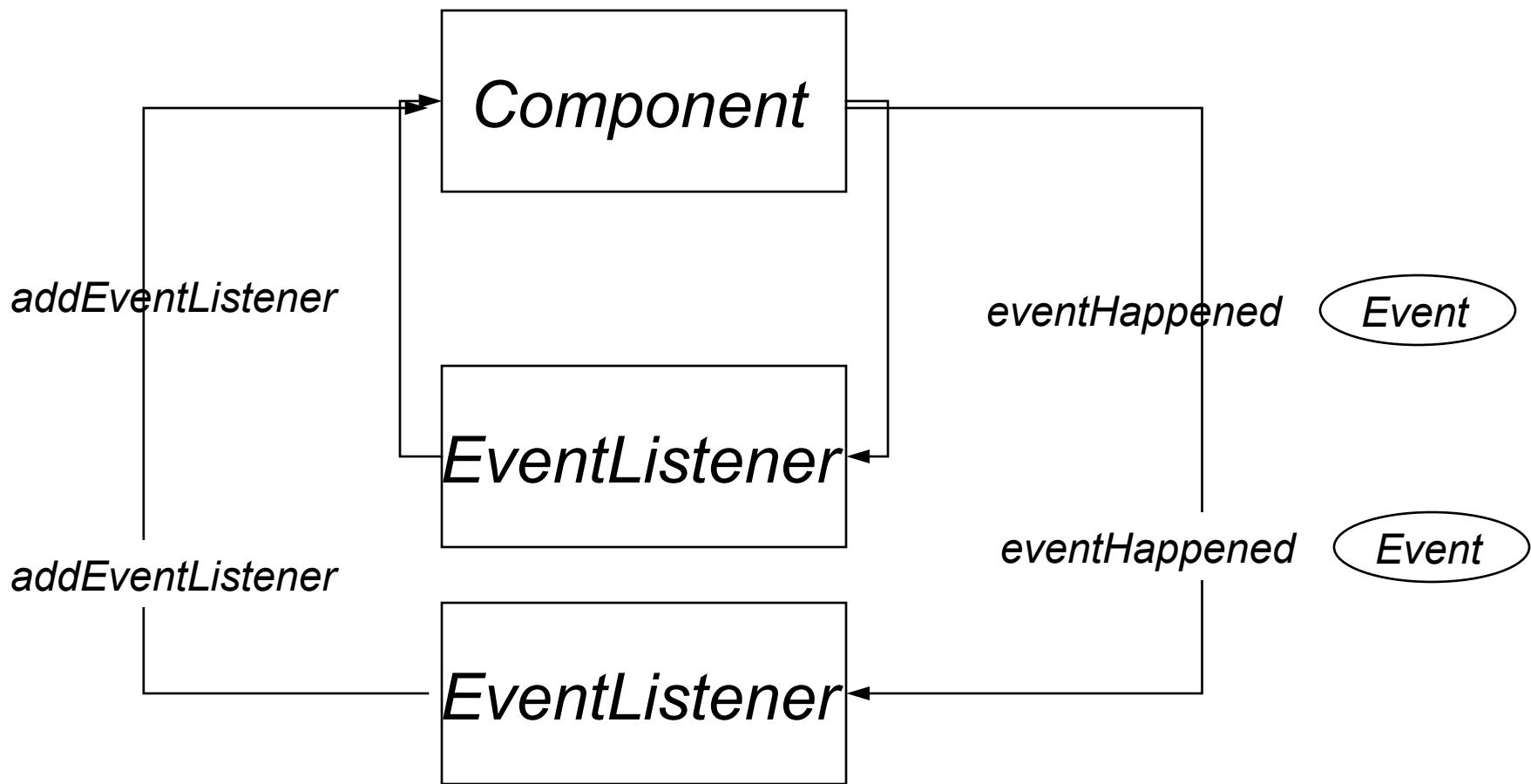
- In `actionPerformed(ActionEvent evt)`, `ActionEvent` is a class in `java.awt.event`.

# Registering Listeners

---

- Listeners register themselves with component
  - public void addXXXListener(XXXListener)
  - addActionListener, addItemListener, etc.
- Multiple listeners can be registered for the same event on a component
  - One event can trigger numerous responses
  - Events are broadcast to all listeners

# Multiple Listeners



# Example Listeners

---

```
public interface ActionListener {  
    public void actionPerformed(ActionEvent e);  
}  
  
public interface ItemListener {  
    public void itemStateChanged(ItemEvent e);  
}  
  
public interface ComponentListener {  
    public void componentHidden(ComponentEvent e);  
    public void componentMoved(ComponentEvent e);  
    public void componentResized(ComponentEvent e);  
    public void componentShown(ComponentEvent e);  
}
```

# A mouse event

---

- Mouse methods respond to events. An event is an action related to the mouse. The source of this event is a mouse.

# MouseListener

---

```
public interface MouseListener
{
    void mouseClicked (MouseEvent event);
        // When mouse clicked on a UI component

    void mouseEntered (MouseEvent event);
        // Mouse is first over a component

    void mouseExited (MouseEvent event);
        // Mouse has just left a component

    void mousePressed (MouseEvent event);
        // Button presses down on a component

    void mouseReleased (MouseEvent event);
        // User lets go of a pressed-down mouse button
}
```

# Standard AWT Event Listeners (Details Continued)

---

- **ContainerListener**

- Triggered when window adds/removes GUI controls
  - `componentAdded(ContainerEvent event)`
  - `componentRemoved(ContainerEvent event)`

- **FocusListener**

- Detects when controls get/lose keyboard focus
  - `focusGained(FocusEvent event)`
  - `focusLost(FocusEvent event)`

# Standard AWT Event Listeners (Details Continued)

---

- **ItemListener**

- Handles selections in lists, checkboxes, etc.
  - `itemStateChanged(ItemEvent event)`

- **KeyListener**

- Detects keyboard events
  - `keyPressed(KeyEvent event)` -- any key pressed down
  - `keyReleased(KeyEvent event)` -- any key released
  - `keyTyped(KeyEvent event)` -- key for printable char released

# Standard AWT Event Listeners (Details Continued)

---

- MouseMotionListener
  - Handles mouse movement
    - mouseMoved(MouseEvent event)
    - mouseDragged(MouseEvent event)

# Standard AWT Event Listeners

---

- **TextListener**
  - Applies to textfields and text areas
    - `textValueChanged(TextEvent event)`
- **WindowListener**
  - Handles high-level window events
    - `windowOpened`, `windowClosing`, `windowClosed`, `windowIconified`,  
`windowDeiconified`, `windowActivated`, `windowDeactivated`
      - `windowClosing` particularly useful

# Window Closing

---

- A very common event directed towards a window is a *close* event.
- The default behavior is to simply hide the Frame when the user closes the window.
- We prefer that the program terminate when the user closes the main window.
  - Two steps are required to accomplish this:
    - Write an event handler for the close event that will terminate the program.
    - Register the handler with the appropriate event source.

# Action Listener

---

- Generally one ActionListener will be responsible for handling the events generated by a group of buttons.
  - The getActionCommand() method determines which button was pressed
- Selection type events (ItemEvent) generate two events per user action (selection & deselection)

# Listeners for TextFields

---

- An ActionListener listens for hitting the return key

- An ActionListener demands

```
public void actionPerformed(ActionEvent e)
```

- use getText( ) to get the text

- A TextListener listens for any and all keys

- A TextListener demands

```
public void textValueChanged(TextEvent e)
```

# Buttons

---

- Buttons generate **action** events.
- The ActionListener interface
  - void actionPerformed(ActionEvent event);
  - Note that there is no need for an ActionAdapter class

# MyButtonListener

```
public void init () {  
    ...  
    b1.addActionListener (new MyButtonListener ());  
}  
  
class MyButtonListener implements ActionListener {  
    public void actionPerformed (ActionEvent e) {  
        showStatus ("Ouch!");  
    }  
}
```



# Adapters

---

- A class that implements a Listener Interface with empty methods
- Only useful for Listeners with more than one method *Interface*
- Primarily for convenience

# Adapter Classes

| Adapter class       | Listener Interface   |
|---------------------|----------------------|
| Component Adapter   | Component Listener   |
| Container Adapter   | Container Listener   |
| Focus Adapter       | Focus Listener       |
| Key Adapter         | Key Listener         |
| Mouse Adapter       | Mouse Listener       |
| MouseMotion Adapter | MouseMotion Listener |
| Window Adapter      | Window Listener      |

# Inner Classes

---

- Inner classes can access all the instance variables and methods of the outer class
- A separate compiled file is generated for each class in a program file

# Inner Classes

---

- Java Classes can be defined anywhere
  - Nested inside other classes
  - Method calls
- Have access to all outer classes members and methods
- Can be named or anonymous
- Can be extensions of Classes or implementations of Interfaces
- Very useful in Event Handling

# Named Inner Classes

---

- Defined just like normal classes
- Cannot be *public*

```
public class ApplicationFrame {  
    ....  
    class ButtonHandler implements ActionListener {  
        Public void actionPerformed(ActionEvent e) {  
            doTheOKThing();  
        }  
    }  
  
    private void doTheOKThing() { // here's where I handle OK  
    }  
    ....  
    JButton okButton = new JButton("OK");  
    okButton.addActionListener(new ButtonHandler()); // create inner class listener  
    ....
```

# Anonymous Inner Classes

---

- Inner classes with no name
- Defined within the addXXXListener method

```
Public class ApplicationFrame {  
  
    ....  
    JButton okButton = new JButton("OK");  
    okButton.addActionListener(new ActionListener() {  
        public void actionPerformed(ActionEvent event) {  
            doTheOKThing();  
        }  
    } );  
    ....  
    private void doTheOKThing() { // here's where I handle the OK  
    }  
    ....
```

# Why Use Inner Classes?

---

- Code is much more readable
  - Non trivial event handling in real-life apps
  - Avoid long case statements
  - Name handling methods something meaningful
  - Break up the event handling
  - Simplifies interfaces for main application classes
- Many online examples use them

# **Graphics**

# Designing Graphics

---

- AWT has a collection of classes (packages) that offer the means for representing basic shapes, components and events
- Java 2D API includes a new set of classes for processing shapes, text and images
- `java.awt.Graphics`, `java.awt.Graphics2D` and `java.awt.event`

# Color in Java

---

- The class Color provides methods and constants for manipulating colors in Java
- Color values are determined by a three primary colors: red, green, and blue values (RGB)

# Color in Java

---

- A fundamental method:
- Public Color (int r, int g, int b)
  - Creates a color based on red, green and blue values expressed in integer

# Creating a color in Java

---

Code example:

```
Color my_color = new Color( 255, 0, 0);
```

Color can be created based on pre-established constants:

```
Color my_color = Color.orange;
```

# Color Constants

There are several color constants in Java

| Color Constant | RGB Value   |
|----------------|-------------|
| Color Red      | 255,0,0     |
| Color Green    | 0,255,0     |
| Color Blue     | 0,0,255     |
|                |             |
| Color Orange   | 255,200,0   |
| Color Pink     | 255,175,175 |
| Color Yellow   | 255,255,0   |
| Color Gray     | 128,128,128 |

# Color in Java

---

- Another way to set color in Java
  - Using color constants
  - `g.setColor( Color.blue )`
  - Using three values for RGB
  - `g.setColor( new Color( 255, 0, 0) )`

# Drawing with Graphics Class

---

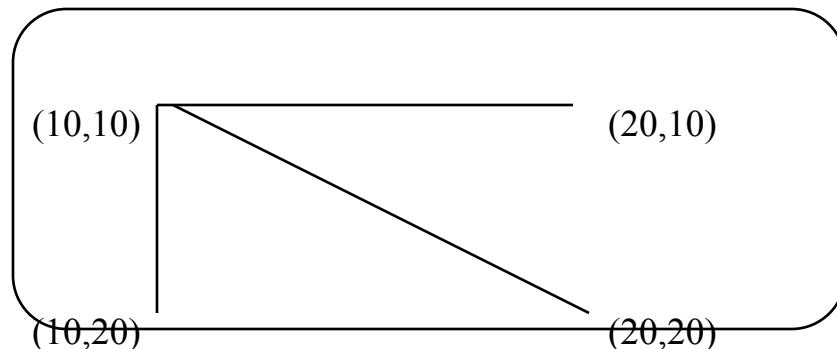
- To draw a red line:

```
public void paint( Graphics g ) {  
    g.setColor( Color.red );  
    g.drawLine(5, 30, 350, 30 );  
}
```

# Program Example: Drawing Lines

- Can draw any line (in pixel width) using two pairs of coordinate values

```
Public void paint (Graphics g) {  
    // drawing a line from (10,10) to (20,10)  
    // then drawing another line (10,10) to (10,20)  
    // then drawing another line (10,10) to (20,20)  
    g.drawLine(10,10,20,10);  
    g.drawLine(10,10,10,20);  
    g.drawLine(10,10,20,20);  
}
```



# Drawing with Graphics Methods

---

- Some of the other important methods are:

- `drawRect( int x, int y, int width, int height)`

- Drawing a rectangle, top left corner is at x,y

- `drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight )`

- Drawing a rounded edged rectangle

# Drawing with Graphics Methods

- Important drawing methods ...

- `draw3DRect( int x, int y, int width, int height, boolean b)`

- Draw raised rectangle

- `drawOval( int x, int y, int width, int height)`

- Draw an oval, with top left corner of the bounding rectangle at x, y

# Drawing Shapes (2-D)

---

- Also available are “filling” methods that can be used to create solid rectangles or ovals filled with a certain color

```
public void paint( Graphics g) {  
  
    g.setColor( Color.yellow );  
    g.fillOval( 290, 100, 90, 55 );  
  
}
```

# Drawing multi-sided Shapes

---

- Two major types: using coordinate values or using Polygon class
- Can use the Graphics methods:
- `drawPolygon()`, `drawPolyline()`, `fillPolygon()`

# Using the Polygon Class

---

- Code example:

```
Int xValues[] = { 20, 40, 50, 30, 20, 15 };  
Int yValues[] = { 50, 50, 60, 80, 80, 60 };  
Polygon poly = new PolyGon( xValues, yValues, 6 );
```

```
//drawing the polygon
```

```
g.drawPolygon( poly );
```

# Using Graphics method for drawing Polygons

---

- Code example:

```
int xValues[] = { 70, 90, 100, 80, 70, 65, 60 };
int yValue[] = { 100, 100, 110, 130, 110, 90 };

g.drawPolyline( xValues, yValues, 7 );
```

# Using Java Fonts

---

- Fonts have logical **names**: Serif, MonoSpaced, SansSerif, etc.
- Fonts have **styles**: Plain, Italic, or Bold
- Fonts have **sizes** (e.g., 18 point size)

# Using Java Font Class

---

- Code example:

```
Font my_font;
```

```
my_font = new Font( "Serif", Font.Bold, 12);
```

```
g.setFont( my_font );
```

```
g.drawString( "Serif 12 point bold font", 20, 50);
```

# Using Font with Colors

---

```
g.setColor( Color.red );
```

```
String font_name = g.getFont().getFontName();
```

```
g.setFont( new Font( font_name, Font.Bold, 12 ) );  
g.drawString( "Changed font appearance", 20, 110);
```

# Getting system fonts

---

- You can get all system fonts that are available.

```
GraphicsEnvironment e =
```

```
    GraphicsEnvironment.getLocalGraphicsEnvironment();
```

```
Font[] fonts = e.getAllFonts(); // get the fonts
```

# Unit Summary

---

- general organization of AWT
- component class hierarchy.
- Handling events
- Implementing listeners
- java's layout Manager.
- setting components layout manager
- Using inner classes.
- Graphics in Java

*Welcome to*  
**Reflection**

Java's mechanism for dynamic programming

# Unit Objective

---

- After completing this unit you should be able to:
  - What is reflection in Java
  - Need of reflection
  - `Java.lang.reflect.*` package
  - Implementing and testing reflection.

# Reflection API

---

- What if you want to access information not just about the Object, but about that Object's Class?
- What if you want to access a method, field, or constructor whose name you don't know at compile time?

# What can you do with reflection?

---

- Determine the class of an object.
- Get information about a class's modifiers, fields, methods, constructors, and superclasses.
- Find out what constants and method declarations belong to an interface.

# What can you do with reflection?

---

- Create an instance of a class whose name is not known until runtime.
- Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
- Invoke a method on an object, even if the method is not known until runtime.
- Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

# Classes in the Reflection API

---

- class Class
- class Constructor
- class Field
- class Method

# Class

---

- Represents, or reflects, the class.
- Exists for each class
- Immutable
- Maintained by the Java Runtime Environment (JRE)
- With the reflection API, you can invoke methods on a Class object to get return Constructor, Method, and Field objects.
  - Can call methods to get use these objects to get information about the corresponding constructors, methods, and fields defined in the class.

# Getting the Class Object – Way 1

---

- Invoke `Object.getClass()`.
  - Need an instance of the class
  - Useful when you have the object, but don't know its class.

```
Class c = mystery.getClass();
```

# Getting the Class Object – Way 2

---

- `getSuperclass` method
  - Retrieves the `Class` object for the superclass that another `Class` object reflects
  - Need an instance of `Class`

```
TextField t = new TextField();
Class c = t.getClass(); // TextField
Class s = c.getSuperclass(); //
```

TextComponent

# Getting the Class Object – Way 3

---

- `ClassName.class`
  - Must know the name of the class at compile time

```
Class c = java.awt.Button.class;
```

```
Class o = Object.class;
```

# Getting the Class Object – Way 4

---

- `Class.forName("className")`
  - Class name doesn't have to be known at compile time
  - Typically used to load database drivers at runtime

```
String s = "java.util.Date";
// can also prompt for classname
Class c = Class.forName(s);
```

# Class Methods

---

- Let's take a look at some of the methods available in Class...
- Write a method that, given an instance of some class, prints the names of the classes its inheritance hierarchy

# Code

---

```
import java.lang.reflect.*;
import java.awt.*;
class SampleSuper {
    public static void main(String[] args) {
        Button b = new Button();
        printSuperclasses(b);
    }
    static void printSuperclasses(Object o) {
        Class subclass = o.getClass();
        Class superclass = subclass.getSuperclass();
        while (superclass != null) {
            String className = superclass.getName();
            System.out.println(className);
            subclass = superclass;
            superclass = subclass.getSuperclass();
        }
    }
}
```

java.awt.Component  
java.lang.Object

# Class Modifiers

---

- A class declaration may include the following modifiers:
  - public, abstract, and/or final.
- The class modifiers precede the class keyword in the class definition.
- int *getModifiers()* method in Class returns the Java language modifiers for this class or interface, encoded in an integer.
  - Use the isPublic, isAbstract, and isFinal static methods in Modifier to determine which modifiers exist

# Class Modifiers Example

```
import java.lang.reflect.*;

class SampleModifier {

    public static void main(String[] args) {
        String s = new String();
        printModifiers(s);
    }

    public static void printModifiers(Object o) {
        Class c = o.getClass();
        int m = c.getModifiers();
        if (Modifier.isPublic(m))
            System.out.println("public");
        if (Modifier.isAbstract(m))
            System.out.println("abstract");
        if (Modifier.isFinal(m))
            System.out.println("final");
    }

    public
    final
```

# Identifying the Interfaces Implemented by a Class

```
void printInterfaceNames(Object o) {  
    Class c = o.getClass();  
    Class[] interfaces = c.getInterfaces();  
  
    for (int i = 0; i < theInterfaces.length; i++) {  
        String interfaceName = interfaces[i].getName();  
        System.out.println(interfaceName);  
    }  
}
```

# Identifying the Interfaces Implemented by a Class

```
import java.lang.reflect.*;
import java.io.*;
class SampleInterface {
    public static void main(String[] args) {
        try {
            RandomAccessFile r = new
                RandomAccessFile("myfile", "r"); printInterfaceNames(r);
            } catch (IOException e) {
                System.out.println(e);
            }
        }
    static void printInterfaceNames(Object o) {
        Class c = o.getClass();
        Class[] theInterfaces = c.getInterfaces();
        for (int i = 0; i < theInterfaces.length; i++) { String interfaceName =
            theInterfaces[i].getName(); System.out.println(interfaceName);
        }
    }
}
```

java.io.DataOutput  
java.io.DataInput

# Fields

---

- Refers to attributes of a class
- Call `Class.getFields()` to get an array of **public** fields
- Call `getDeclaredFields()` to get an array of **all** fields
- Let's examine some methods in the `Field` class

# Getting Fields

```
import java.lang.reflect.*;
import java.awt.*;

class SampleField {

    public static void main(String[] args) {
        GridBagConstraints g = new GridBagConstraints();
        printFieldNames(g);
    }

    static void printFieldNames(Object o) {
        Field[] publicFields = o.getClass().getFields();
        for (int i = 0; i < publicFields.length; i++) {
            String fieldName = publicFields[i].getName();
            Class typeClass = publicFields[i].getType();
            String fieldType = typeClass.getName();
            System.out.println("Name: " + fieldName +
                ", Type: " + fieldType);
        }
    }
}
```

# Manipulating Objects

---

- Software development tools, such as GUI builders and debuggers, need to manipulate objects at runtime.
  - For example, a GUI builder may allow the end-user to select a Button from a menu of components, create the Button object, and then click the Button while running the application within the GUI builder

# Setting Field Values

---

- To modify the value of a field you have to:
  - Create a Class object
  - Create a Field object by invoking `getField` on the Class
  - Invoke the appropriate set method on the Field object
- The Field class provides several set methods.
  - Specialized methods, such as `setBoolean` and `setInt`, are for modifying primitive types.
  - If the field you want to change is an object invoke the set method.
    - You can call set to modify a primitive type, but you must use the appropriate wrapper object for the value parameter

# Setting Fields

---

```
static void modifyWidth(Rectangle r, Integer widthParam)
{
    Field widthField;
    Integer widthValue;
    Class c = r.getClass();
    try {
        widthField = c.getField("width");
        widthField.set(r, widthParam);
    } catch (NoSuchFieldException e) {
        System.out.println(e);
    } catch (IllegalAccessException e) {
        System.out.println(e);
    }
}
```

# Methods

---

- To find out what public methods belong to a class, invoke the method named `getMethods`.
- You can use a `Method` object to uncover a method's name, return type, parameter types, set of modifiers, and set of throwable exceptions.
- With `Method.invoke`, you can even call the method itself.
- Let's look at the methods available in the `Method` class

# Steps for Invoking Methods

---

- Create a Method object by invoking getMethod on the Class object.
  - The getMethod method has two arguments
    - A String containing the method name
    - An array of Class objects for the parameters
- Invoke the method by calling invoke.
  - The invoke method has two arguments
    - An array of argument values to be passed to the invoked method,
    - An object whose class declares or inherits the method

# Example Start

---

```
import java.lang.reflect.*;

class SampleInvoke {

    public static void main(String[] args) {
        String firstWord = "Hello ";
        String secondWord = "everybody.";
        String bothWords = append(firstWord, secondWord);
        System.out.println(bothWords);
    }
}
```

# Invoking Methods

```
public static String append(String firstWord, String secondWord) {  
    String result = null;  
    Class c = String.class;  
    Class[] parameterTypes = new Class[] {String.class};  
    Method concatMethod;  
    Object[] arguments = new Object[] {secondWord};  
    try {  
        concatMethod = c.getMethod("concat", parameterTypes);  
        result = (String) concatMethod.invoke(firstWord, arguments);  
    } catch (NoSuchMethodException e1) {  
        System.out.println(e1);  
    } catch (IllegalAccessException e2) {  
        System.out.println(e2);  
    } catch (InvocationTargetException e3) {  
        System.out.println(e3);  
    }  
    return result;  
}
```

# Class Constructors

---

- Used to create instances of this class
- Like methods, constructors can be overloaded and are distinguished from one another by their signatures
- Call `Class.getConstructors` to get an array of `Constructor` objects.
- You can use the methods provided by the `Constructor` class to determine the constructor's name, set of modifiers, parameter types, and set of throwable exceptions.
- You can also create a new instance of the `Constructor` object's class with the `Constructor.newInstance` method.

# Constructor

```
class SampleConstructor {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle();  
        showConstructors(r); }  
        static void showConstructors(Object o) {  
            Class c = o.getClass();  
            Constructor[] theConstructors = c.getConstructors();  
            for (int i = 0; i < theConstructors.length; i++) {  
                System.out.print("(" );  
                Class[] parameterTypes = theConstructors[i].getParameterTypes();  
                for (int k = 0; k < parameterTypes.length; k++) {  
                    String parameterString = parameterTypes[k].getName();  
                    System.out.print(parameterString + " ");  
                }  
                System.out.println(")");  
            }  
        }  
        ( )  
        ( int int )  
        ( int int int )  
        ( java.awt.Dimension )  
        ( java.awt.Point )  
        ( java.awt.Point java.awt.Dimension )  
        ( java.awt.Rectangle )
```

# Arrays I

---

- To determine whether an object `obj` is an array,
  - Get its class `c` with `Class c = obj.getClass();`
  - Test with `c.isArray()`
- To find the type of components of the array,
  - `c.getComponentType()`
    - Returns `null` if `c` is not the class of an array
- There is an `Array` class in `java.lang.reflect` that provides *static* methods for working with arrays

# Code

---

```
class SampleComponent {  
    public static void main(String[] args) {  
        int[] ints = new int[2];  
        Button[] buttons = new Button[6];  
        String[][] twoDim = new String[4][5];  
        printComponentType(ints);  
        printComponentType(buttons);  
        printComponentType(twoDim);  
    }  
    static void printComponentType(Object array) {  
        Class arrayClass = array.getClass();  
        String arrayName = arrayClass.getName();  
        Class componentClass = arrayClass.getComponentType();  
        String componentName = componentClass.getName();  
        System.out.println("Array: " + arrayName + ", Component: " +  
            componentName);  
    }  
}  
  
Array: [I, Component: int  
Array: [Ljava.awt.Button;, Component: java.awt.Button  
Array: [[Ljava.lang.String;, Component: [Ljava.lang.String;
```

# Arrays II

---

- To create an array,
- `Array.newInstance(Class componentType, int size)`
  - This returns, as an `Object`, the newly created array
    - You can cast it to the desired type if you like
  - The `componentType` may itself be an array
    - This would create a multiply-dimensioned array
    - The limit on the number of dimensions is usually 255
- `Array.newInstance(Class componentType, int[] size)`
  - This returns, as an `Object`, the newly created multidimensional array (with `size.length` dimensions)

# Arrays III

---

- To get the value of array elements,
  - `Array.get(Object array, int index)` returns an `Object`
  - `Array.getBoolean(Object array, int index)` returns a `boolean`
  - `Array.getByte(Object array, int index)` returns a `byte`
  - etc.
- To store values into an array,
  - `Array.set(Object array, int index, Object value)`
  - `Array.setBoolean(Object array, int index, boolean z)`
  - `Array.setByte(Object array, int index, byte b)`
  - etc.

# Unit Summary

---

- reflection in Java
- Need of reflection
- Java.lang.reflect.\* package
- Implementing and testing reflection.
- Using reflection classes and methods

*Welcome to*  
**Exception handling**

# Unit Objective

---

- Topics:
  - Introduction
  - Errors and Error handling
  - Exceptions
  - Types of Exceptions
  - Coding Exceptions
  - Use of throw and throws
  - User defined exceptions
  - Summary

# Introduction

---

- *open a file*
- *read a line from the file*
- But here's what you might *have* to do:
  - *open a file*
  - *if the file doesn't exist, inform the user*
  - *if you don't have permission to use the file, inform the user*
  - *if the file isn't a text file, inform the user*
  - *read a line from the file*
  - *if you couldn't read a line, inform the user*
  - *etc., etc.*

# Introduction

---

- Users will use our programs in unexpected ways.
- Due to design errors or coding errors, our programs may fail in unexpected ways during execution

# Introduction

---

- It is our responsibility to produce quality code that does not fail unexpectedly.
- Consequently, we must design error handling into our programs.

# Errors and Error Handling

---

- An Error is any unexpected result obtained from a program during execution.
- Unhandled errors may manifest themselves as incorrect results or behavior, or as abnormal program termination.
- Errors should be handled by the programmer, to prevent them from reaching the user.

# Errors and Error Handling

---

- Some typical causes of errors:
  - Memory errors (i.e. memory incorrectly allocated, memory leaks, “null pointer”)
  - File system errors (i.e. disk is full, disk has been removed)
  - Network errors (i.e. network is down, URL does not exist)
  - Calculation errors (i.e. divide by 0)

# Errors and Error Handling

---

- More typical causes of errors:
  - Array errors (i.e. accessing element –1)
  - Conversion errors (i.e. convert ‘q’ to a number)
  - Can you think of some others?

# Errors and Error Handling

---

- Traditional Error Handling
  - 1. Every method returns a value (flag) indicating either success, failure, or some error condition. The calling method checks the return flag and takes appropriate action.
  - Downside: programmer must remember to always check the return value and take appropriate action. This requires much code and something may get overlooked.

# Overcoming the problem

---

Open the file

```
if(CanOpenFile){  
    if(CanReadFile){  
        if(IsTextFile){  
            if(IsNotEmpty){  
                read a line}  
            else{  
                send error -1}  
            }else{  
                send error -2}  
            }else{  
                send error -3}  
        }else{  
            send error -4}
```

# Errors and Error Handling

---

- Exceptions – a better error handling to separate “normal” code from error handling
  - Exceptions are a mechanism that provides the best of both worlds.
  - Exceptions act similar to method return flags in that any method may raise an exception should it encounter an error.
  - Exceptions act like global error methods in that the exception mechanism is built into Java; exceptions are handled at many levels in a program, locally and/or globally.

# A Little Demo

```
public class Test {  
    public static void main(String[] args) {  
        int i = Integer.parseInt(args[0]);  
        int j = Integer.parseInt(args[1]);  
        System.out.println(i/j);  
    }  
}
```

```
> javac Test.java  
> java Test 6 3  
2  
>
```

# Exceptions

```
> java Test
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
at Test.main(Test.java:3)
> java Test 6 0
Exception in thread "main"
java.lang.ArithmetricException: / by zero
at Test.main(Test.java:4)
```

In early languages, that's all that happened: error message, core dump, terminate.

Modern languages like Java support *exception handling*.

# An Exception Is An Object

---

- The names of exceptions are class names, like **NullPointerException**
- Exceptions are objects of those classes
- In the previous examples, the Java language system automatically creates an object of an exception class and *throws* it
- If the program does not *catch* it, it terminates with an error message

# Throwable Classes

---

- To be thrown as an exception, an object must be of a class that inherits from the predefined class **Throwable**
- There are four important predefined classes in that part of the class hierarchy:
  - **Throwable**
  - **Error**
  - **Exception**
  - **RuntimeException**

# Exceptions

---

- What are they?
  - An exception is a representation of an error condition or a situation that is not the expected result of a method.
  - Exceptions are built into the Java language and are available to all program code.
  - Exceptions isolate the code that deals with the error condition from regular program logic.

# Exceptions

---

- How are they used?
  - Exceptions fall into two categories:
    - Checked Exceptions
    - Unchecked Exceptions
  - Checked exceptions are inherited from the core Java class `Exception`. They represent exceptions that are frequently considered “non fatal” to program execution
  - Checked exceptions must be handled in your code, or passed to parent classes for handling.

# Exceptions

---

- How are they used?
  - Unchecked exceptions represent error conditions that are considered “fatal” to program execution.
  - You do not have to do anything with an unchecked exception. Your program will terminate with an appropriate error message.

# Exceptions

---

- Examples:
  - Checked exceptions include errors such as “array index out of bounds”, “file not found” and “number format conversion”.
  - Unchecked exceptions include errors such as “null pointer”.

# Exceptions

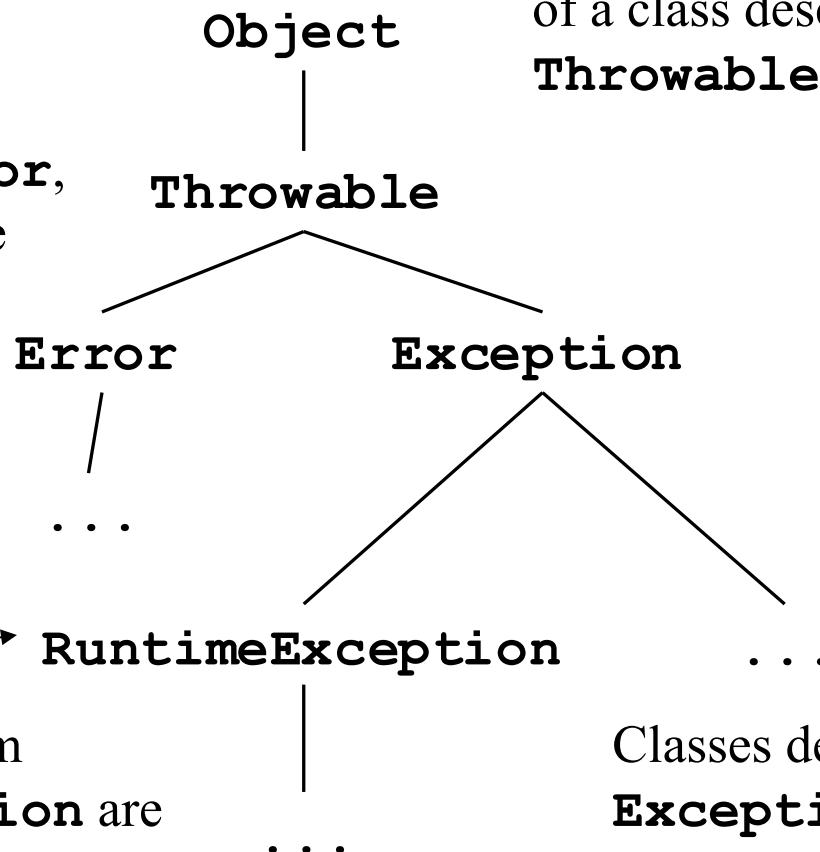
---

- How do you handle exceptions?
  - Exception handling is accomplished through the “try – catch” mechanism, or by a “throws” clause in the method declaration.
  - For any code that throws a checked exception, you can decide to handle the exception yourself, or pass the exception “up the chain” (to a parent class).

Classes derived from

**Error** are used for serious, system-generated errors, like **OutOfMemoryError**, that usually cannot be recovered from

Need not be caught



Classes derived from

**RuntimeException** are used for ordinary system-generated errors, like **ArithmaticException**

Java will only throw objects of a class descended from **Throwable**

Classes derived from

**Exception** are used for ordinary errors that a program might want to catch and recover from

# Exceptions

---

- How do you handle exceptions?
  - To handle the exception, you write a “try-catch” block.  
To pass the exception “up the chain”, you declare a throws clause in your method or class declaration.
  - If the method contains code that may cause a checked exception, you MUST handle the exception OR pass the exception to the parent class (remember, every class has Object as the ultimate parent)

# Coding Exceptions

---

- Try-Catch Mechanism
  - Wherever your code may trigger an exception, the normal code logic is placed inside a block of code starting with the “try” keyword:
  - After the try block, the code to handle the exception should it arise is placed in a block of code starting with the “catch” keyword.

# Coding Exceptions

---

- Example

```
-try {  
    ... do the “normal” code, ignoring possible exceptions  
}  
  
catch(Exception e) {  
    ... exception handling code  
}
```

# Example

```
public class Test {  
    public static void main(String[] args) {  
        try {  
            int i = Integer.parseInt(args[0]);  
            int j = Integer.parseInt(args[1]);  
            System.out.println(i/j);  
        }  
        catch (ArithmetcException a) {  
            System.out.println("You're dividing by zero!");  
        }  
    }  
}
```

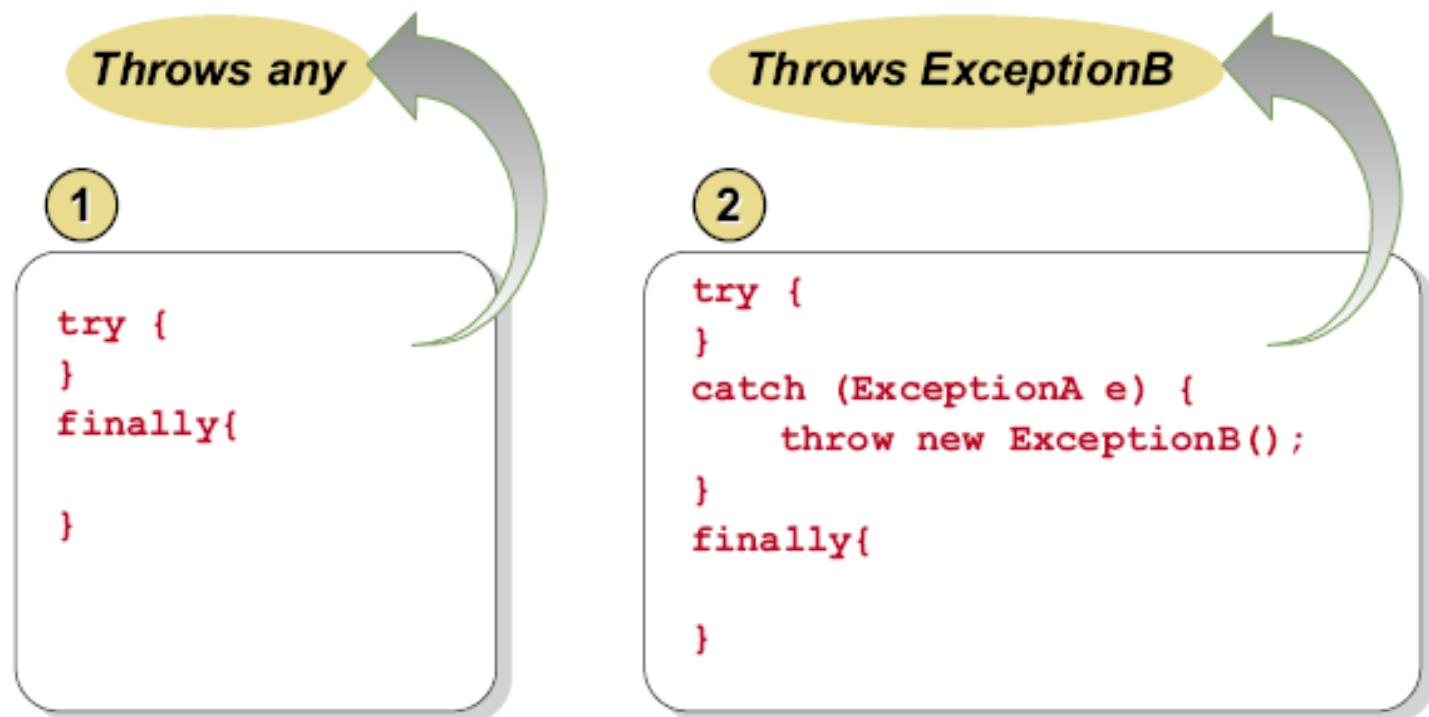
This will catch and handle any **ArithmetcException**. Other exceptions will still get the language system's default behavior.

# Coding Exceptions

---

- Try-Catch Mechanism
  - You may also write an optional “finally” block. This block contains code that is ALWAYS executed, either after the “try” block code, or after the “catch” block code.
  - Finally blocks can be used for operations that must happen no matter what (i.e. cleanup operations such as closing a file)

## *finally* Block



# Coding Exceptions

---

- Passing the exception
  - In any method that might throw an exception, you may declare the method as “throws” that exception, and thus avoid handling the exception yourself
  - Example
    - ```
public void myMethod throws IOException {  
    ... normal code with some I/O  
}
```

# Coding Exceptions

---

- Types of Exceptions
  - All checked exceptions have class “Exception” as the parent class.
  - You can use the actual exception class or the parent class when referring to an exception

# Coding Exceptions

---

- Types of Exceptions

- Examples:

- public void myMethod throws Exception {
    - public void myMethod throws IOException {
    - try { ... }
    - catch (Exception e) { ... }
    - try { ... }
    - catch (IOException ioe) { ... }

# Summary

---

- Exceptions are a powerful error handling mechanism.
- Exceptions in Java are built into the language.
- Exceptions can be handled by the programmer (try-catch),  
or handled by the Java environment (throws).

# Throw From Called Method

---

- The `try` statement gets a chance to catch exceptions thrown while the `try` part runs
- That includes exceptions thrown by methods called from the `try` part

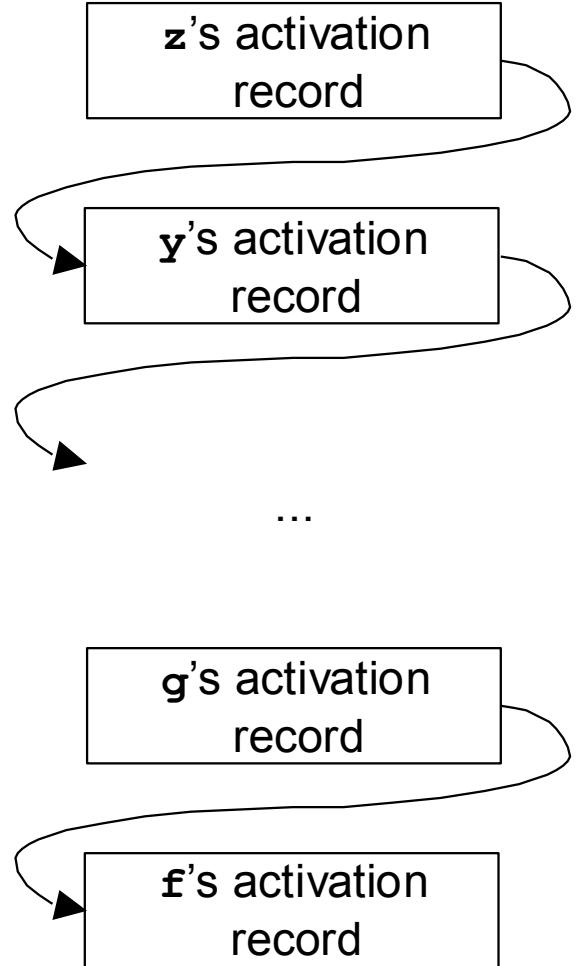
# Example

---

```
void f() {  
    try {  
        g();  
    }  
    catch (ArithmeticException a) {  
        ...  
    }  
}
```

- If **g** throws an **ArithmeticException**, that it does not catch, **f** will get it
- In general, the throw and the catch can be separated by any number of method invocations

- If **z** throws an exception it does not catch, **z**'s activation stops...
- ...then **y** gets a chance to catch it; if it doesn't, **y**'s activation stops...
- ...and so on all the way back to **f**



# Long-Distance Throws

---

- That kind of long-distance throw is one of the big advantages of exception handling
- All intermediate activations between the throw and the catch are stopped and popped
- If not throwing or catching, they need not know anything about it

# Multiple catch Parts

---

```
<try-statement> ::= <try-part> <catch-parts>
<try-part> ::= try <compound-statement>
<catch-parts> ::= <catch-part> <catch-parts>
                  | <catch-part>
<catch-part> ::= catch (<type> <variable-name>)
                  <compound-statement>
```

- To catch more than one kind of exception, a **catch** part can specify some general superclass like **RuntimeException**
- But usually, to handle different kinds of exceptions differently, you use multiple **catch** parts

# Example

---

```
public static void main(String[] args) {  
    try {  
        int i = Integer.parseInt(args[0]);  
        int j = Integer.parseInt(args[1]);  
        System.out.println(i/j);  
    }  
    catch (ArithmetcException a) {  
        System.out.println("You're dividing by zero!");  
    }  
    catch (ArrayIndexOutOfBoundsException a) {  
        System.out.println("Requires two parameters.");  
    }  
}
```

This will catch and handle both **ArithmetcException** and **ArrayIndexOutOfBoundsException**

# Example

---

```
public static void main(String[] args) {  
    try {  
        int i = Integer.parseInt(args[0]);  
        int j = Integer.parseInt(args[1]);  
        System.out.println(i/j);  
    }  
    catch (ArithmaticException a) {  
        System.out.println("You're dividing by zero!");  
    }  
    catch (ArrayIndexOutOfBoundsException a) {  
        System.out.println("Requires two parameters.");  
    }  
    catch (Runtimeexception a) {  
        System.out.println("Runtime exception.");  
    }  
}
```

# Overlapping Catch Parts

---

- If an exception from the `try` part matches more than one of the `catch` parts, only the first matching `catch` part is executed
- A common pattern: `catch` parts for specific cases first, and a more general one at the end
- Note that Java does not allow unreachable `catch` parts, or unreachable code in general

# The `throw` Statement

---

*<throw-statement>* ::= **throw** *<expression>* ;

- Most exceptions are thrown automatically by the language system
- Sometimes you want to throw your own
- The *<expression>* is a reference to a throwable object—usually, a new one:

```
throw new NullPointerException();
```

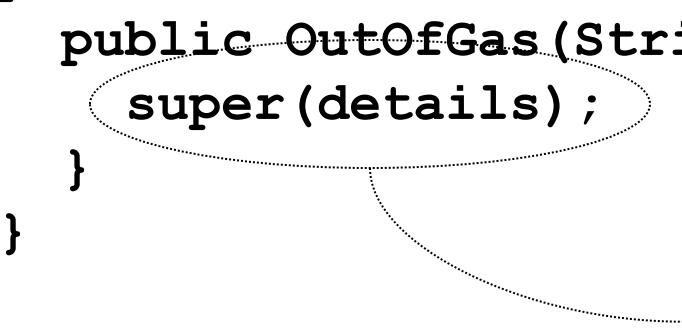
# Using The Exception Object

---

- The exception that was thrown is available in the catch block—as that parameter
- It can be used to communicate information from the thrower to the catcher
- All classes derived from **Throwable** inherit a method **printStackTrace**
- They also inherit a **String** field with a detailed error message, and a **getMessage** method to access it

# Example

```
public class OutOfGas extends Exception {  
    public OutOfGas(String details) {  
        super(details);  
    }  
}
```



This calls a base-class constructor to initialize the field returned by **getMessage()**.

```
try {  
    throw new OutOfGas("You have run out of gas.");  
}  
catch (OutOfGas e) {  
    System.out.println(e.getMessage());  
}
```

# About `super` In Constructors

---

- The first statement in a constructor can be a call to `super` (with parameters, if needed)
- That calls a base class constructor
- Used to initialize inherited fields
- All constructors (except in `Object`) start with a call to another constructor—if you don’t include one, Java calls `super()` implicitly

# More About Constructors

---

- Also, all classes have at least one constructor—if you don’t include one, Java provides a no-arg constructor implicitly

```
public class OutOfGas extends Exception {  
}
```

```
public class OutOfGas extends Exception {  
    public OutOfGas() {  
        super();  
    }  
}
```

These are equivalent!

```
public class OutOfGas extends Exception {  
    private int miles;  
    public OutOfGas(String details, int m) {  
        super(details);  
        miles = m;  
    }  
    public int getMiles() {  
        return miles;  
    }  
}  
  
try {  
    throw new OutOfGas("You have run out of gas.",19);  
}  
catch (OutOfGas e) {  
    System.out.println(e.getMessage());  
    System.out.println("Odometer: " + e.getMiles());  
}
```

# Using printStackTrace( )

```
catch(StackException x)
{
    x.printStackTrace(System.out);
}
```

...

```
StackException: overflow
    at FixedStack.push(FixedStack.java:18)
    at StackTest.doTest(StackTest.java, Compiled Code)
    at StackTest.main(StackTest.java:6)
```

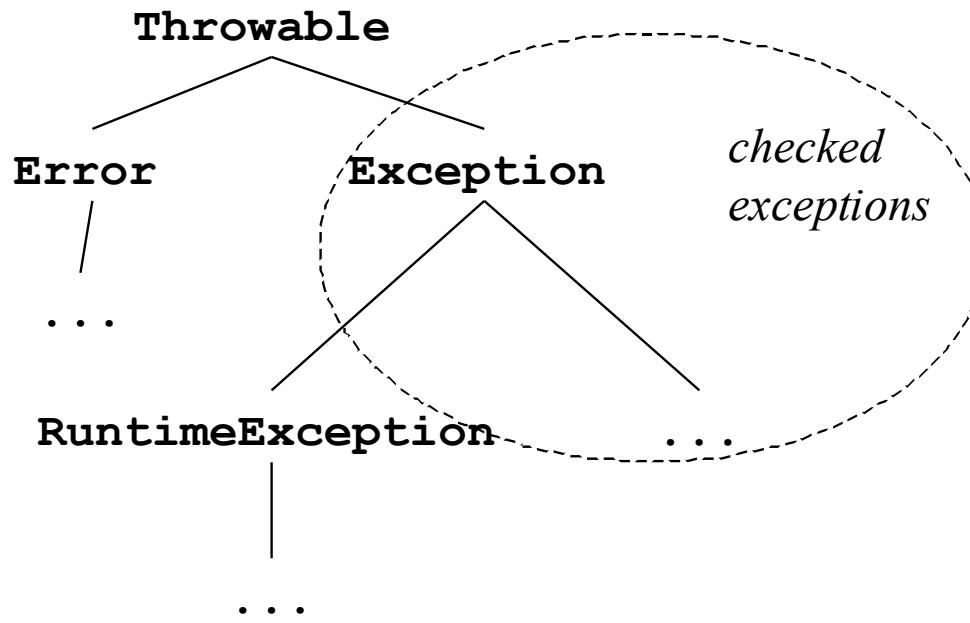
# Checked Exceptions

---

```
void z() {  
    throw new OutOfGas("You have run out of gas.", 19);  
}
```

- This method will not compile: “The exception **OutOfGas** is not handled”
- Java has not complained about this in our previous examples—why now?
- Java distinguishes between two kinds of exceptions: checked and unchecked

# Checked Exceptions



The checked exception classes are **Exception** and its descendants, excluding **RuntimeException** and its descendants

# What Gets Checked?

---

- A method that can get a checked exception is not permitted to ignore it
- It can catch it
  - That is, the code that generates the exception can be inside a **try** statement with a **catch** part for that checked exception
- Or, it can declare that it does *not* catch it
  - Using a **throws** clause

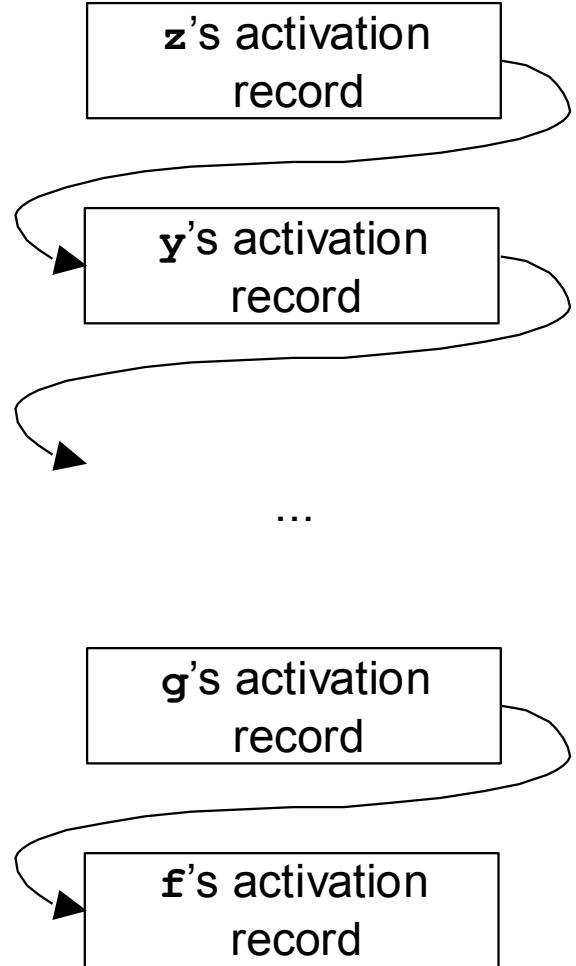
# The Throws Clause

---

```
void z() throws OutOfGas {  
    throw new OutOfGas("You have run out of gas.", 19);  
}
```

- A **throws** clause lists one or more throwable classes separated by commas
- This one always throws, but in general, the **throws** clause means *might* throw
- So any caller of **z** must catch **OutOfGas**, or place it in its own **throws** clause

- If **z** declares that it **throws** `OutOfGas`...
  - ...then **y** must catch it, or declare it **throws** it too...
  - ...and so on all the way back to **f**



# Why Use Checked Exceptions

---

- The **throws** clause is like documentation: it tells the reader that this exception can result from a call of this method
- But it is *verified* documentation; if any checked exception can result from a method call, the compiler will insist it be declared
- This can make programs easier to read and more likely to be correct

# How To Avoid Checked Exceptions

---

- You can always define your own exceptions using a different base class, such as `Error` or `Throwable`
- Then they will be unchecked
- Weigh the advantages carefully

# **java.lang.RuntimeException Subclasses (*sample*)**

---

- ArithmeticException (e.g., divide by 0)
- ArrayStoreException
- ClassCastException
- IllegalArgumentException
- IndexOutOfBoundsException
- NullPointerException
- UnsupportedOperationException

# Principle

---

- “Use checked exceptions for recoverable conditions and run-time exceptions for programming errors” (Bloch, *Effective Java*)

# Exceptions and Inheritance

---

- Methods overridden in subclasses must maintain the parent method's contract
  - substitutability
  - cannot add exceptions to specification
  - can omit, however
  - can throw subclasses of parent's exceptions

## Relaxing the Exception Specification

```
class Parent
{
    public void f() throws Exception
    { }
}

class Child extends Parent
{
    public void f()          // OK!
    { }
}

class Override
{
    public static void main(String[] args)
    {
        Child c = new Child();
        c.f();
    }
}
```

## Throwing a Subclass Exception

```
class MyException extends Exception {}  
class AnotherException extends MyException {}  
  
class Parent {  
    public void f() throws MyException  
    {}  
}  
  
class Child extends Parent {  
    public void f() throws AnotherException  
    {}  
}  
  
class Override {  
    public static void main(String[] args)  
        throws AnotherException  
    {  
        Child c = new Child();  
        c.f();  
    }  
}
```

# Exception-handling Syntax

---

```
try
{
    // normal code (conditionally executes)
}
catch (ExceptionType1 x)
{
    // handle ExceptionType1 error
}
...
catch (ExceptionTypeN x)
{
    // handle ExceptionTypeN error
}
finally
{
    // invariant code ("always" executes)
}
```

# The finally Clause

---

- For code that must ALWAYS run
  - No matter what!
  - Even if a `return` or `break` occurs first
  - Exception: `System.exit( )`
- Placed after handlers (if they exist)
  - try-block must either have a handler or a finally-block

```
class FinallyTest
{
    public static void f()
        throws Exception
    {
        try
        {
            // return;                                // 0
            // System.exit(0);                        // 2
            // throw new Exception();                // 3a
        }
        catch (Exception x)
        {
            // throw new Exception();                // 3b
        }
        finally
        {
            System.out.println("finally!");
        }

        System.out.println("last statement");
    }
}
```

```
public static void main(String[] args)
{
    try
    {
        f();
    }
    catch(Exception x)
    {
    }
}
```

# Program Output

---

- 0:  
`finally!`  
`last statement`
- 1:  
`finally!`
- 2:  
(no output)
- 3a:  
same as 0:
- 3a + 3b:  
compiler error (last statement not reachable)

# Managing Resources

---

- Other than memory
  - files, connections, etc.
- Need to deallocate, even if exceptions occur
- Use **finally**

```
import java.io.*;
class Manage
{
    public static void f(String fname)
        throws IOException
    {
        FileReader f = null; // must define outside try
        try
        {
            f = new FileReader(fname);
            System.out.println("File opened");
            int c = f.read(); // read a byte
            // ...
        }
        finally
        {
            if (f != null)
            {
                System.out.println("File closed");
                f.close(); // beware lost exception!!!
            }
        }
    }
}
```

```
public static void main(String[] args)
{
    try
    {
        f(args[0]);
    }
    catch (Exception x)
    {
        System.out.println(x);
    }
}
```

# Program Output

---

- If no file name given (`args.length == 0`):

`java.lang.ArrayIndexOutOfBoundsException: 0`

- If file doesn't exist:

`java.io.FileNotFoundException: <file name>`

- If file opened successfully:

`file opened`

`file closed`

# When to Handle Exceptions

---

- Note: **Manage.f( )** didn't catch anything
  - wouldn't know what to do if it did!
- You often let exceptions pass up the call stack
- Or you can *re-throw* in a catch
  - throw x; // in a handler where x was caught
  - or re-throw a new type of exception

# Exception Etiquette

---

- Don't catch what you can't (at least partially) handle
  - re-throw if only partially handled ("catch & release": if you're not going to eat it, throw it back!)
- Don't catch & ignore
  - catch (Exception x){} // disables exceptions!

# How Exceptions Work

---

- When an exception is thrown execution backtracks up the runtime stack (list of active function invocations)
- Each stack frame contains information regarding local handlers, if any
  - Otherwise, execution returns up to the next caller, looking for a suitable catch
- What happens if there isn't a matching catch?

# Uncaught Exceptions

---

- What if there is no handler for an exception?
- The thread dies!
  - exceptions belong to a thread (stack-specific)

# Unit Summary

---

- Exceptions
- Exception versus Errors
  - Run time exceptions
  - Checked exceptions
- Four ways to deal with exceptions
  - Ignore exception
  - Handle the exception
  - Throw the exception to the calling method
  - Handle and rethrow the exception to calling method
- Creating and throwing your own exceptions
- Exceptions and overriding.

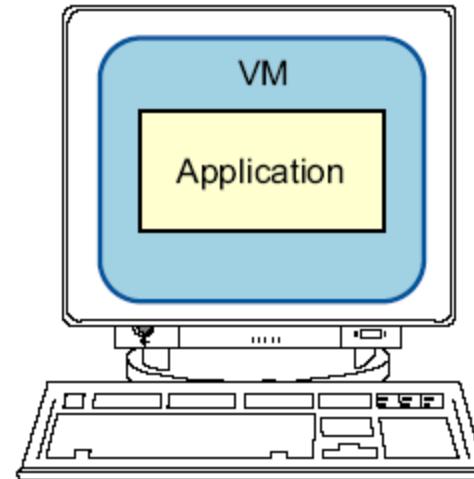
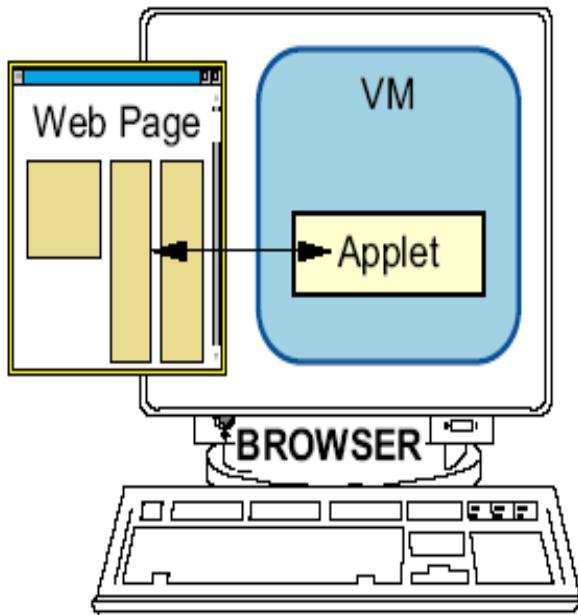
*Welcome to*  
**Applets**

# Unit Objective

---

- After completing this unit you should be able to
  - Compare and contrast applets and applications
  - Describe life cycle of applet and its inherited methods
  - Embed an applet into HTML document
  - Explain applet security restrictions.

# Java Applet versus Java Application



# Applications vs. Applets

---

- Java applications are run like conventional applications, from your hard disk. They run within the Java virtual machine, but they appear to be normal applications on your desktop
- Java applets are run within a web browser. They have limited abilities to read or write to your disk, access the network, etc, for security reasons.

# Applets

---

- An applet is a Panel that allows interaction with a Java program
- A applet is typically embedded in a Web page and can be run from a browser
- You need special HTML in the Web page to tell the browser about the applet
- For security reasons, applets run in a sandbox: they have no access to the client's file system

# Applet Support

---

- Most modern browsers support Java 1.4 if they have the appropriate plugin
- The best support isn't a browser, but the standalone program appletviewer
- In general you should try to write applets that can be run with any browser

# What an applet is

---

- You write an applet by extending the class Applet
- Applet is just a class like any other; you can even use it in applications if you want
- When you write an applet, you are only writing *part* of a program
- The browser supplies the main method

# The genealogy of Applet

---

java.lang.Object

|

+----java.awt.Component

|

+----java.awt.Container

|

+----java.awt.Panel

|

+----java.applet.Applet

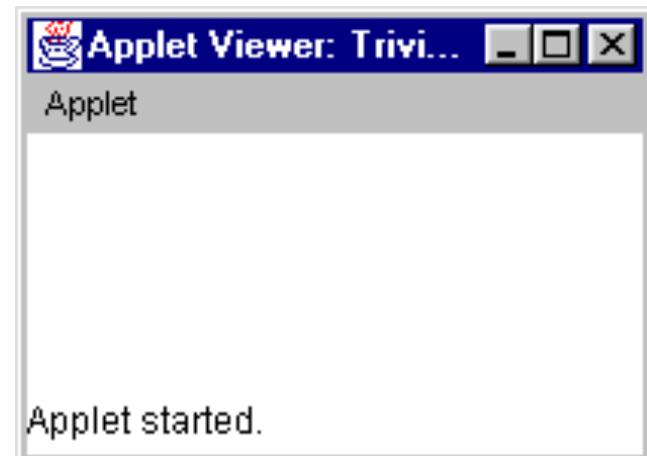
# The simplest possible applet

TrivialApplet.java

```
import java.applet.Applet;  
public class TrivialApplet extends Applet { }
```

TrivialApplet.html

```
<applet  
    code="TrivialApplet.class"  
    width=150 height=100>  
</applet>
```



# The simplest reasonable applet

```
import java.awt.*;
import java.applet.Applet;

public class HelloWorld extends Applet {
    public void paint( Graphics g ) {
        g.drawString( "Hello World!", 30, 30 );
    }
}
```



# Applet methods

---

```
public void init ()  
public void start ()  
public void stop ()  
public void destroy ()  
public void paint (Graphics)
```

Also:

```
public void repaint()  
public void update (Graphics)  
public void showStatus(String)  
public String getParameter(String)
```

# Why an applet works

---

- You write an applet by *extending* the class Applet
- Applet defines methods `init( )`, `start( )`, `stop( )`,  
`paint(Graphics)`, `destroy( )`
- These methods do nothing--they are stubs
- You make the applet do something by overriding these methods

# **public void init ( )**

---

- This is the first method to execute
- It is an ideal place to initialize variables
- It is the best place to define the GUI Components (buttons, text fields, scrollbars, etc.), lay them out, and add listeners to them
- Almost every applet you ever write will have an init( ) method

# **public void start ( )**

---

- Not always needed
- Called after init( )
- Called each time the page is loaded and restarted
- Used mostly in conjunction with stop( )
- start() and stop( ) are used when the Applet is doing time-consuming calculations that you don't want to continue when the page is not in front

# **public void stop( )**

---

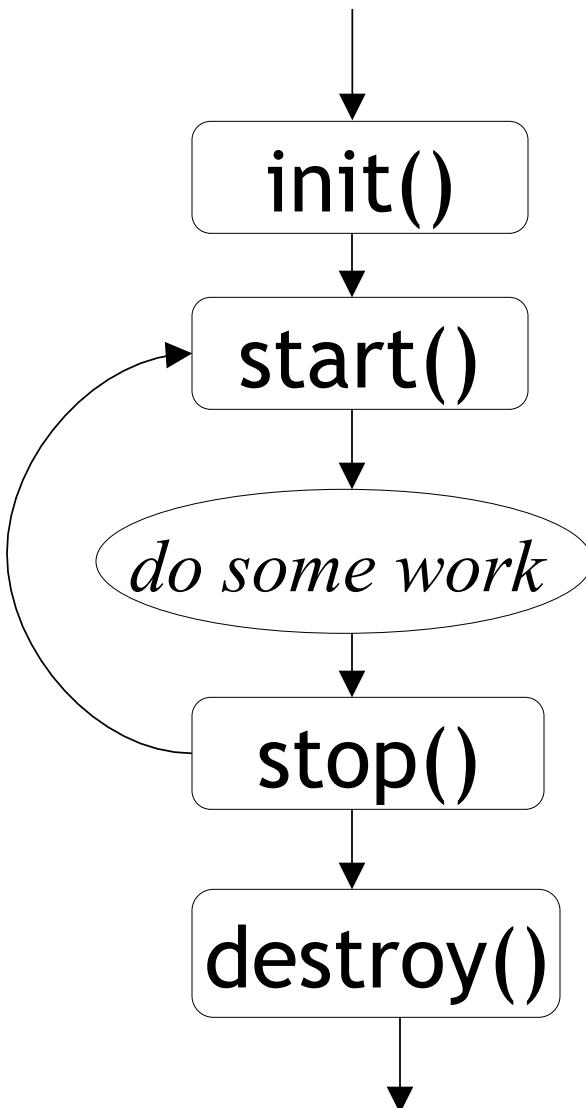
- Not always needed
- Called when the browser leaves the page
- Called just before destroy( )
- Use stop( ) if the applet is doing heavy computation that you don't want to continue when the browser is on some other page
- Used mostly in conjunction with start()

# **public void destroy( )**

---

- Seldom needed
- Called after stop( )
- Use to explicitly release system resources (like threads)
- System resources are usually released automatically

# Methods are called in this order



- `init` and `destroy` are only called once each
- `start` and `stop` are called whenever the browser enters and leaves the page
- `do some work` is code called by your *listeners*
- `paint` is called when the applet needs to be repainted

# **public void paint(Graphics g)**

---

- Needed if you do any drawing or painting other than just using standard GUI Components
- Any painting you want to do should be done here, or in a method you call from here
- Painting that you do in other methods may *or may not* happen
- *Never call paint(Graphics), call repaint( )*

# repaint( )

---

- Call repaint( ) when you have changed something and want your changes to show up on the screen
- repaint( ) is a *request*--it might not happen
- When you call repaint( ), Java schedules a call to update(Graphics g)

# update( )

---

- When you call repaint( ), Java schedules a call to update(Graphics g)
- Here's what update does:

```
public void update(Graphics g) {  
    // Fills applet with background color, then  
    paint(g);  
}
```

# Sample Graphics methods

- A **Graphics** is something you can paint on

`g.drawString("Hello", 20, 20);`

Hello

`g.drawRect(x, y, width, height);`



`g.fillRect(x, y, width, height);`



`g.drawOval(x, y, width, height);`



`g.fillOval(x, y, width, height);`



`g.setColor(Color.red);`



# Painting at the right time is hard

---

- **Rule #1:** Never call `paint(Graphics g)`, call `repaint()`.
- **Rule #2:** Do *all* your painting in `paint`, or in a method that you call from `paint`.
- **Rule #3:** If you paint on any `Graphics` other than the Applet's, call its `update` method from the Applet's `paint` method.
- **Rule #4.** Do your painting in a separate `Thread`.
- These rules aren't perfect, but they should help.

# Other useful Applet methods

---

- `System.out.println(String s)`
  - Works from appletviewer, not from browsers
  - Automatically opens an output window.
- `showStatus(String)`
  - displays the String in the applet's status line.
  - Each call overwrites the previous call.
  - You have to allow time to read the line!

# Other useful Applet methods

## •**getParameter**

public String **getParameter**(String name)

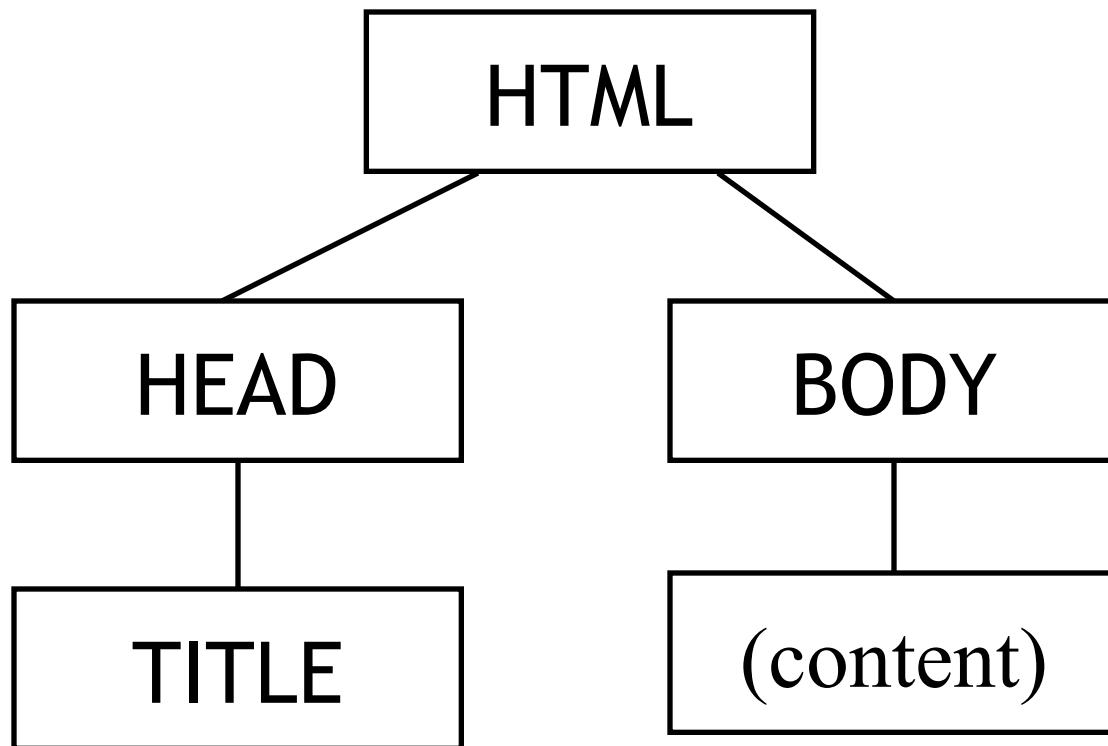
- ◆ Returns the value of the named parameter in the HTML tag. For example, if this applet is specified as <applet code="Clock" width=50 height=50> <param name=Color value="blue"> </applet> then a call to **getParameter("Color")** returns the value "blue".
- ◆ The name argument is case insensitive.

# **Applets are not magic!**

---

- Anything you can do in an applet, you can do in an application.
- You can do some things in an application that you can't do in an applet.
- If you want to access files from an applet, it must be a “trusted” applet.

# Structure of an HTML page



- Most HTML tags are containers.
- A container is `<tag>` to `</tag>`

# HTML

```
<html>
  <head>
    <title> Hi World Applet </title>
  </head>

  <body>
    <applet code="HiWorld.class"
            width=300 height=200>
      <param name="arraysize" value="10">
    </applet>
  </body>
</html>
```

# Example

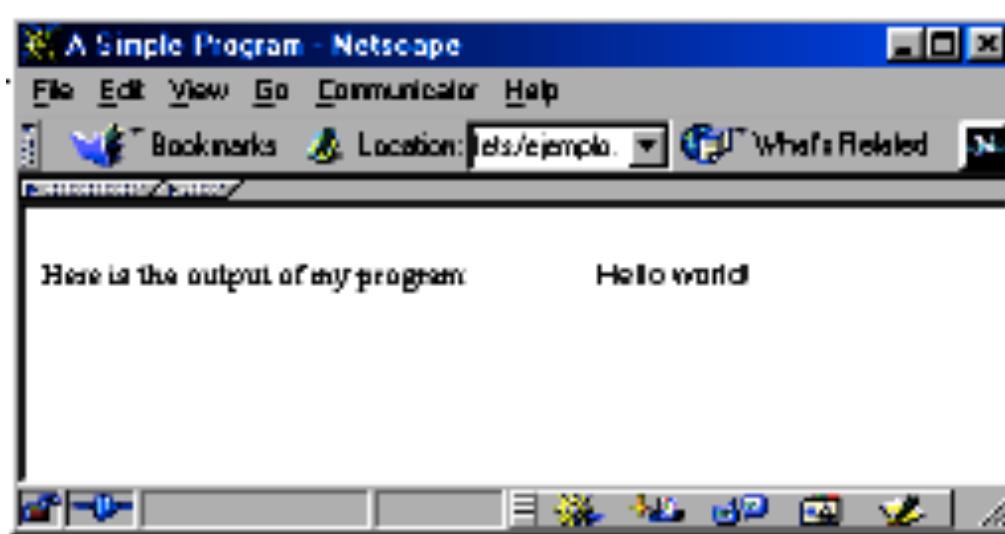
---

*HelloWorld.java*

```
import java.applet.Applet;  
import java.awt.Graphics;  
public class HelloWorld extends Applet {  
    public void paint (Graphics g) {  
        g.drawString ("Hello world!", 50, 25);  
    }  
}
```

# Example.html

```
<HTML>  
<HEAD> <TITLE> A Simple Program </TITLE> </HEAD>  
<BODY>  
Here is the output of my program:  
<APPLET CODE="HelloWorld" WIDTH=150 HEIGHT=25>  
</APPLET>  
</BODY>  
</HTML>
```



# Html Applet Element

---

```
<applet code = “.....” width = xxx height = xxx .....>  
..  
</applet>
```

Required Attributes:-

Code: -

Designates the filename of java class file to load

Filename interpreted with current directory of the current html page unless CODEBASE is supplied.

Width and Height: -

Specifies area the applet will occupy

# Html Applet Element

---

- Other attributes
- Align, Hspace, vspace
  - Controls position and border spacing
- Archive
  - Designates jar file containing all classes and images used by applets
  - Save considerable time when downloading multiple files.

---

```
<param name="arraysize" value="10">
```

- public String getParameter(String name)
- String s = getParameter("arraysize");
- try { size = Integer.parseInt (s) }  
catch (NumberFormatException e) {...}

# Displaying Images

---

- The code base, returned by the **Applet getCodeBase()** method, is a URL that specifies the directory from which the applet's classes were loaded.
- The document base, returned by the **Applet getDocumentBase()** method, specifies the directory of the HTML page that contains the applet.

# Example on Displaying Images

---

- Images must be in GIF or JPEG format.
- Example:
- Image file `yawn.gif` is in directory “images”.
- To create an image object “`nekopicture`” that contains “`yawn.gif`”:

```
Image nekopicture = new Image nekopicture()  
getImage(getCodeBase(), "images/yawn.gif");
```

# Playing Sounds

---

- The AudioClip interface in the **Applet** class provides basic support for playing sounds.
- Sound format: 8 bit, 8000 Hz, one-channel, Sun ".au" files.
- Methods in AudioClip that need to be implemented in the applet:  
loop(): starts playing the clip repeatedly      play() & stop() to play & stop the clip.

# Applets and Security

---

- Java applets have a strict security model for the following reasons:
  - Applets are loaded and run without the knowledge or intervention by the consumer.
  - Applets are likely to be loaded across fire-wall boundaries.
- Without run-time restrictions, client applets would have all of the access privileges of the underlying process.
- Embedded Java Virtual Machines give applets different access rights depending if they were loaded across the network or from the local file system.

# Network Loaded Applets

---

Network loaded applets are prevented from:

- Running any executable code on the local machine.
- Reading and writing disk information from the local machine.
- Establishing a network connection with a host other than from which they were downloaded.
- Loading native code libraries.

*Welcome to*  
**Designing Web Page**

# **Objectives:**

---

- Understand how web pages are put together
- Learn basic HTML tags and develop your own web pages
- Learn how images and Java applets are added to web pages

# HTML —

---

- HyperText

Text contains “hot links.” When touched or clicked, a link takes you to the specified place

- Markup

Formatting commands are embedded in the text as “tags” (e.g., `<b> ... </b>` makes it bold)

- Language

A very limited “language,” just a few dozen tags

# HTML from Source to Display

```
<html>  
...  
<body>  
<h1 align="center">HTML</h1>  
  
<p> <i>WYSIWYG</i> stands for  
What You See Is What You Get...  
  
</body>  
  
</html>
```

Display (“what you get”):



# **HTML Source is Device- and Platform-Independent.**

---

- But:
  - may be displayed differently according to the capabilities of a particular device (computer screen, printer), its size, resolution, colors, etc.
  - may be interpreted differently by different software (*Netscape Navigator, Internet Explorer, etc.*)

# HTML Features

---

- Tags for formatting and positioning text
- Tags for lists, tables, embedded pictures, and Java applets
- Tags for hyperlinks and “anchors”

# HTML Syntax

---

- Each tag is enclosed in angular brackets:

<sometag>

- HTML is case-blind: doesn't distinguish between upper case and lower case.

- Many tags require a matching closing tag:

<sometag>my text</sometag>

# HTML Syntax (cont'd)

- Some tags may take attributes:

```
<p align="center">
```

```
<sometag  
attr="somevalue">
```

- Certain characters (<, >, &, ©, etc.) are represented by an “escape sequence.”

|   |   |        |
|---|---|--------|
| < | → | &lt;   |
| > | → | &gt;   |
| & | → | &amp;  |
| © | → | &copy; |

# HTML Syntax (cont'd)

---

- Tags can be nested:

```
<font color="blue"><i>Red Sea</i></font>
```

Or:

```
<strong>Click
```

```
<a href="details.html">here</a> for
```

```
details.</strong>
```

# HTML Document Structure

```
<html>
  <head>
    <title>...<title>
    <meta name="author" content="...">
    <meta name="keywords" content="...">
    ...
  </head>
  ...
  <body>
    ...
    <address>
      ...
    </address>
  </body>
</html>
```

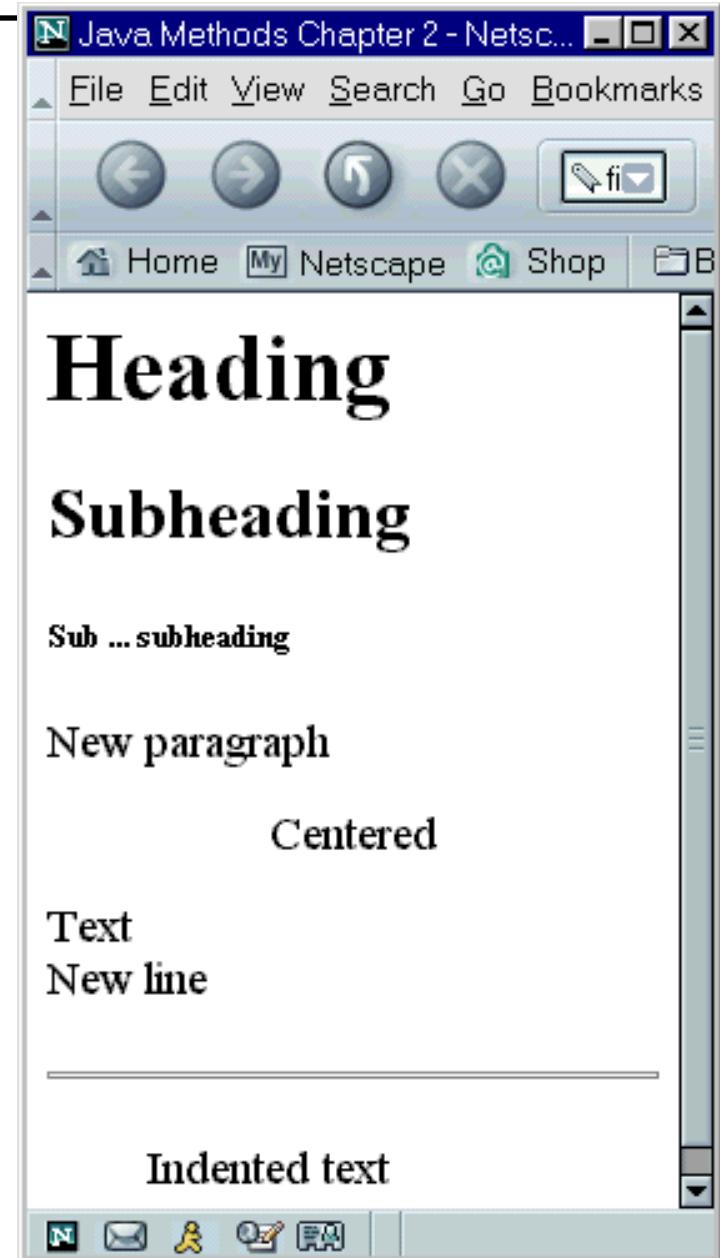
Info about the document

Contact info for webmaster, etc.

Info to be displayed

# Text Layout

```
<h1>Heading</h1>
<h2>Subheading</h2>
<h6>Sub ... subheading</h6>
<p>New paragraph
<p align="center">Centered
<p>Text<br>New line
<hr width="95%">
<blockquote>
Indented text
</blockquote>
```



# Text Formatting

```
<p>Regular <b>Bold</b> <i>Italic</i>  
<u>Underlined</u>
```

```
<p><strong>Emphasis</strong>  
<cite>Citation</cite>
```

```
<p><code>Typewriter font</code>  
<p><big>Big</big> Regular  
<small>Small</small> <sub>below</sub>  
<sup>above</sup>
```

```
<p><font color="red" size="+2">Big & red</font>
```



# Anchors and Hyperlinks

---

- An “anchor” defines a location in the current HTML document.
- An anchor uses an `<a>` tag with a `name` attribute:

```
<a name="panda">  
<h3>Giant Pandas</h3>
```

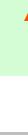
- An anchor can be placed anywhere in a doc

An anchor  
named  
“panda”

# Anchors and Hyperlinks (cont'd)

- A hyperlink defines “hot text” and the destination (a URL) to go to when the link is clicked.
- A hyperlink uses the `<a>` tag with an `href` attribute and a closing `</a>` tag:

Only 1630 `<a href="#panda">`Giant pandas`</a>` are left in the world.



When clicked, takes you to the anchor named “panda”

# URLs

---

- URL stands for “Uniform (or Universal) Resource Locator.”
- A hyperlink can link to any URL.
- A URL can point to an HTML file, a pdf file, an image, audio, or video file, and even to an e-mail address.
- A URL can be absolute or relative.

# Absolute URLs

---

- An absolute URL defines the absolute location of a resource on the Internet.
- Examples:

`http://www.myzoo.com/reptiles.html`



Protocol      Host computer  
                  (web server)      File name

`mailto:fanmail@britneyspears.com`



Protocol      e-mail address

# Relative URLs

---

- A relative URL in a link describes a location relative to the location of the document that holds that link.
- Examples:

#panda

volleyball.html

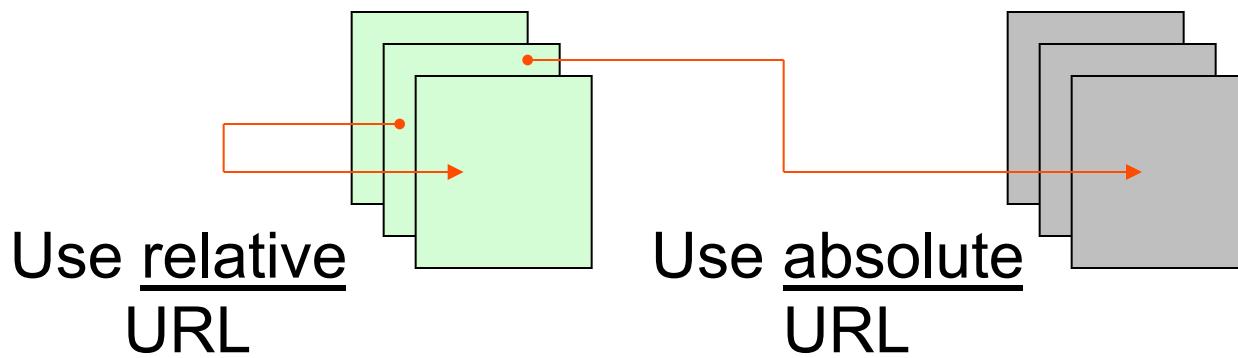
athletics.html#swimteam

images/lucie.jpg

../courses/webdesign

# URLs in Hyperlinks

- Use relative URLs to link to resources on your own web site.
- Use absolute URLs to link to resources on other web sites.



# Lists

```
<ul>  
<li>Bike</li>  
<li>Car</li>  
<li>Unicycle</li>  
</ul>
```

```
<ol>  
<li>Bike</li>  
<li>Car</li>  
<li>Unicycle</li>  
</ol>
```

```
<dl>  
<dt>Car</dt>  
<dd>4 wheels</dd>  
<dt>Bike</dt>  
<dd>2 wheels</dd>  
<dt>Unicycle</dt>  
<dd>1 wheel</dd> </  
dl>
```

- Car
- Bike
- Unicycle

1. Car
2. Bike
3. Unicycle

Car  
4 wheels  
Bike  
2 wheels  
Unicycle  
1 wheel

# Images

---

- .gif files

GIF, Graphics Interchange Format

- .jpg files

JPEG, Joint Photographic Experts Group

# 

---

- Other attributes (optional):

alt="*some text*"

align="top/center/bottom"

border="*thickness*" (0 — no border)

usemap="#*mapname*"

# Images and Hyperlinks

---

- To turn an image into a hot link, surround it with `<a href=...>` and `</a>` tags.
- To turn different sections of an image into different hot links, define a “map”:

```
<map name="mapname">  
  <area shape="circle/rect/square" coords="..."  
        href="URL">  
  <area ...>  
  ...  
</map>
```

# Tables

---

- Tables can be used to:
  - Display data
  - Place text and an image side by side
  - Make narrow columns of text
  - Box text messages by adding border
  - Add color background to text boxes
- Tables can be nested.

# <table ...> Tag

---

- Optional attributes:

`border="thickness"` (0 — no border)

`width="n% (or number of pixels)"`

`cellpadding="number of pixels"`

(additional space between  
data and cell border)

`cellspacing="number of pixels"`

(additional space between  
cells)

# <table ...> Tag (cont'd)

```
<table>
```

```
  <th>
```

```
    <td> ... </td>  <td> ... </td> ... <td> ... </td>
```

```
  </th>
```

Header row  
(optional)

```
<tr>
```

```
    <td> ... </td>  <td> ... </td> ... <td> ... </td>
```

```
</tr>
```

Regular  
row

```
...
```

```
<tr>
```

Individual cell

```
...
```

```
</tr>
```

```
<caption> ... </caption>
```

```
</table>
```

Optional caption

# The <applet> Tag

---

- The tag adds a Java applet to the web page.
- An applet's code consists of .class files and may also include images, audioclips, etc.
- Only one class, the “main” class, is listed in the <applet> tag.

# The <applet> Tag (cont'd)

```
<applet code="ClassName" width=... height=...
        alt="some text" codebase="URL">
```



Optional attributes

Your browser is ignoring the &lt;applet&gt; tag

```
<param name="..." value="...">
<param ...>
</applet>
```



Optional parameters  
for the applet

# Unit Summary

---

- applets and applications
- life cycle of applet and its inherited methods
- Embedding applets
- applet security restrictions.
- Designing web page

*Welcome to*  
**Streams**

# Unit Objective

---

- After completing this unit you should be able to
  - Define a stream
  - Differentiate between byte stream and character stream
  - Recognize abstraction of byte stream through `InputStream` and `OutputStream` classes
  - Recognize abstraction of characterstream through `Reader` and `Writer` classes
  - Nest streams using wrapper classes
  - Perform File I/O
  - Perform serialization and deserialization
  - Use transient key word.

# Input/Output

---

- It's not very interesting to always type in information or provide static information. We want some way to send data to the application, and files are a classic way to do that.

# I/O in Java

---

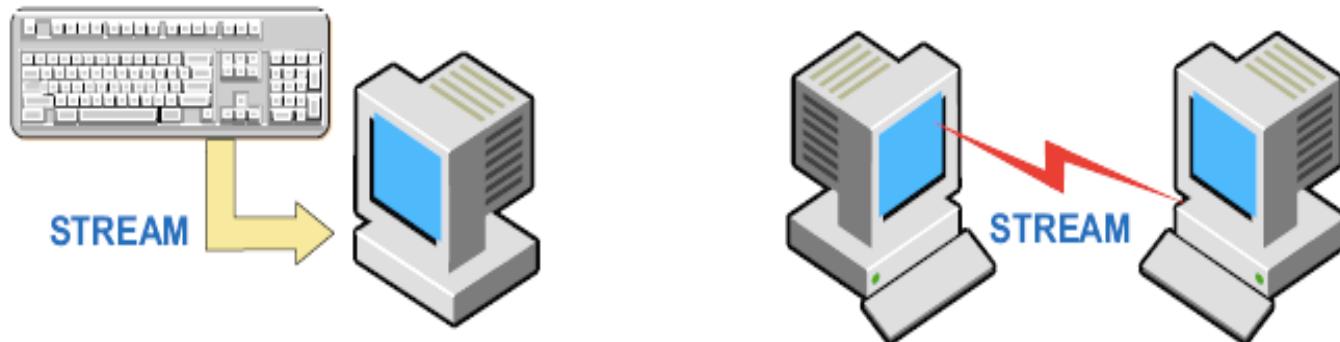
- Many packages and libraries associated with Java provide sophisticated ways to input and output information, e.g.:
  - **GUI-based approaches:** *input* from low-level keyboard and mouse events, or higher level semantic operations on buttons, menus, etc; *output* through arbitrarily sophisticated graphics (Swing, AWT, Java2d, etc).

# I/O in Java

---

- **(Server-side) Web-based approaches:** *input* from HTML forms (or via client-side Javascript); *output* through dynamically generated Web pages (Java Servlets, Java Server Pages, etc).
- Nevertheless, we often need to resort to more traditional methods to read and write information from terminal keyboard and screen, or ordinary files.

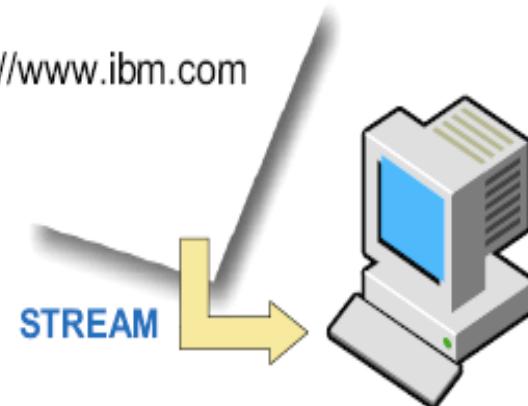
# What is a *Stream*?



A stream is a path of information  
from a source to a destination.



<http://www.ibm.com>



# Streams

---

- All modern I/O is stream-based
- A *stream* is a connection to a source of data or to a destination for data (sometimes both)
- A keyboard may serve as an input stream
- A file may serve as an input stream
- Files have a definite length, and therefore an end; keyboard input goes on indefinitely

# How to do I/O

---

- `import java.io.*;`
- Open the stream
- Use the stream (read, write, or both)
- Close the stream

# Streams

---

Byte Streams

InputStream

OutputStream

Character Streams

Reader

Writer

# Streams

---

- 2 kinds
  - **character streams**
    - Read and write 16-bit Unicode characters
    - Reader and Writer
    - Should be used for reading/writing text information
  - **byte streams** Read and write 8-bit (= 1 byte) characters
    - InputStream and OutputStream
    - Should be used for reading/writing images or sound files
    - Used for Object serialization

# I/O Streams

---

- The `java.io` package contains many classes that allow us to define various streams with particular characteristics
- Some classes assume that the data consists of characters
- Others assume that the data consists of raw bytes of binary information
- Streams can be further subdivided as follows:
  - *data stream*, which acts as either a source or destination
  - *processing stream*, which alters or manipulates the basic data in the stream

# Character vs. Byte Streams

---

- A *character stream* manages 16-bit Unicode characters
- A *byte stream* manages 8-bit bytes of raw binary data
  - A program must determine how to interpret and use the bytes in a byte stream
  - Typically they are used to read and write sounds and images

# Character vs. Byte Streams

---

- The InputStream and OutputStream classes (and their descendants) represent byte streams
- The Reader and Writer classes (and their descendants) represent character streams

# Data vs. Processing Streams

---

- A *data stream* represents a particular source or destination such as a string in memory or a file on disk
- A *processing stream* (also called a *filtering stream*) manipulates the data in the stream
  - It may convert the data from one format to another
  - It may buffer the stream

# The IOException Class

---

- Operations performed by the I/O classes may throw an **IOException**
  - A file intended for reading or writing might not exist
  - Even if the file exists, a program may not be able to find it
  - The file might not contain the kind of data we expect
- An IOException is a checked exception

# Standard I/O

---

- There are three standard I/O streams:
  - *standard input* – defined by System.in
  - *standard output* – defined by System.out
  - *standard error* – defined by System.err
- System.in typically represents keyboard input
- System.out and System.err typically represent a particular window on the monitor screen
- We use System.out when we execute println statements

# Standard I/O

---

- PrintStream objects automatically have print and println methods defined for them
- The PrintWriter class is needed for advanced internationalization and error checking

# OutputStream

---

```
abstract class OutputStream {  
    public abstract void write(int b) // writes a byte (0..255)  
    throws IOException; // (byte is signed)  
    // (char is 16bit)  
    public void write(byte[] data)  
    throws IOException;  
    public void write(byte[] data, int offset, int length)  
    throws IOException;  
    public void flush() // in this class empty  
    throws IOException;  
    public void close()  
    throws IOException  
}
```

# InputStream

---

```
abstract class InputStream {  
    public abstract int read() // blocking -1=EOS  
    throws IOException;  
    public int read(byte[] input) // blocking -1=EOS  
    throws IOException;  
    public int read(byte[] input, int offset, int length)  
    throws IOException; // blocking -1=EOS  
    public long skip(long n)  
    throws IOException;  
    public int available() // >= 0 [default impl: 0]  
    throws IOException;  
    public void close()  
    throws IOException  
    public void mark(int readAheadLimit);  
    public void reset() throws IOException;  
    public boolean markSupported(); // default impl: false  
}
```

# Converting a Byte Stream into a Character Stream

- ***InputStreamReader*** converts a byte stream into a character stream



```
InputStreamReader isr = new InputStreamReader(System.in);
```

- ***OutputStreamWriter*** converts a character stream into a byte stream



```
OutputStreamWriter osw = new OutputStreamWriter(System.out);
```

# Wrapper Streams

Wrapping streams allows you to create more efficient programs.

BufferedReader

InputStreamReader

InputStream/System.in

# BufferedInputStream

---

- Buffer data while reading or writing, thereby reducing the number of accesses required on the original data source.  
Buffered streams are typically more efficient than similar nonbuffered streams.
- Because buffer is available, skipping, marking, resetting of streams becomes possible.
  - `BufferedInputStream(InputStream inputStream)`
  - `BufferedInputStream(InputStream inputStream, int bufsize)`

# Wrapping a Stream

---

- Combines the various features of the many streams.

```
BufferedReader in = new BufferedReader(source);
```

- The code opens a BufferedReader on source, which is another reader of a different type.
  - “Wraps” source in a BufferedReader.
  - Reads from the BufferedReader, which in turn reads from source.
  - Can use BufferedReader's convenient readLine method.

# Example

---

```
import java.io.*;  
class StreamExample  
{  
    public static void main(String[] args)  
    {  
        // Notice the nesting of streams  
        InputStream is = System.in;  
        InputStreamReader isr = new InputStreamReader(is);  
        BufferedReader br = new BufferedReader(isr);
```

# Example

---

```
try
{
String s;
// For a String you test for null
while((s = br.readLine()) != null)
{
    System.out.println(s);
}
br.close();
}
catch(IOException e)
{
    System.out.println("main(): " + e);
}
}
```

# The kinds of InputStream

**ObjectInputStream**

Input of objects

**SequenceInputStream**

A sequence of streams - eg set of files

**ByteArrayInputStream**

Read from byte array in memory

**PipedInputStream**

Streams between threads

**FileInputStream**

Read bytes from file

**FilterInputStream**

Processes data through stream

**PushbackInputStream**

Can push back

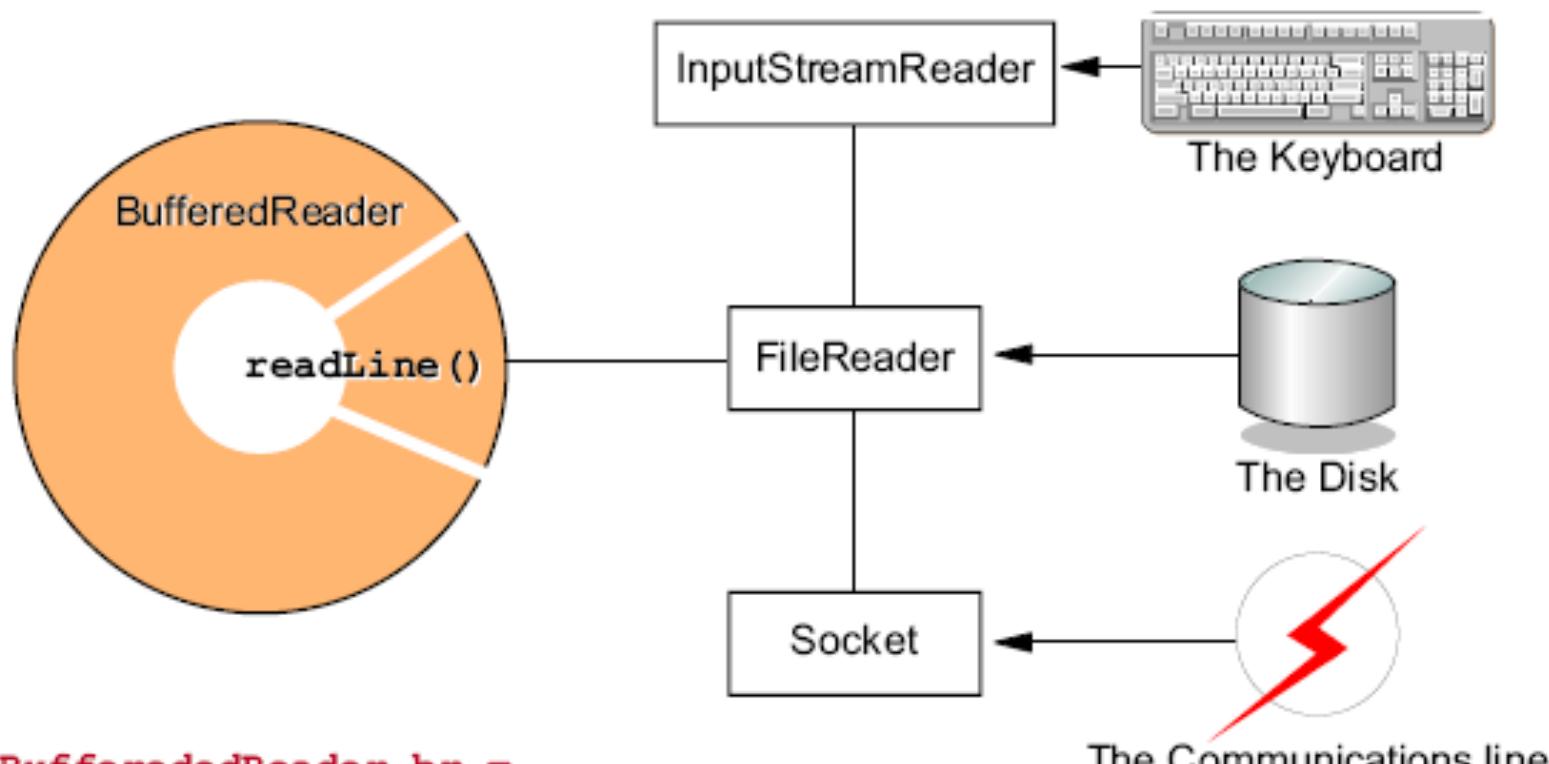
**BufferedInputStream**

Has buffer

**DataInputStream**

Reads primitive types

# I/O Objects in Java



```
BufferedReader br =  
    new BufferedReader(  
        new FileReader(args[0]));  
BufferedReader br =  
    new BufferedReader(  
        new InputStreamReader(System.in));
```

# Console Example

---

```
import java.io.*;  
  
public class BufferedConsole{  
    public static void main(String[ ] args){  
        try{  
            BufferedReader br = new BufferedReader(  
                new InputStreamReader(System.in));  
  
            BufferedWriter bw = new BufferedWriter(  
                new OutputStreamWriter(System.out));  
        }  
    }  
}
```

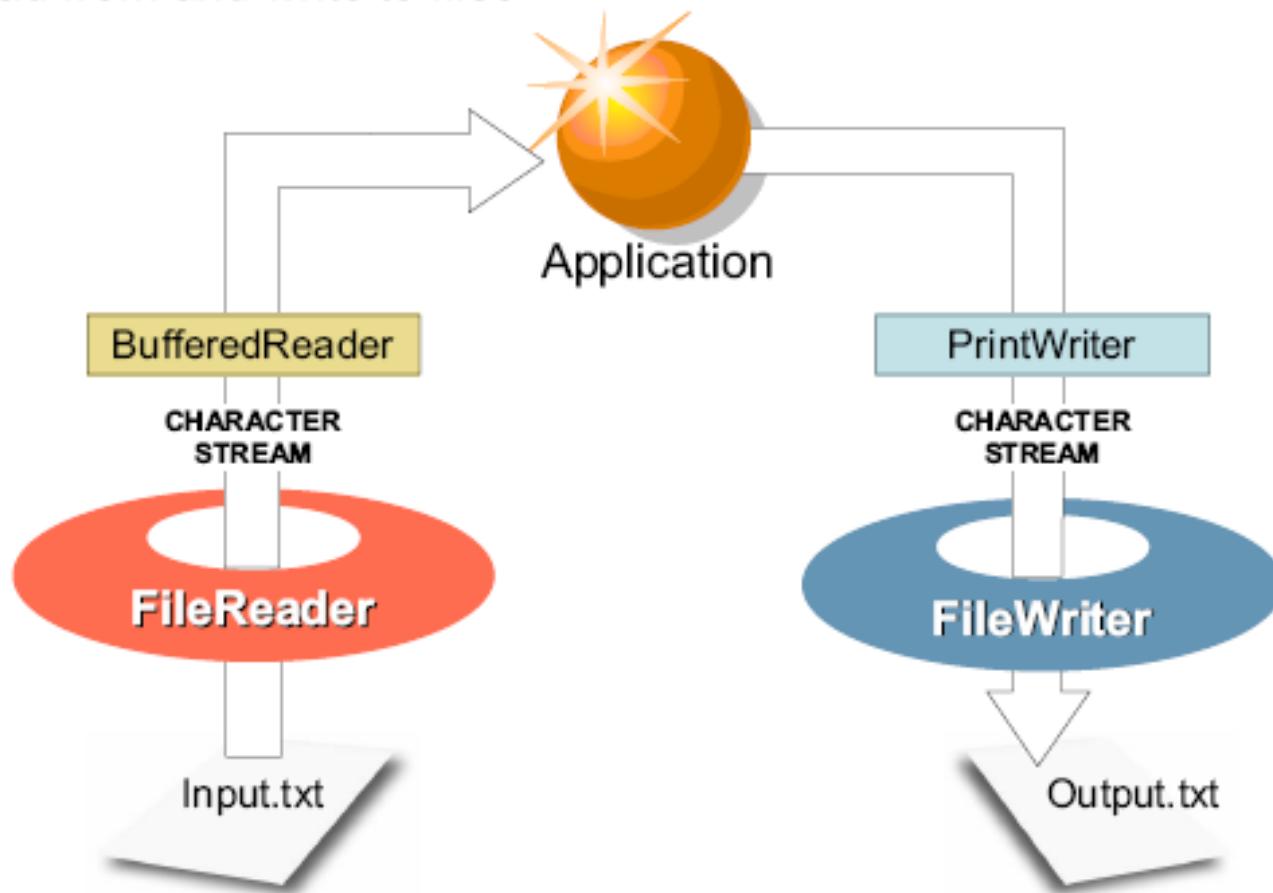
# Console Example

---

```
String s = br.readLine();
while(s != null){
    bw.write("s: " + s);
    s = br.readLine(); // end while loop
}
bw.flush();
bw.close();
br.close();
} // end try block
catch(IOException io){
    System.out.println(io);
}
}
}
```

# File I/O

- Using FileReader and FileWriter you can read from and write to files



# Text File I/O Streams

---

- Input

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

- Output

```
PrintWriter out = new PrintWriter(new BufferedWriter(new  
    FileWriter("foo.out")));
```

- BufferedWriter is not necessary but makes things more efficient
- Output streams should be closed explicitly  
`out.close();`

# File I/O Example

---

```
class StreamExample {  
    public static void main(String[] args) {  
        FileReader fr = null;  
        BufferedReader br = null;  
        FileWriter fw = null;  
        PrintWriter pw = null;  
        try {  
            // fr and fw must be declared in a try/catch  
            // block.  
            fr = new FileReader("input.txt");  
            fw = new FileWriter("output.txt");  
            br = new BufferedReader(fr);  
            pw = new PrintWriter(fw);  
            String s;
```

# File I/O Example

---

```
while((s = br.readLine()) != null)
{
    pw.write(s);
    // Don't forget to flush the output stream.
    pw.flush();
}
}

catch(IOException e1){
    System.out.println("main(): " + e1);
}
finally {
    // Don't forget to close your streams.
    try {
        if (br != null) br.close();
        if (pw != null) pw.close();
    } catch (IOException e2) {}
}
}
```

# FileInputStream/FileOutputStream

- FileInputStream class creates an InputStream that can be used to read bytes from files.
  - `FileInputStream(String filepath)`
  - `FileInputStream(File fileobj)`
- FileOutputStream class creates an OutputStream that can be used to write bytes to a file.
  - `FileOutputStream(String filepath)`
  - `FileOutputStream(File fileobj)`

# Using FileInputStream

---

```
byte data=0;
int bytesInFile=0;
FileInputStream fr = null;
try
{
    fr = new FileInputStream("io1.dat");
    bytesInFile = fr.available();
    for (int i=0; i<bytesInFile; i++)
    {
        data=(byte)fr.read();
        System.out.println("Read "+data);
    }
    fr.close();
}
catch (IOException ioe)
{
    System.out.println("Problem with file");
}
```

# DataInputStream

---

- DataInputStream is a more sophisticated object for getting input. It wraps around a type of input stream, like a FileInputStream.
- It has more complex read operations--readByte, readDouble, etc.

# Data Streams

---

- The data streams provide methods for reading and writing all basic Java types.

# DataInput and DataOutput

- Interface `java.io.DataInput`
  - Provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.
  - Some `DataInput` methods:
    - `boolean readBoolean() throws IOException`
    - `byte readByte() throws IOException`
    - `char readChar() throws IOException`
    - `double readDouble() throws IOException`
    - `float readFloat() throws IOException`
    - `void readFully(byte[]) throws IOException`
    - `int readInt() throws IOException`
    - `String readLine() throws IOException`
    - `long readLong() throws IOException`
    - `short readShort() throws IOException`
    - `String readUTF() throws IOException`
    - `int skipBytes(int n) throws IOException`

# DataInput and DataOutput

---

- Interface `java.io.DataOutput`
  - Provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream.
  - Some `DataOutput` methods:
    - `void write(byte[] b)`
    - `void write(byte[] b, int off, int len)`
    - `void write(int b)`
    - `void writeBoolean(boolean b) throws IOException`
    - `void writeByte(int b) throws IOException`
    - `void writeBytes(String s) throws IOException`
    - `void writeChars(String s) throws IOException`
    - `void writeDouble(double v) throws IOException`
    - `void writeFloat(float v) throws IOException`
    - `void writeInt(int v) throws IOException`
    - `void writeLong(long v) throws IOException`
    - `void writeShort(int v) throws IOException`
    - `void writeUTF(String s) throws IOException`

# Data Input/Output Streams

---

- Input/Output streams that implement the `DataInput`/`DataOutput` interfaces
- `DataInputStream`
  - Lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.
  - Constructor:
    - `DataInputStream(InputStream in)`
    - Creates a `DataInputStream` that uses the specified underlying `InputStream`.

# Data Input/Output Streams

---

- **DataOutputStream**
  - Lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.
  - Constructor:
    - `DataOutputStream(OutputStream out)`
    - Creates a DataOutputStream that uses the specified underlying OutputStream.

# Code

---

- Code generally looks like this:

```
FileInputStream fis = new FileInputStream("MyFile");
DataInputStream dis = new DataInputStream(fis);

dis.readInt(); dis.readByte();....
```

# Chaining of Streams

---

- Like buffered streams, data streams are also “chained” with other I/O streams.
- For example:
- ```
DataInputStream in = new DataInputStream(
    new BufferedInputStream(
        new
    FileInputStream("test.dat")));

```
- The order of chaining is important. In our example, `DataInputStream` should be used as the “outer” one because we need to use the `DataInputStream` methods; `FileInputStream` should be used as the “inner” one because it will open and interact with the file.

# Chaining for Object Streams

---

- Similar to data streams and buffer streams, object streams are usually chained with other streams (like file streams).

# Reading and Updating Data

---

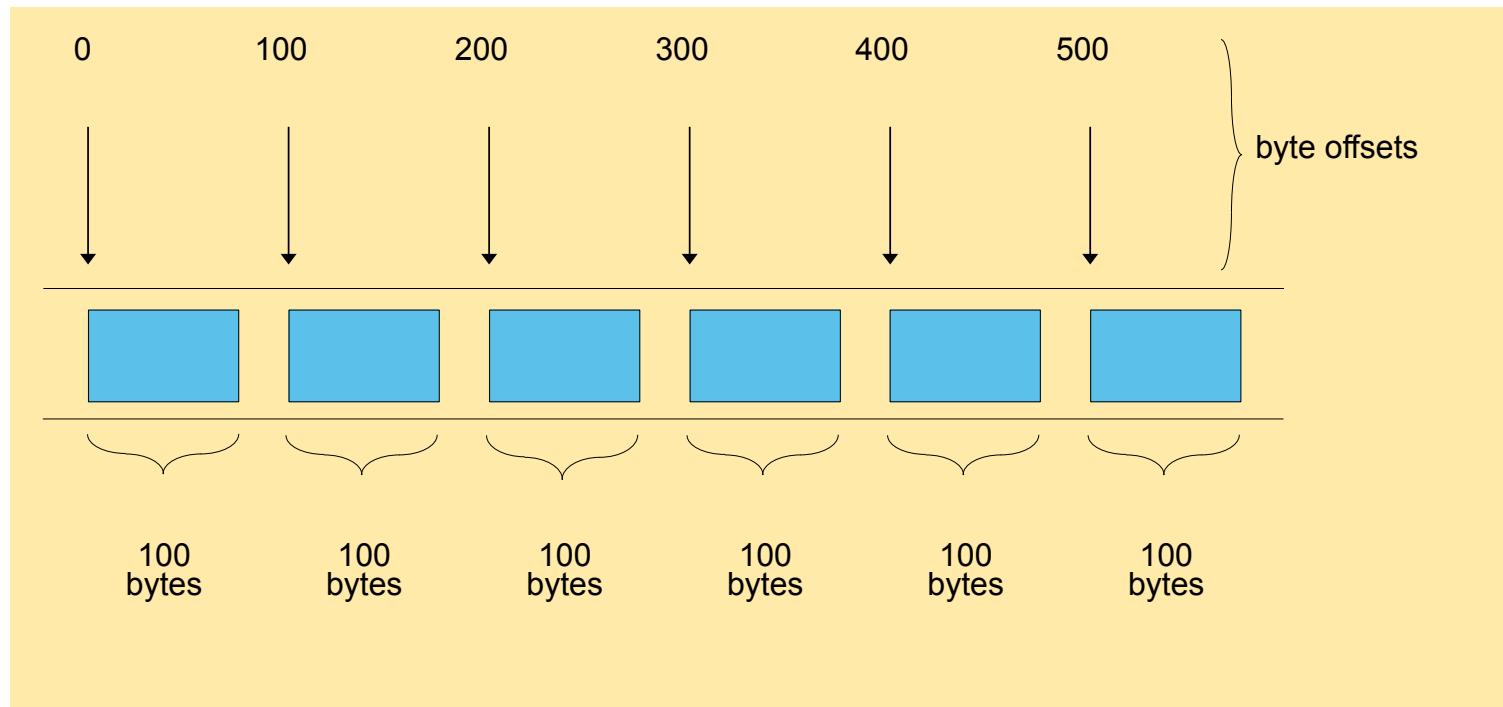
- Accessing a sequential file
  - Data must be read in same format and order it was written
- Updating Sequential-Access Files
  - Difficult to update a sequential-access file
  - Entire file must be rewritten to change one field
  - Only acceptable if many records being updated at once

# Random Access File

---

- “Instant-access” applications
  - Record must be located immediately
  - Transaction-processing systems require rapid access
- Random access files
  - Access individual records directly and quickly
  - Use fixed length for every record
    - Easy to calculate record locations
  - Insert records without destroying other data in file
- Byte offsets can be set anywhere in the file

# Java's view of a random-access file



# RandomAccessFile

---

- RandomAccessFile
  - Like DataInputStream and DataOutputStream because it implements the DataInput and the DataOutput interfaces
  - Does not support ObjectInput and the ObjectOutput
  - Unlike streams, RandomAccessFile takes care of both reading and writing
  - Reads or writes data in spot specified by file-position pointer
    - Manipulates all data as primitive types
    - Normally writes one object at a time to file

# RandomAccessFile

---

- RandomAccessFile constructors:
  - `public RandomAccessFile(File file, String mode) throws FileNotFoundException`
    - Creates a random access file stream to read from, and optionally to write to, the file specified by the `File` argument.
  - `public RandomAccessFile(String filename, String mode) throws FileNotFoundException`
    - Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

# RandomAccessFile

---

- Four different modes for RandomAccessFile
- "r"
  - Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.
- "rw"
  - Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
- "rws"
  - Open for reading and writing, as with "rw", and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.
- "rwd"
  - Open for reading and writing, as with "rw", and also require that every update to the file's content be written synchronously to the underlying storage device.

# RandomAccessFile

---

- RandomAccessFile methods
  - public void close() throws IOException
  - public long length() throws IOException
  - public void setLength(long newLength)  
throws IOException
  - public void seek(long pos)  
throws IOException
- All other methods in the DataInput and DataOutput interfaces

# Writing Data Randomly to a Random-Access File

- `RandomAccessFile` method `seek`
  - Determines location in file where record is stored
  - Sets file-position pointer to a specific point in file

# Reading and Writing Data from a Random-Access File

---

- The seek method can take you to anywhere in the file.
- Need to be careful in not seeking to position outside the current file (beyond the length of the file).
- Also need to make sure that you are reading the correct data type from the current position.
- Seek time can be long. Excessive seeking operations will make your programs run significantly more slowly!

# ByteArrayInputStream

---

- `ByteArrayInputStream` is an implementation of `InputStream` that uses a byte array as the source
  - `ByteArrayInputStream(byte array[])`
  - `ByteArrayInputStream(byte array[],int start, int end)`

# ByteArrayOutputStream

---

ByteArrayOutputStream is an implementation of OutputStream that uses a byte array as the destination.

ByteArrayOutputStream()

ByteArrayOutputStream(int numBytes)

# Filters

---

- Filter data as it's being read from or written to the stream.
- Classes: FilterInputStream, FilterOutputStream,.
- A filter stream is constructed on another stream
- The read method gets input from the underlying stream, filters it, and passes on the filtered data to the caller.
- The write method in a writable filter stream filters the data and then writes it to the underlying stream.
- The filtering done by the streams depends on the stream.
  - Some streams buffer the data,
  - Some count data as it goes by, and
  - Others convert data to another form.

# Filtered Byte Stream

---

- Processing streams perform some sort of operation, such as buffering or character encoding, as they read and write.
- You attach a filtered stream to another stream to filter the data as it's read from or written to the original stream.
  - `FilterOutputStream(OutputStream os)`
  - `FilterInputStream(InputStream is)`

# Using FilterInputStream 1

```
class InputDoubler extends FilterInputStream
{
    InputDoubler(FileInputStream is)
    {
        super(is);
    }

    public int read() throws IOException
    {
        int c = super.read();
        if (c==-1) return c;
        else return (2*c);
    }
}
```

**This class reads bytes from a file and doubles them**

# Using FilterInputStream 2

```
int b=0;
InputDoubler id = null;
try { // try to open file
    id = new InputDoubler(new FileInputStream("io1.dat"));
    while (b!=-1) // until end of file
        { // read and display bytes (doubled)
            try { b=id.read(); System.out.println(b); }
            catch (IOException ioe) { System.out.println(ioe); }
        }
    try { id.close(); } // close file
    catch (IOException ioe) { System.out.println(ioe); }
}
catch (IOException ioe) { System.out.println(ioe); }
```

**Instantiating and using  
the sub-classed filter**

# SequenceInputStream

---

- ❖ Has a constructor which takes a set of InputStreams.
- ❖ Then reading from it is like reading from the InputStreams one after the other
- ❖ Actually two constructors
  - one takes an Enumeration of InputStreams
  - other just takes two

# SequenceInputStream example

---

```
public class Join
{ // concatenates two files on command line
public static void main(String[] args)
{
    FileInputStream file1;
    FileInputStream file2;
    SequenceInputStream sis;
    try {
        file1= new FileInputStream(args[0]);
        file2= new FileInputStream(args[1]);
        sis= new SequenceInputStream(file1, file2);
        int ch=0;
        boolean ended = false;
        while (!ended)
        {
            ch= sis.read();
            if (ch!=-1) System.out.print((char)ch);
            else ended=true;
        }
    }
    catch (IOE .. couldn't open a file..
```

# PrintStream

---

- ◆ Contain convenient printing methods. These are the easiest streams to write to, so you will often see other writable streams wrapped in one of these.
  - PrintStream(OutputStream outputStream)
  - PrintStream(OutputStream outputStream,boolean flushOnNewLine)

Supported Methods:

print()

println()

# FileReader

---

- A FileReader is used to connect to a file that will be used for input
- `FileReader fileReader =  
 new FileReader (fileName);`
- The fileName specifies where the (external) file is to be found

# Example

---

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

# CharArrayReader

---

```
public class CharArrayReaderDemo {  
  
    public static void main(String[] args) throws IOException {  
        String temp = "welcome to this world";  
        int length = temp.length();  
        char c[] = new char[length];  
        temp.getChars(0,length,c,0);  
        CharArrayReader charreader1 = new CharArrayReader(c);  
        CharArrayReader charreader2 = new CharArrayReader(c,0,5);  
    }  
}
```

# CharArrayReader

---

```
int i;  
System.out.println("first array is");  
while((i = charreader1.read())!= -1)  
System.out.println((char)i);  
  
System.out.println("second reader is");  
while((i = charreader2.read())!= -1)  
System.out.println((char)i);  
}  
}
```

# PrintWriter

---

- PrintWriter is a character oriented version of PrintStream.  
`PrintWriter(OutputStream outputstream)`
  - PrintWriter(OutputStream *outputstream*, boolean  
`flushOnNewLine`)
- Buffers are automatically flushed when the program ends normally
- Usually it is your responsibility to flush buffers if you don't think the program will end normally

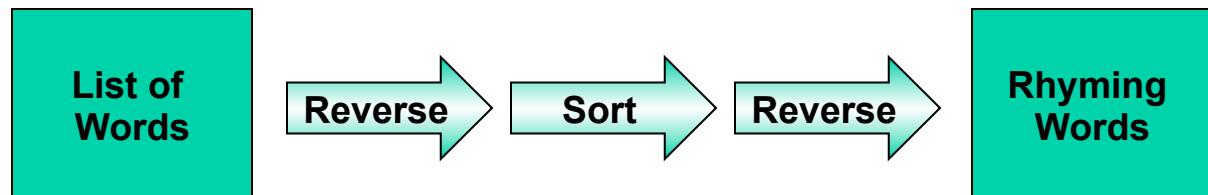
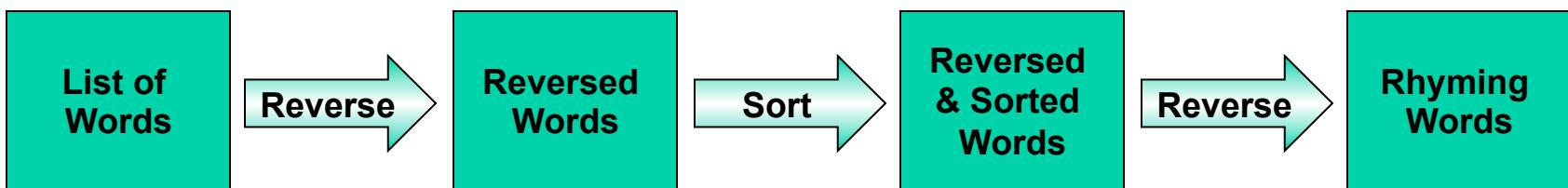
# Pipes

---

- Pipes are used to channel the output from one thread into the input of another.
- Why is this useful?
  - Consider a class that implements various string manipulation utilities, such as sorting and reversing text.
  - It would be nice if the output of one of these methods could be used as the input for another so that you could string a series of method calls together to perform a higher-order function.
    - For example, you could reverse each word in a list, sort the words, and then reverse each word again to create a list of rhyming words.

# Pipes

- Without pipe streams, the program would have to store the results somewhere (such as in a file or in memory) between each step, as shown here:
- With pipe streams, the output from one method could be piped into the next, as shown in this figure:



# Pipes

---

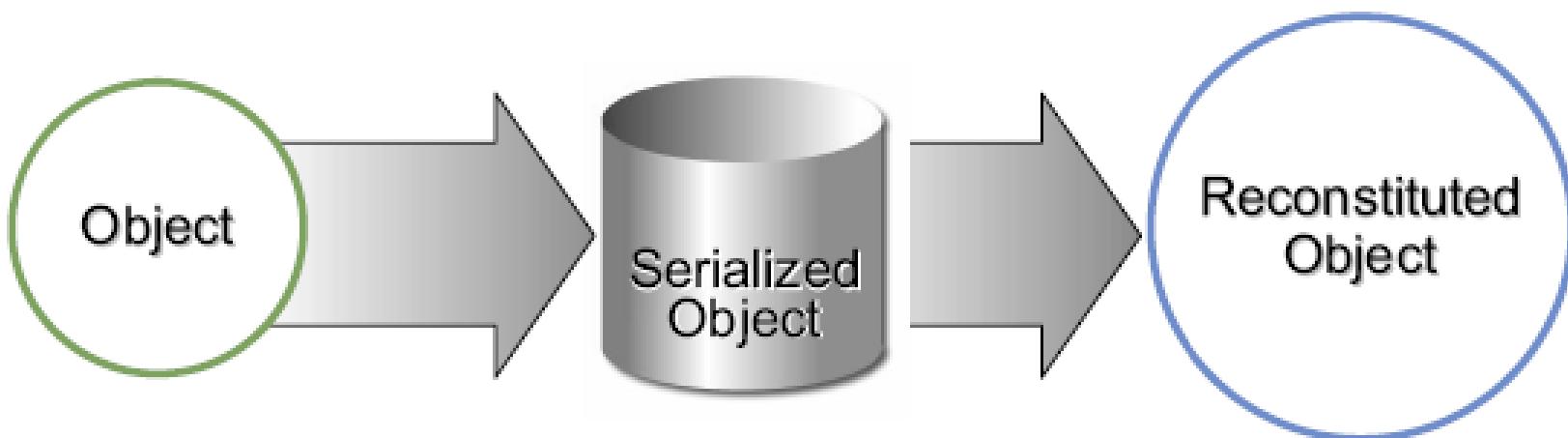
- Both character and byte stream classes for Pipes
  - PipedReader, PipedWriter, PipedInputStream and PipedOutputStream

# Serialization

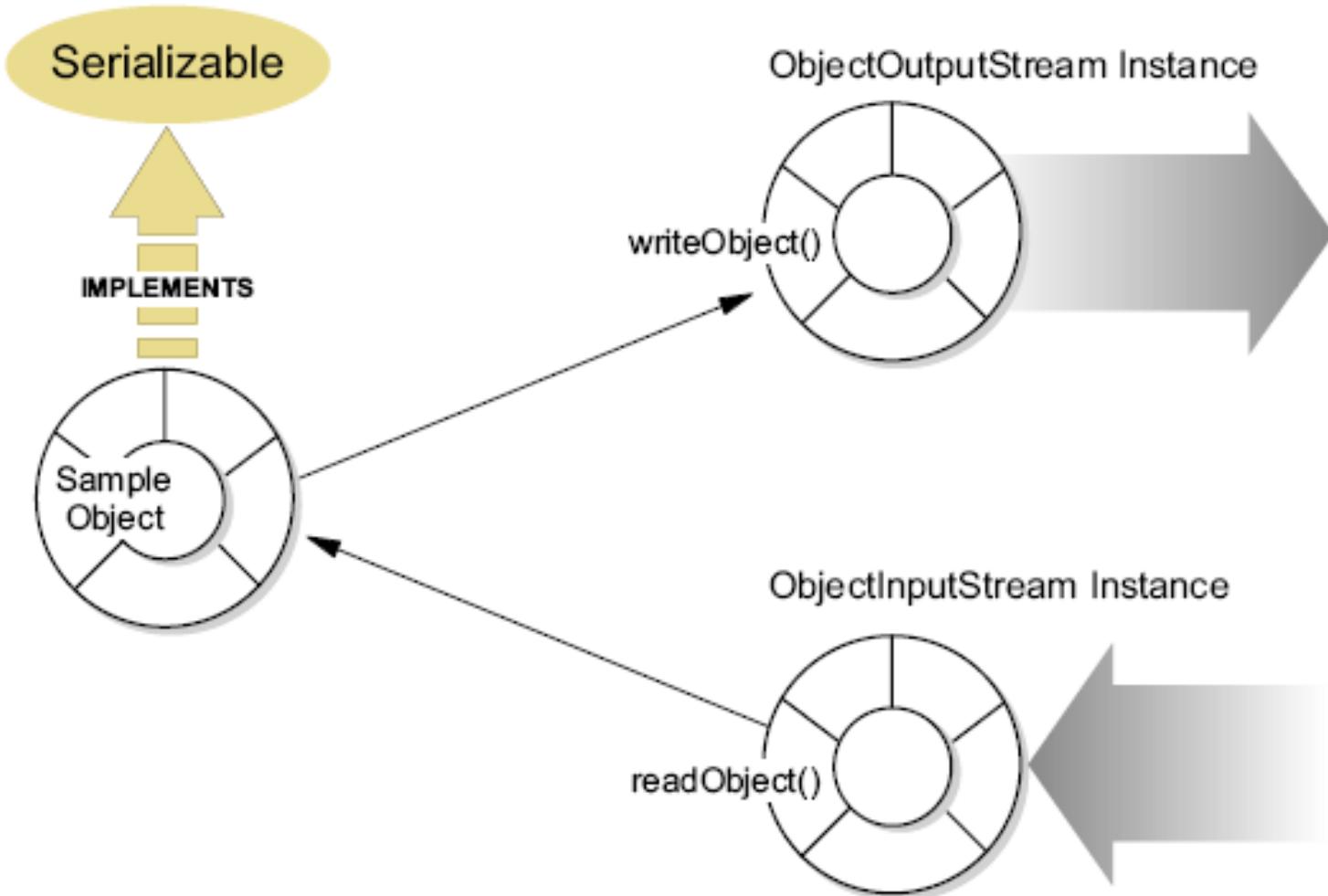
Serialization provides a standard way to save and restore object graphs between sessions on the same or different systems.

Serialization saves the following:

- Object type
- Internal information
- References to other objects



# Marking an Object for Serialization



# Object Input/Output Streams

---

- Data streams deal with primitive types; object streams can deal with both objects and primitive types because they implement both the `DataInput/DataOutput` and `ObjectInput/ObjectOutput` interfaces

# Object Input/Output Streams

---

- An `ObjectInputStream` deserializes (reads) primitive data and objects previously written using an `ObjectOutputStream`.
- Constructor:
  - `ObjectInputStream(InputStream in)`
  - Creates an `ObjectInputStream` that uses the specified underlying `InputStream`.

# Object Input/Output Streams

---

- An `ObjectOutputStream` writes primitive data types and graphs of Java objects to an `OutputStream`. The objects can be read (reconstituted) using an `ObjectInputStream`.
- Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be reconstituted on another host or in another process.
- Only objects that support the `java.io.Serializable` interface can be written to streams.
- Constructor:
  - `ObjectOutputStream(OutputStream out)`
  - Creates an `ObjectOutputStream` that uses the specified underlying `OutputStream`.

# Object Stream Methods

---

- **ObjectInputStream**

- `public final Object readObject() throws IOException, ClassNotFoundException`
  - Read an object from the input stream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are read.

- **ObjectOutputStream**

- `public final void writeObject(Object obj) throws IOException`
  - Write the specified object to the output stream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written.

# Writing the Object to File

---

```
// To write the object, you will need a  
// FileOutputStream and an ObjectOutputStream.  
  
FileOutputStream fos = null;  
  
fos = new FileOutputStream("SerializedObj.ser");  
  
oos = new ObjectOutputStream(fos);  
  
oos.writeObject(originalObj1);  
  
oos.writeObject(originalObj2);  
}catch(Exception e) {  
    System.out.println("Main: main(): " + e);  
}  
  
finally {  
    oos.close();  
}  
}  
}  
}
```

# Writing the Object to File

---

```
import java.io.*;  
class Main {  
    public static void main(String[] args) {  
        ObjectOutputStream oos = null;  
        try {  
            // Get an instance of the SampleObjects and  
            // set their state.  
            SampleObject originalObj1 = new SampleObject();  
            SampleObject originalObj2 = new SampleObject();  
            originalObj1.setName("Mary Smith");  
            originalObj1.setAge(32);  
            originalObj2.setName("John Doe");  
            originalObj2.setAge(42);  
        } catch (IOException e) {  
            e.printStackTrace();  
        } finally {  
            if (oos != null) {  
                try {  
                    oos.close();  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

# The `writeObject` method

---

- Serializes the specified object, traverses its references to other objects recursively, and writes them all.
  - Preserves relationships between objects
- `ObjectOutputStream` implements the `DataOutput` interface
  - Has methods for writing primitive data types, such as `writeInt`, `writeFloat`, or `writeUTF`.
  - You can use these methods to write primitive data types to an `ObjectOutputStream`.
- The `writeObject` method throws a `NotSerializableException` if it's given an object that is not serializable.
  - An object is serializable only if its class implements the `Serializable` interface.

# Reading the Serialized Object from a File

---

```
import java.io.*;
class Main
{
    public static void main(String[] args)
    {
        try
        {
            // To read the objects, you will need a
            // FileInputStream and an ObjectInputStream.

            FileInputStream fis = null;
            ObjectInputStream ois = null;
            fis = new FileInputStream("SerializedObj.ser");
            ois = new ObjectInputStream(fis);
        }
    }
}
```

# Reading the Serialized Object from a File

---

```
SampleObject newObj1 = (SampleObject) ois.readObject();
SampleObject newObj2 = (SampleObject) ois.readObject();
ois.close();
// And the results
System.out.println("SampleObject1 name: " +
newObj1.getName());
System.out.println("SampleObject2 name: " +
newObj2.getName());
}
catch(Exception e)
{ System.out.println("Main: main(): "
+
e);
}
}
```

# How to Read an Object

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

- Deserializes the String Today and the current Date.
- ObjectInputStreams also have to be wrapped on another stream
  - We are reading in from the same file to which we wrote the objects

# Serializable Interface

---

- Known as a marker interface
- Classes that do not implement this interface will not have any of their state serialized or deserialized.
- All subtypes of a serializable class are themselves serializable.
- The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

# Conditions for serializability

---

- If an object is to be serialized:
  - The class must be declared as public
  - The class must implement `Serializable`
  - The class must have a no-argument constructor
  - All fields of the class must be serializable: either primitive types or serializable objects

# Implementing the **Serializable** interface

---

- To "implement" an interface means to define all the methods declared by that interface
- The **Serializable** interface does not define any methods!
- **Serializable** is used only as a kind of flag to tell Java it needs to do extra work with this class

# Object Serialization

---

- *Object serialization* is the mechanism for saving an object, and its current state, so that it can be used again in another program
- The idea that an object can “live” beyond the program execution that created it is called *persistence*
- Object serialization is accomplished using the Serializable interface and the ObjectOutputStream and ObjectInputStream classes
- The writeObject method is used to serialize an object
- The readObject method is used to deserialize an object

# Object Serialization

---

- Serialization takes into account any other objects that are referenced by an object being serialized, saving them too
- Each such object must also implement the Serializable interface
- Many classes from the Java class library implement Serializable, including the String class
- The ArrayList class also implements the Serializable interface, permitting an entire list of objects to be serialized in one operation

# The transient Modifier

---

- When we serialize an object, sometimes we prefer to exclude a particular piece of information such as a password
- The reserved word `transient` modifies the declaration of a variable so that it will not be included in the byte stream when the object is serialized
- For example

```
private transient int password;
```

# Unit Summary

---

- stream
- byte stream and character stream
- abstraction of byte stream through `InputStream` and `OutputStream` classes
- abstraction of characterstream through `Reader` and `Writer` classes
- Nesting of streams
- File I/O
- serialization and deserialization
- transient key word.

**Welcome to  
Multithreading**

# Roadmap

---

- Introduction
- Java thread model
- Creating a thread
- Thread properties & synchronization
- Suspending,Resuming & stopping Threads

# What is a Thread?

---

- First, let's discuss a sequential program
  - Each has a beginning,
  - an execution sequence,
  - and an end.
- At any given time during the runtime of the program, there is a single point of execution.
  - HelloWorld

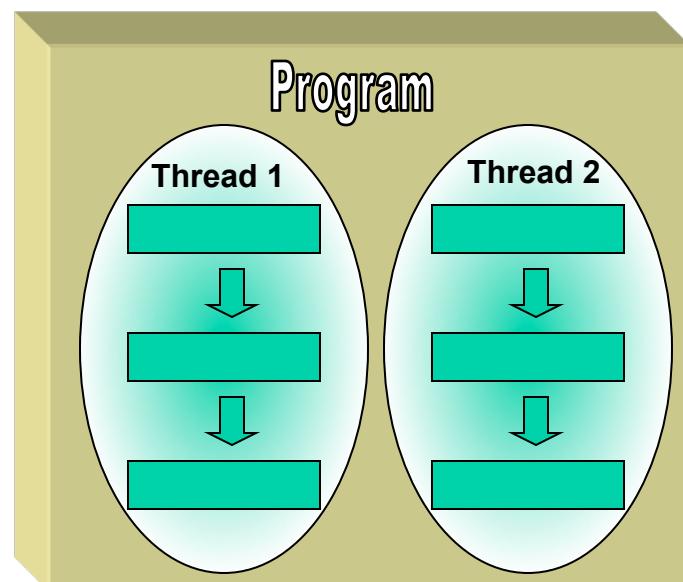
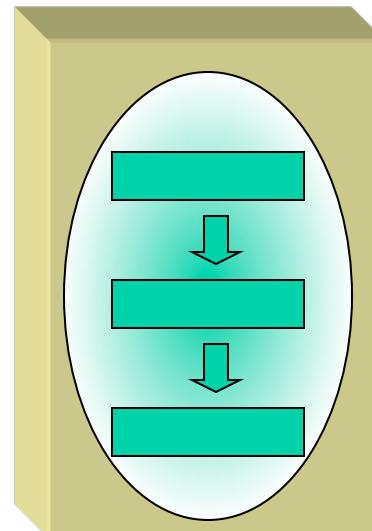
# Thread vs. Sequential Programs

---

- A thread is similar to sequential programs
- A single thread also has
  - a beginning,
  - a sequence,
  - and an end
  - and at any given time during the runtime of the thread, there is a single point of execution.
- However, a thread itself is not a program;
  - it cannot run on its own.
  - it runs within a program.

# Threads in a Program

- A thread is a single sequential flow of control within a program.
- You can also use multiple threads in a single program
  - Run at the same time and perform different tasks



# Threads as Lightweight Processes

---

- Some refer to threads as *lightweight process*
  - It runs within the context of a full-blown program
  - Takes advantage of the resources allocated for that program and the program's environment.

# Resources used by threads

---

- As a sequential flow of control, a thread must carve out some of its own resources within a running program.
  - It must have its own execution stack and program counter for example.
- The code running within the thread works only within that context.
- Thus, some use the term *execution context* as a synonym for thread.

# Thread of execution

---

- A sequence of instructions

```
for (int j = 0; j < 2; j++) {      // line 1  
    System.out.println("hello!"); // line 2  
}
```

- aka *thread of control*
- The sequence of execution is 1,2,1,2
- So far we've only looked at programs that are **single-threaded**
  - The JVM runs the program by executing a single sequence of instructions.
  - In Java applications, the thread begins execution at the first statement in `main`.

# Why to avoid threads

---

- Difficult to use
- Make programs harder to debug
- Deadlock
  - You must take extreme care that any threads you create don't invoke any methods on Swing components
    - Swing components are not thread safe
    - Once a Swing component has been created and displayed, only the event-dispatching thread should affect or query the component
- But...

# Threads can be invaluable

---

- You can use them to improve your program's perceived performance
  - With GUI applications, you want to avoid making it look like the application is “stuck” or slow, so you may display a wait message while saving a large file.
- Sometimes threads can simplify a program's code or architecture
  - For timed events, instead of constantly looping and polling the system for the time, you can use the Timer class to notify you when an amount of time has lapsed.

# Uses for threads

---

- Some typical uses for threads include:
  - To move a time-consuming initialization task out of the main thread, so that the GUI comes up faster.
    - Examples of time-consuming tasks include making extensive calculations and blocking for network or disk I/O (loading images, for example).
  - To move a time-consuming task out of the event-dispatching thread, so that the GUI remains responsive
  - To perform an operation repeatedly, usually with some predetermined period of time between operations.
  - To wait for messages from other programs

# Multi threading

---

## Multiprocessing

- Using Multiple Processors.

## Multitasking

Perform more than one task at a time for each user, to improve response time.

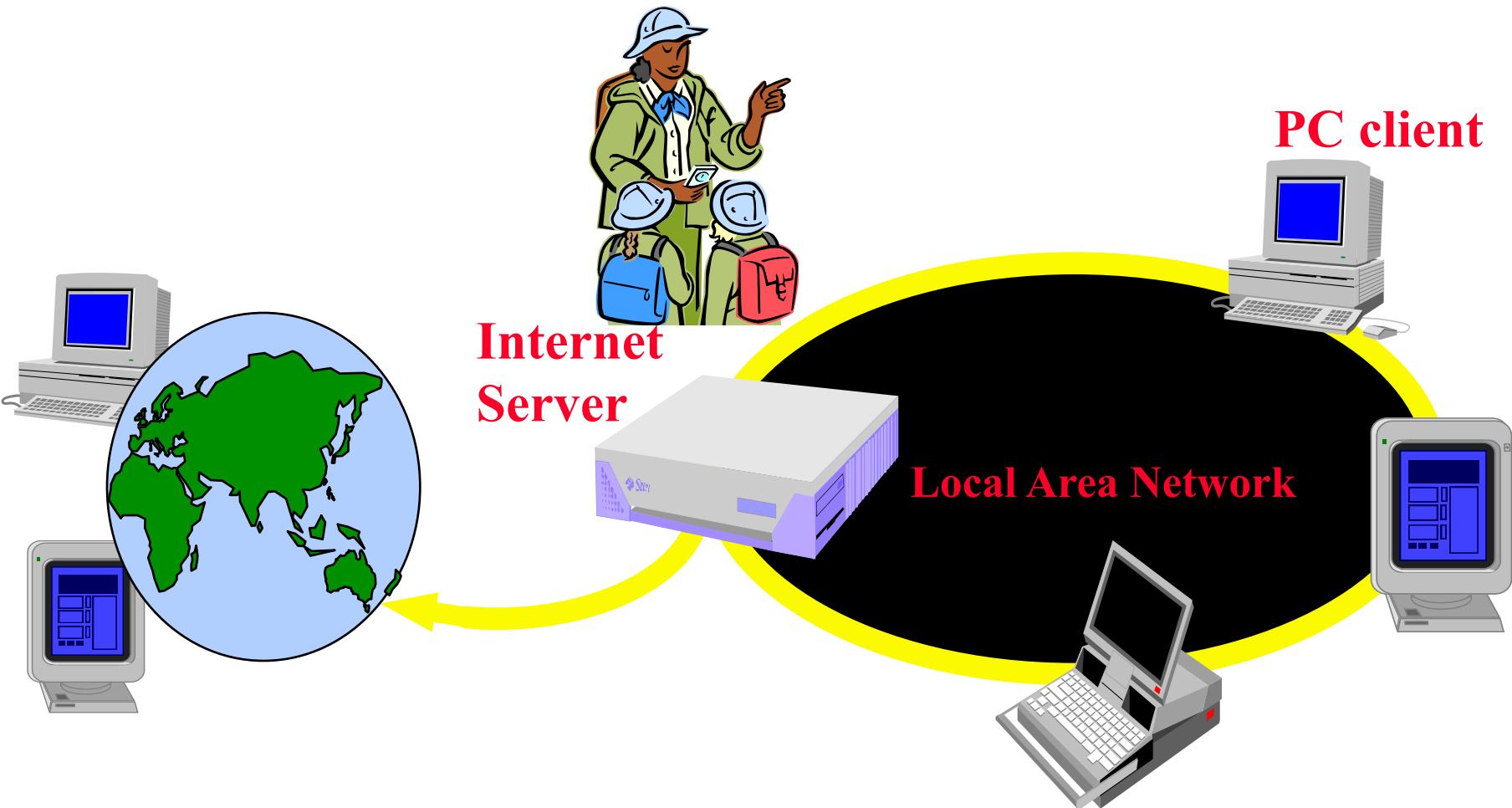
separate processes run independent of each other, each accessing its own memory space, cooperating perhaps through system/OS implemented interprocessing communication (IPC) shared memory, semaphores, etc.

## Multithreading

- in multithreading, threads share code segments, inherently work in the same memory space (under one process, the Java VM)
- Threads in a user task (process) can *share* code, data, and system resources, and access them concurrently.
- Each thread has a separate program counter, registers, and a run-time stack.

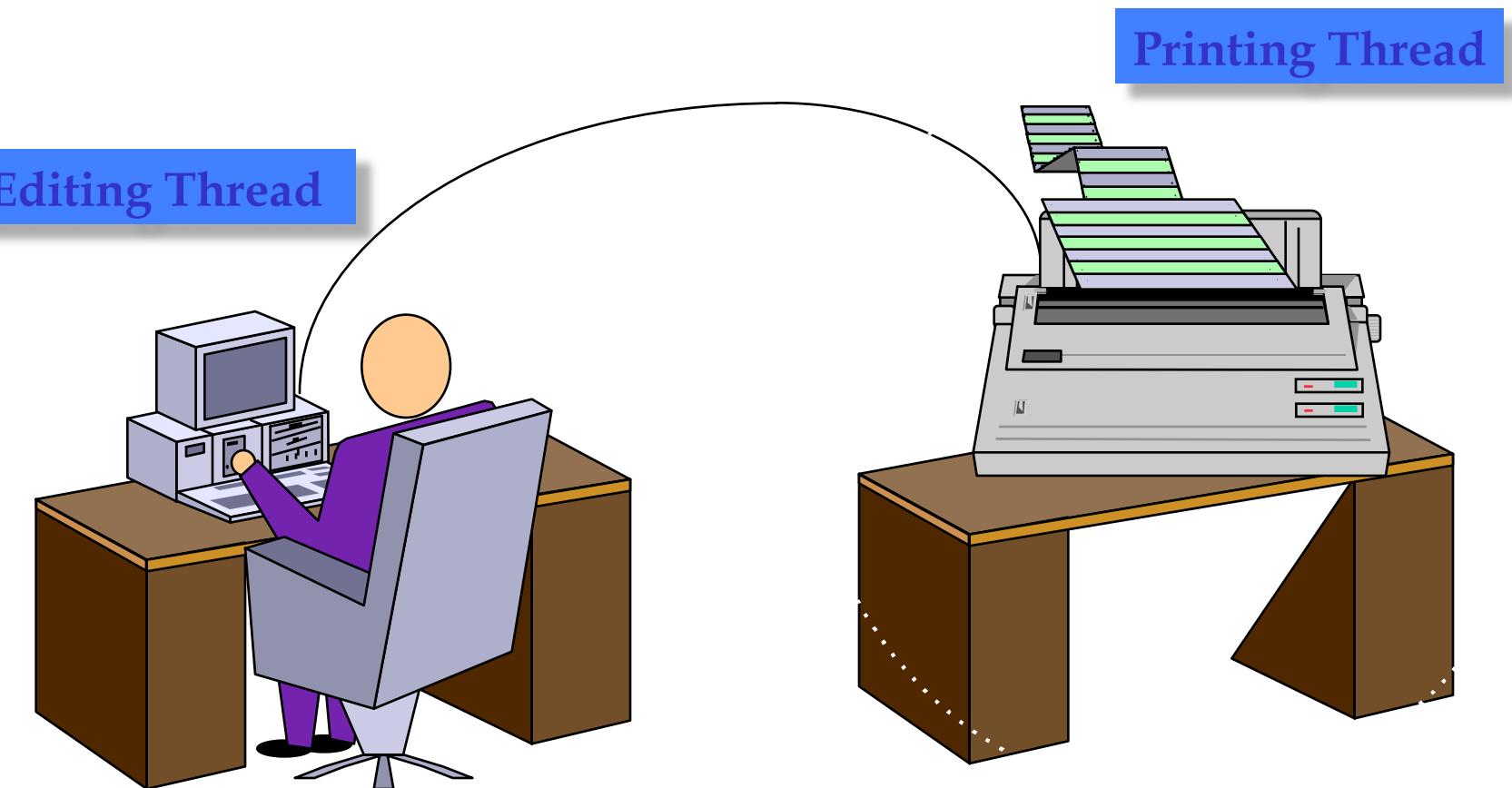
# Web/Internet Applications:

## Serving Many Users Simultaneously



# Modern Applications need Threads:

Editing and Printing documents in background.



# Java Threads

---

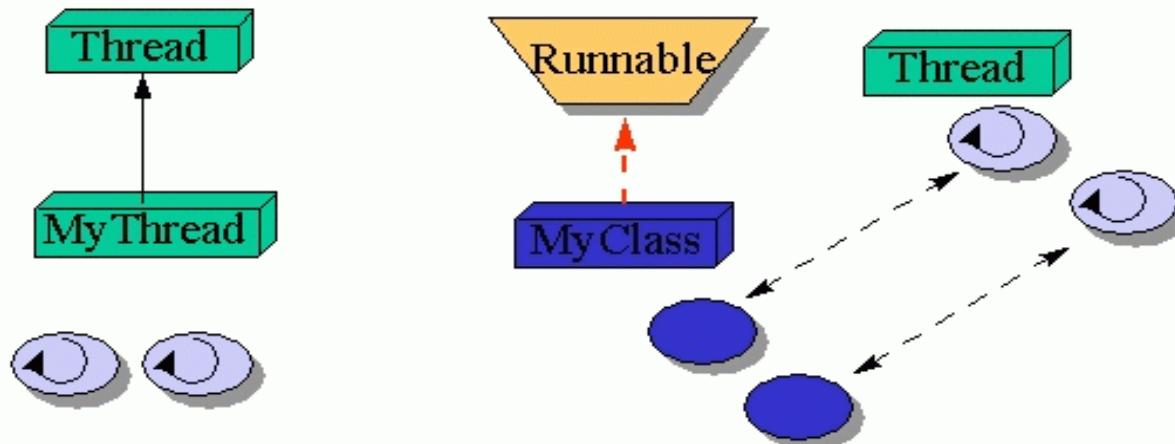
- Java has built in thread support for Multithreading
  - Synchronization
  - Thread Scheduling
- Inter-Thread Communication:

|                 |       |             |
|-----------------|-------|-------------|
| – currentThread | start | setPriority |
| – yield         | run   | getPriority |
| – sleep         | stop  | suspend     |
| – resume        |       |             |
- Java Garbage Collector is a low-priority thread.

# How to create Java Threads

- There are two ways to create a Java thread:
  - Extend the `java.lang.Thread` class
  - Implement the `java.lang.Runnable` interface

## Threading Mechanisms



# Extending the Thread class

---

- In order to create a new thread we may subclass `java.lang.Thread` and customize what the thread does by overriding its empty `run` method.
- The `run` method is where the action of the thread takes place.
- The execution of a thread starts by calling the `start` method.

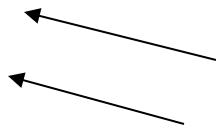
# Demo

---

```
class MyThread extends Thread {  
    private String name, msg;  
    public MyThread(String name, String msg) {  
        this.name = name;  
        this.msg = msg;  
    }  
    public void run() {  
        System.out.println(name + " starts its execution");  
        for (int i = 0; i < 5; i++) {  
            System.out.println(name + " says: " + msg);  
            try { Thread.sleep(5000);  
            }catch (InterruptedException ie) {  
                }  
        }  
        System.out.println(name + " finished execution");  
    }  
}
```

# Example I (cont.)

```
public class test {  
    public static void main(String[] args) {  
        MyThread mt1 = new MyThread("thread1", "ping");  
        MyThread mt2 = new MyThread("thread2", "pong");  
        mt1.start();  
        mt2.start();  
    }  
}
```



The threads will run in parallel

# Example I (cont.)

---

Typical output of the previous example:

thread1 starts its execution

thread1 says: ping

thread2 starts its execution

thread2 says: pong

thread1 says: ping

thread2 says: pong

thread1 finished execution

thread2 finished execution

# Implementing the Runnable interface

---

- In order to create a new thread we may also provide a class that implements the `java.lang.Runnable` interface
- Preferred way in case our class has to subclass some other class
- A Runnable object can be wrapped up into a Thread object
  - `Thread(Runnable target)`
  - `Thread(Runnable target, String name)`
- The thread's logic is included inside the run method of the runnable object

# Example II

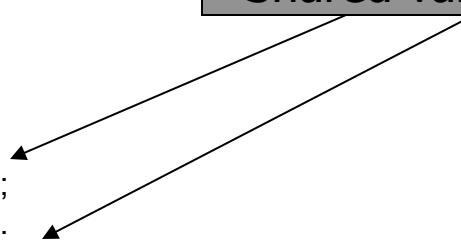
---

```
class MyClass implements Runnable {  
    private String name;  
    private A sharedObj;  
    public MyClass(String name, A sharedObj) {  
        this.name = name; this.sharedObj = sharedObj;  
    }  
    public void run() {  
        System.out.println(name + " starts execution");  
        for (int i = 0; i < 5; i++) {  
            System.out.println(name + " says: " + sharedObj.getValue());  
            try {Thread.sleep(5000);  
            }catch(InterruptedException ie) {  
            }  
        }  
        System.out.println(name + " finished execution");  
    }  
}
```

# Example II (cont.)

```
class A {  
    private String value;  
    public A(String value) {  
        this.value = value;  
    }  
    public String getValue() {  
        return value;  
    }  
}  
  
public class test2 { public static void main(String[] args) {  
    A sharedObj= new A("some value");  
    Thread mt1 = new Thread(new MyClass("thread1", sharedObj));  
    Thread mt2 = new Thread(new MyClass("thread2", sharedObj));  
    mt1.start(); mt2.start();  
}
```

Shared variable



# Example II (cont.)

---

- Typical output of the previous example:

thread1 starts execution

thread1 says: some value

thread2 starts execution

thread2 says: some value

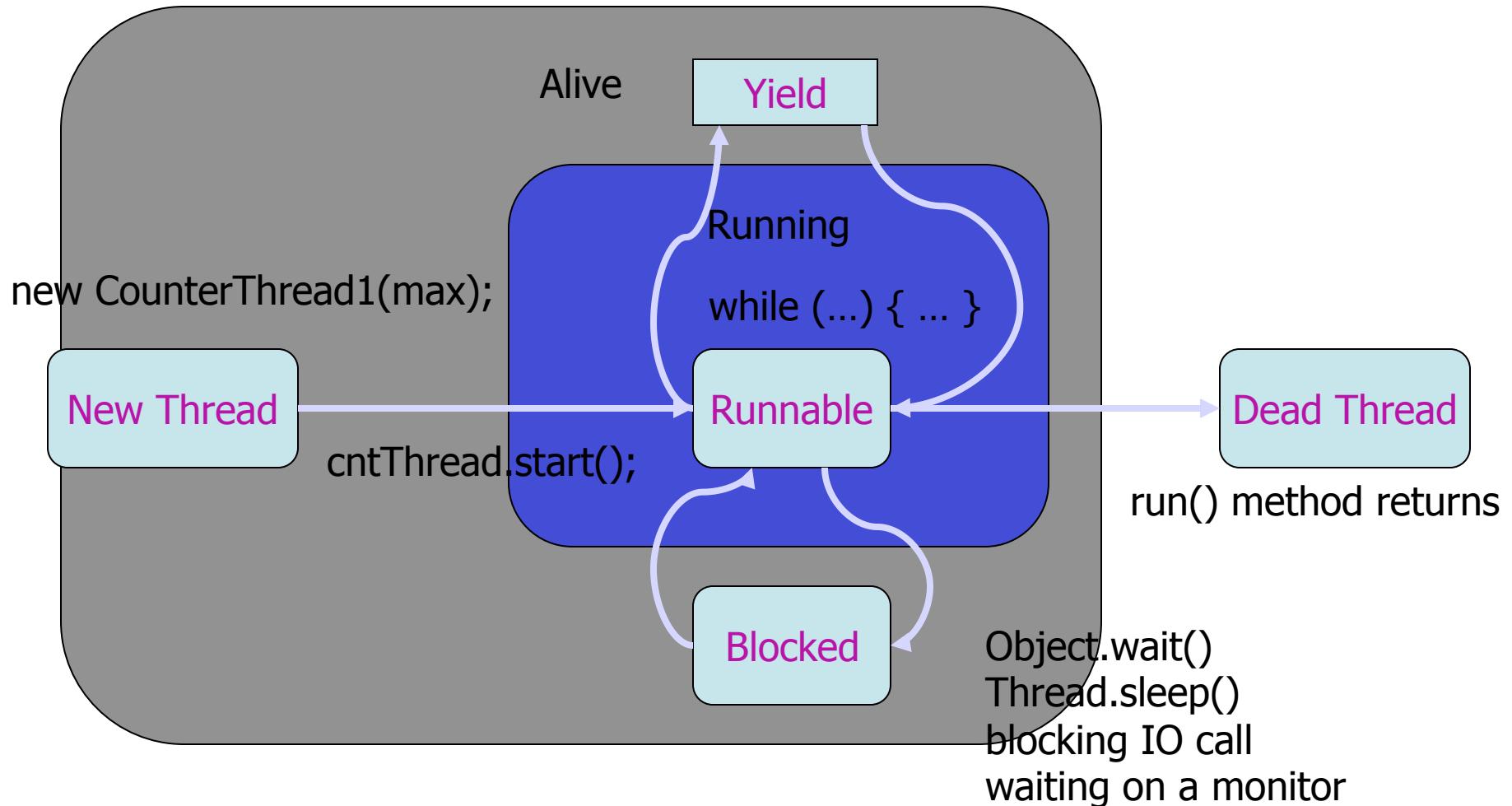
thread1 says: some value

thread2 says: some value

thread1 finished execution

thread2 finished execution

# Thread State Diagram



# The lifecycle of a thread(1)

---

- The diagram shows the states that a Java thread can be in during its life.
- It also illustrates which method calls cause a transition to another state.

# The lifecycle of a thread(2)

```
Thread t = new Thread(this); // New Thread  
t.start();
```

- When a thread is a New Thread (or in its initial state), it is merely an empty Thread object
  - no system resources have been allocated for it yet.
- When a thread is in this state, you can only start the thread.
  - Calling any method besides start when a thread is in this state causes an **IllegalThreadStateException**.

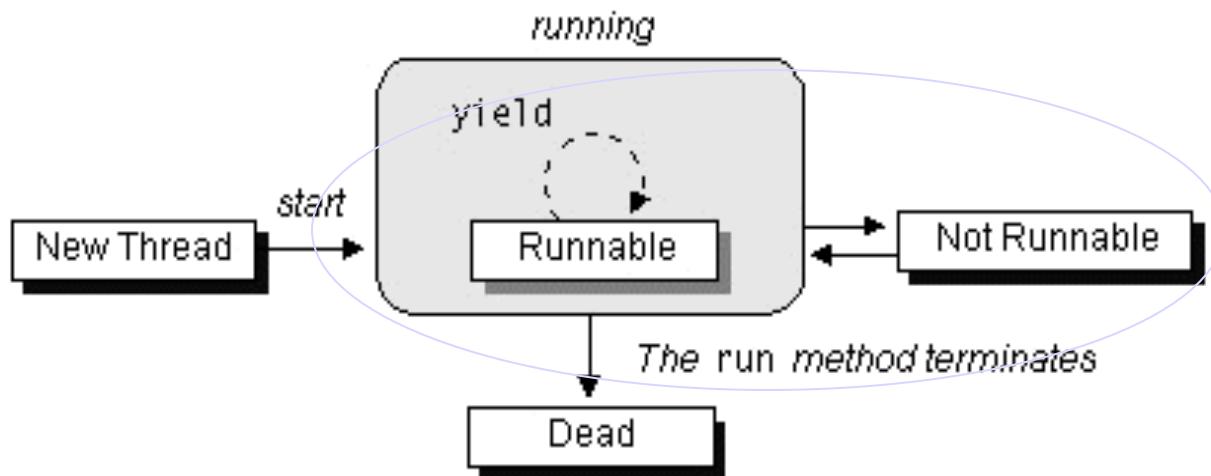
# The lifecycle of a thread (3)

```
Thread t = new Thread(this);      // New Thread  
t.start(); // starting a thread
```

- The start method:
  - creates the system resources necessary to run the thread,
  - schedules the thread to run,
  - and calls the thread's run method
- After the start method has returned, the thread is "running". Yet, it's somewhat more complex than that...

# The lifecycle of a thread(4)

- A thread that has been started is in the Runnable state.
- Many computers have a single processor, thus making it impossible to run all "running" threads at the same time.
- The Java runtime system must implement a scheduling scheme that shares the processor between all "running" threads.
- So at any given time, a "running" thread actually may be waiting for its turn in the CPU.



# The lifecycle of a thread(5)

---

- A thread becomes Not Runnable (or **blocked**) when one of these events occurs:
  - Its sleep method is invoked.
  - The thread calls the wait method to wait for a specific condition to be satisfied.
  - The thread is blocking on I/O.

# The lifecycle of a thread

---

- For each entrance into the Not Runnable state, there is a specific and distinct escape route that returns the thread to the Runnable state.
  - If a thread has been put to sleep, then the specified number of milliseconds must elapse.
  - If a thread is waiting for a condition, then another object must notify the waiting thread of a change in condition by calling `notify` or `notifyAll`
  - If a thread is blocked on I/O, then the I/O must complete.

# Stopping a thread

---

- A program doesn't stop a thread like it stops an applet (by calling a method).
- Rather, a thread arranges for its own death by having a run method that terminates naturally.
- For example, the for loop in this run method is a finite loop-- it will iterate 100 times and then exit:

```
public void run() {  
    for (int i = 0; i < 100; i++) {  
        System.out.println("i = " + i);  
    }  
}
```

- A thread with this run method dies naturally when the loop completes and the run method exits.

# Thread state

---

- The API for the Thread class includes a method called `isAlive()`.
- The `isAlive` method returns true if the thread has been started and not stopped.
- If the `isAlive` method returns **false**, the thread either is:
  - a New Thread
  - or is Dead.

# Thread state

---

- If the `isAlive` method returns **true**, you know that the thread is either:
  - Runnable
  - or Not Runnable.
- You cannot differentiate between a New Thread or a Dead thread, nor can you differentiate between a Runnable thread and a Not Runnable thread.

# Deprecated thread methods

---

- The **stop** method has been deprecated.
- The preferred way to stop a thread is to have the thread return from **run()**.
  - A stopped thread releases any held locks.
- The **suspend** and **resume** methods have been deprecated
  - **suspend** is deadlock-prone and
  - **resume** is only used with **suspend**

# Thread Priority

---

- On a single CPU threads actually run one at a time in such a way as to provide an illusion of concurrency.
- Execution of multiple threads on a single CPU, in some order, is called scheduling.
- The Java runtime supports a very simple scheduling algorithm (fixed priority scheduling). This algorithm schedules threads based on their priority relative to other runnable threads.

# Thread Priority

---

- The runtime system chooses the runnable thread with the highest priority for execution.
- If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.
- The chosen thread will run until:
  - A higher priority thread becomes runnable.
  - It yields, or its `run` method exits.
  - On systems that support time-slicing, its time allotment has expired

# Thread Scheduling

---

- The Java runtime system's thread scheduling algorithm is also *preemptive*.
- If at any time a thread with a higher priority than all other runnable threads becomes runnable, the runtime system chooses the new higher priority thread for execution.
- The new higher priority thread is said to *preempt* the other threads.
- At any given time, the highest priority thread is running.  
However, this is not guaranteed.

# Thread Scheduling

---

- The Java runtime will not preempt the currently running thread for another thread of the same priority.
  - In other words, the Java runtime does not time-slice.
- However, the system implementation of threads underlying the Java Thread class may support time-slicing.
  - Do not write code that relies on time-slicing.
- In addition, a given thread may, at any time, give up its right to execute by calling the yield method.
  - Threads can only yield the CPU to other threads of the same priority--attempts to yield to a lower priority thread are ignored.

# Thread Priority

---

- When a Java thread is created, it inherits its priority from the thread that created it.
- You can modify a thread's priority at any time after its creation using the `setPriority` method.
- Thread priorities are integers ranging between `MIN_PRIORITY` and `MAX_PRIORITY` (constants defined in the `Thread` class).
- The higher the integer, the higher the priority.

# Thread Priorities: Example

```
package java.lang;  
public class Thread extends Object  
    implements Runnable {  
    // constants  
    public static final int MAX_PRIORITY = 10;  
    public static final int MIN_PRIORITY = 1;  
    public static final int NORM_PRIORITY = 5;  
    // methods  
    public final int getPriority();  
    public final void setPriority(int newPriority);  
    public static void yield();  
    ...  
}
```

# Warning

---

- From a real-time perspective, Java's scheduling and priority models are weak; in particular:
  - no guarantee is given that the highest priority runnable thread is always executing
  - equal priority threads may or may not be time sliced
  - where native threads are used, different Java priorities may be mapped to the same operating system priority

# Thread Priority Example

```
class A extends Thread
{
    public void run()
    {
        System.out.println("Thread A started");
        for(int i=1;i<=4;i++)
        {
            System.out.println("\t From ThreadA: i= "+i);
        }
        System.out.println("Exit from A");
    }
}

class B extends Thread
{
    public void run()
    {
        System.out.println("Thread B started");
        for(int j=1;j<=4;j++)
        {
            System.out.println("\t From ThreadB: j= "+j);
        }
        System.out.println("Exit from B");
    }
}
```

# Thread Priority Example

```
class C extends Thread
{
    public void run()
    {
        System.out.println("Thread C started");
        for(int k=1;k<=4;k++)
        {
            System.out.println("\t From ThreadC: k= "+k);
        }
        System.out.println("Exit from C");
    }
}
```

# Thread Priority Example

```
class ThreadPriority
{
    public static void main(String args[])
    {
        A threadA=new A();
        B threadB=new B();
        C threadC=new C();

        threadC.setPriority(Thread.MAX_PRIORITY);
        threadB.setPriority(threadA.getPriority()+1);
        threadA.setPriority(Thread.MIN_PRIORITY);

        System.out.println("Started Thread A");
        threadA.start();
        System.out.println("Started Thread B");
        threadB.start();
        System.out.println("Started Thread C");
        threadC.start();
        System.out.println("End of main thread");
    }
}
```

# Thread Priority Example

Output:

Thread A started

Thread B started

Thread B started

from thread B: 0

from thread B: 1

from thread B: 2

from thread B: 3

from thread B: 4

from thread B: 5

from thread B: 6

from thread B: 7

from thread B: 8

from thread B: 9

exit from thread B

Thread C started

Thread C started

from thread C: 0

from thread C: 1

from thread C: 2

from thread C: 3

from thread C: 4

from thread C: 5

from thread C: 6

from thread C: 7

from thread C: 8

from thread C: 9

exit from thread C

End of Main Thread

Thread A started

from thread A: 0

from thread A: 1

from thread A: 2

from thread A: 3

from thread A: 4

from thread A: 5

from thread A: 6

from thread A: 7

from thread A: 8

from thread A: 9

exit from thread A

# Putting threads to sleep

---

- Java allows us to suspend the execution of a thread by issuing a sleep command

```
public static void sleep(long millisecs)
```

- putting a thread to sleep is risky – it might sleep so long that it is no longer useful and should be terminated
- therefore `sleep()` throws **InterruptedException**
- the `Exception` must be caught and handled (typically to clean-up the thread and then terminate it)

# Example

```
public class Pinger extends Thread {
```

```
    public void run() {
```

```
        try {
```

```
            for (int i = 0; i<10; i++) {
```

```
                System.out.println("ping ...");
```

```
                sleep(1000);
```

```
        }
```

```
}
```

```
    catch(InterruptedException e) {
```

```
        System.out.println("Pinger interrupted");
```

```
}
```

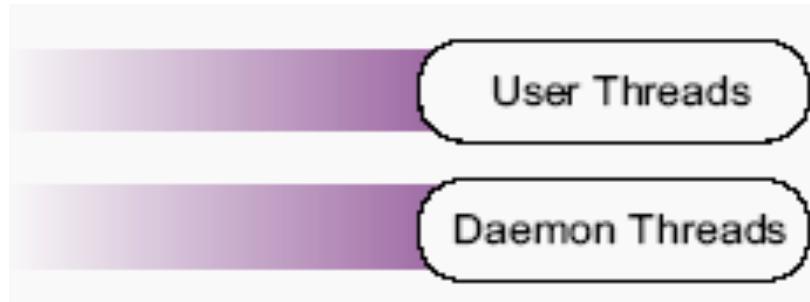
```
}
```

so place all the runnable code inside "try"

may throw InterruptedException

... and catch the exception

# Thread Types



- User threads are the most common.
  - Programmers create them.
  - The main thread is an example.
- The JVM automatically creates daemon threads.
  - They run in the background.
  - They perform services, such as garbage collection.
  - Programmers can create them by calling `setDaemon(true)` on a particular thread.

# User and Daemon threads

---

- The main thread is the application's only user thread, so the application terminates when main returns
- A user thread constructs other user threads
- A daemon thread constructs other daemon threads
- The setDaemon method must be invoked before the thread is started
- The JVM's garbage collector runs as a daemon thread because the JVM itself should terminate if no user threads are alive.

# User and Daemon threads

---

- A *daemon* thread serves *user threads*.
- An application continues to run as long as at least one of its user threads is alive.
- when a user thread stops in an application, the JVM checks whether any other user threads are still alive.
  - If so, the application continues, otherwise
  - the application terminates because the JVM itself halts.

# 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  
**`java.lang.ThreadGroup`**.

# Thread groups

---

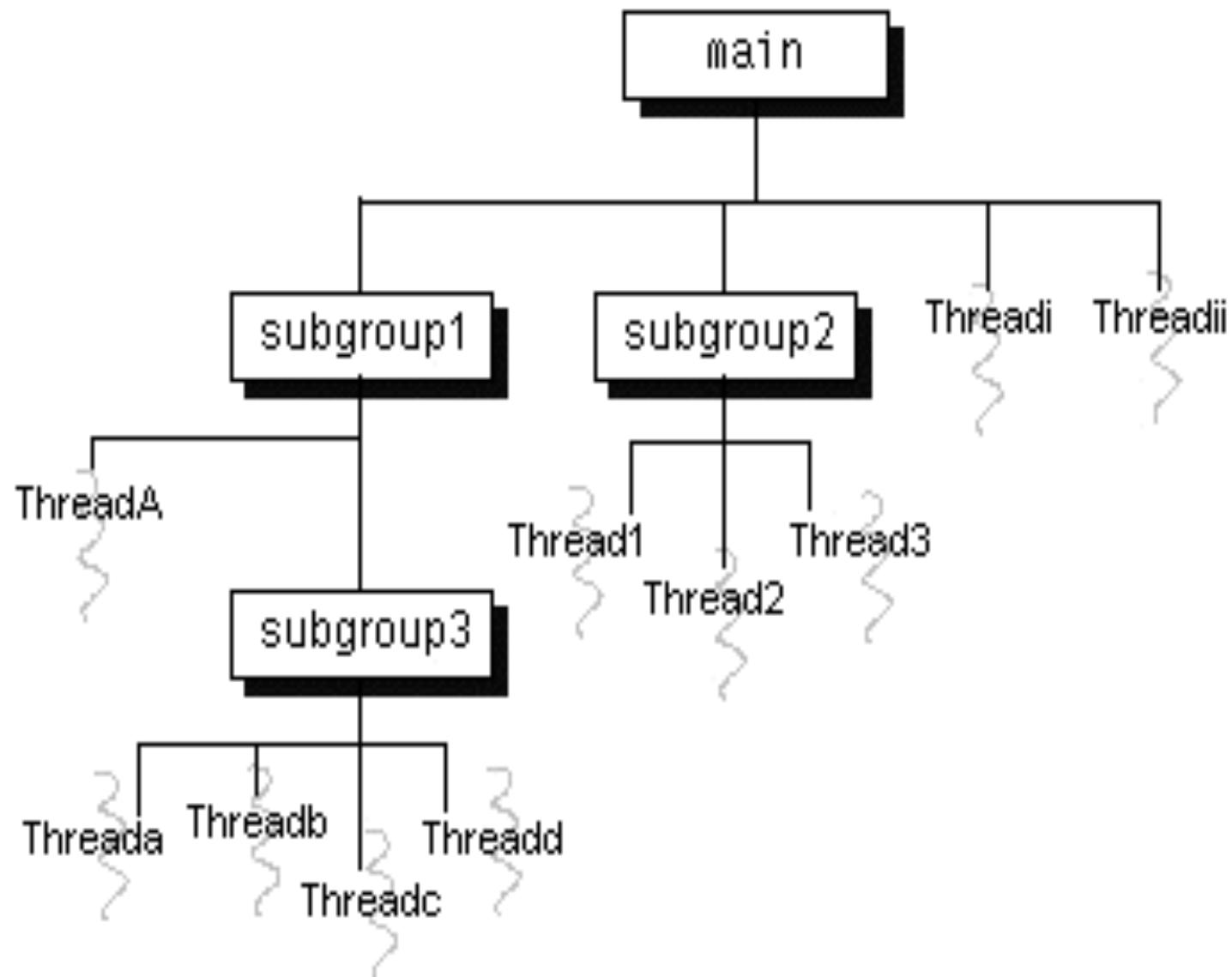
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either:
  - allow the runtime system to put the new thread in some reasonable default group
  - or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation
  - you cannot move a thread to a new group after the thread has been created.

# The Default Thread Group

---

- If you create a new Thread without specifying its group in the constructor, the runtime system automatically places the new thread in the same group (*current thread group*) as the thread that created it (*current thread*)
- When a Java application first starts up, the Java runtime system creates a ThreadGroup named main.
- Unless specified otherwise, all new threads that you create become members of the main thread group.

# The Default Thread Group



# Why Thread Groups?

---

- Many Java programmers ignore thread groups altogether and allow the runtime system to handle all of the details regarding thread groups.
- However, if your program creates a lot of threads that should be manipulated as a group, or if you are implementing a custom security manager, you will likely want more control over thread groups.

# Specifying a thread group

- 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 *runnable*)

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

public Thread(ThreadGroup *group*, Runnable *runnable*, 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

# Specifying a thread group

- The following code sample creates a thread group (`myThreadGroup`) and then creates a thread (`myThread`) in that group:

```
ThreadGroup myThreadGroup =  
    new ThreadGroup("My Group of Threads");  
  
Thread myThread = new Thread(myThreadGroup,  
    "a thread for my group");
```

- The `ThreadGroup` passed into a `Thread` constructor does not necessarily have to be a group that you create
  - It can be a group created by the Java runtime system, or a group created by the application in which your applet is running.

# Getting a thread's group

---

- To find out what group a thread is in, you can call its `getThreadGroup` method:

```
theGroup = myThread.getThreadGroup();
```

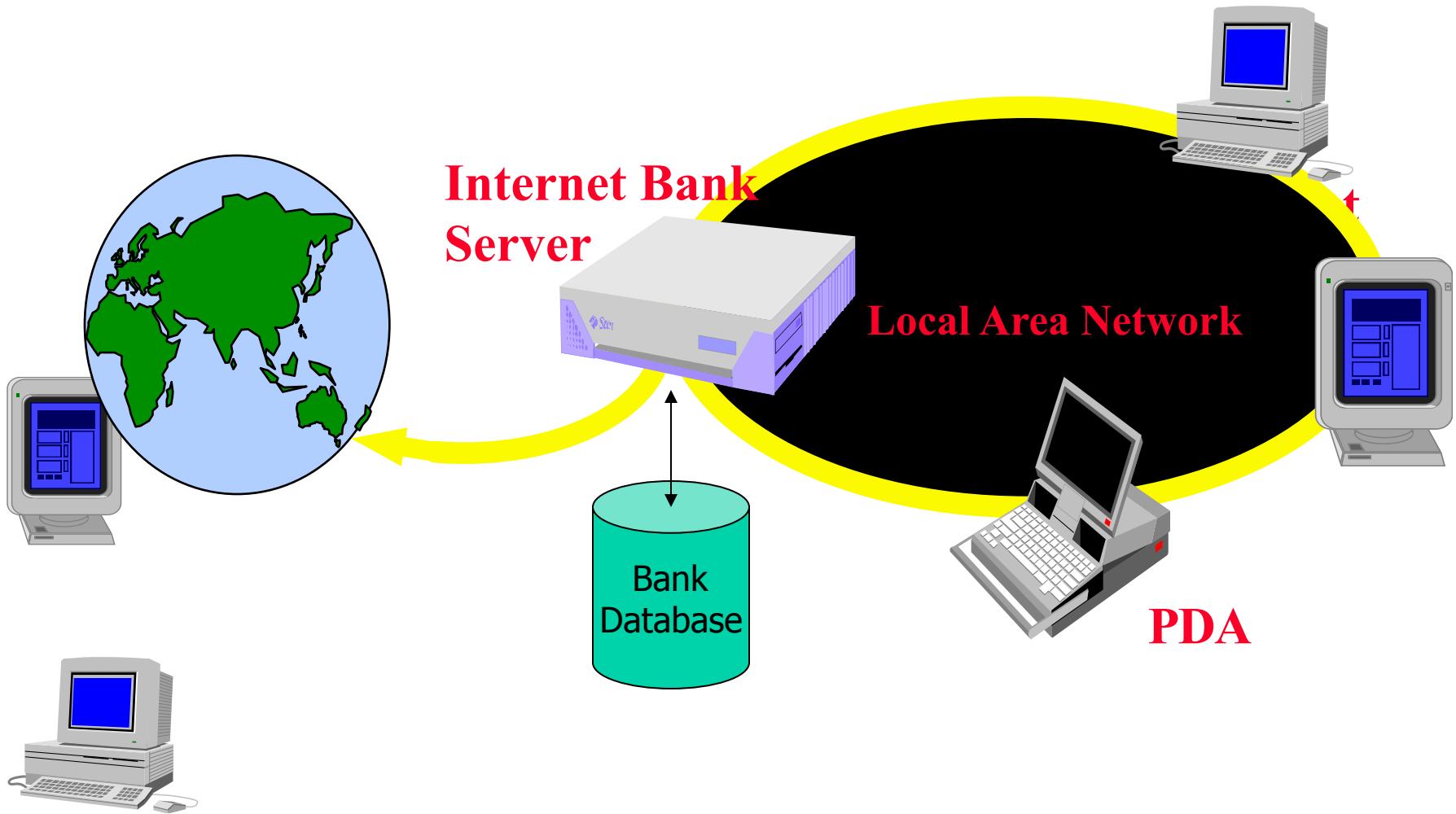
- Once you've obtained a thread's `ThreadGroup`, you can query the group for information, such as what other threads are in the group.
- You can also modify the threads in that group, such as suspending, resuming, or stopping them, with a single method invocation.

# Thread synchronization

---

- So far, we've looked at independent, asynchronous threads.
  - Each thread contained all of the data and methods required for its execution
  - The threads ran at their own pace
    - Don't have to worry about the speed of other threads.
- However, what happens when data is shared between threads?
  - Simultaneous operations on your bank account

# Online Bank: Serving Many Customers and Operations



# Shared Resources



- If one thread tries to read the data and other thread tries to update the same date, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- In Java: “Synchronized” method:
  - `public synchronized void update()`

```
{
```

```
...
```

```
}
```

# Producer/Consumer Examples

---

- You have a Java application where one thread (the producer) writes data to a file while a second thread (the consumer) reads data from the same file.
- Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue.
- Both of these examples use concurrent threads that share a common resource:
  - the first shares a file,
  - the second shares an event queue.
- Because the threads share a common resource, they must be synchronized in some way.

# Producer/Consumer (2)

- The Producer generates an integer between 0 and 9 (inclusive), stores it in a CubbyHole object, and prints the generated number. The Producer sleeps for a random amount of time between 0 and 100 milliseconds before repeating the number generating cycle:

```
public class Producer extends Thread {  
    private CubbyHole cubbyhole;  
    private int number;  
  
    public Producer(CubbyHole c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            cubbyhole.put(i);  
            System.out.println("Producer #" + this.number  
                               + " put: " + i);  
            try {  
                sleep((int)(Math.random() * 100));  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```

# Producer/Consumer (3)

- The Consumer, being ravenous, consumes all integers from the CubbyHole (the exact same object into which the Producer put the integers in the first place) as quickly as they become available.

```
public class Consumer extends Thread {  
    private CubbyHole cubbyhole;  
    private int number;  
  
    public Consumer(CubbyHole c, int number) {  
        cubbyhole = c;  
        this.number = number;  
    }  
  
    public void run() {  
        int value = 0;  
        for (int i = 0; i < 10; i++) {  
            value = cubbyhole.get();  
            System.out.println("Consumer #" + this.number  
                + " got: " + value);  
        }  
    }  
}
```

# Producer/Consumer (4)

---

- The Producer and Consumer in this example share data through a common CubbyHole object.
- Neither the Producer nor the Consumer makes sure that the Consumer is getting each value produced once and only once.
  - The synchronization between these two threads actually occurs at a lower level, within the `get` and `put` methods of the CubbyHole object.
- However, let's assume that the two threads make no arrangements for synchronization and talk about the potential problems...

# Producer/Consumer (5)

---

- One problem arises when the Producer is quicker than the Consumer and generates two numbers before the Consumer has a chance to consume the first one.
- Thus the Consumer would skip a number.
- Part of the output might look like this:

... .

Consumer #1 got: 3

Producer #1 put: 4

Producer #1 put: 5

Consumer #1 got: 5

... .

# Producer/Consumer (6)

---

- Another problem that might arise is when the Consumer is quicker than the Producer and consumes the same value twice.
- In this situation, the Consumer would print the same value twice and might produce output that looked like this:

```
...  
Producer #1 put: 4  
Consumer #1 got: 4  
Consumer #1 got: 4  
Producer #1 put: 5  
...
```

# Producer/Consumer (7)

---

- Either way, the result is wrong.
- You want the Consumer to get each integer produced by the Producer exactly once.
- Problems such as those just described are called *race conditions*.
  - They arise from multiple, asynchronously executing threads trying to access a single object at the same time and getting the wrong result.
- Race conditions in the producer/consumer example are prevented by having the storage of a new integer into the CubbyHole by the Producer be **synchronized** with the retrieval of an integer from the CubbyHole by the Consumer. The Consumer must consume each integer exactly once.

# Synchronizing the Producer/Consumer problem

---

- The activities of the Producer and Consumer must be synchronized in two ways:
  - The two threads must not simultaneously access the CubbyHole.
    - Need to use locks
  - The two threads must do some simple coordination. The Producer must have some way to indicate to the Consumer that the value is ready and the Consumer must have some way to indicate that the value has been retrieved.
    - The Thread class provides a collection of methods--`wait`, `notify`, and `notifyAll`--to help threads wait for a condition and notify other threads of when that condition changes

# Locking an Object

---

- The code segments within a program that access the same object from separate, concurrent threads are called *critical sections*.
- Critical sections
  - Can be a block or a method
  - Identified by synchronized keyword.
- The Java platform then associates a lock with every object that has synchronized code.

# Locking the CubbyHole

---

- In the **producer/consumer** example, the put and get methods of the CubbyHole are the critical sections.
- The Consumer should not access the CubbyHole when the Producer is changing it, and the Producer should not modify it when the Consumer is getting the value.
- So put and get in the CubbyHole class should be marked with the **synchronized keyword**.

# Looking at the CubbyHole

---

```
public class CubbyHole {  
    private int contents;  
    private boolean available = false;  
    public synchronized int get() { ... }  
    public synchronized void put(int value) { ... }  
}
```

- Note that the method declarations for both put and get contain the synchronized keyword.
  - The system associates a unique lock with every instance of CubbyHole (including the one shared by the Producer and the Consumer).
- Whenever control enters a synchronized method, the thread that called the method **locks the object** whose method has been called.
- Other threads cannot call a synchronized method on the same object until the object is unlocked.

# Locking an Object

---

- So, when the Producer calls CubbyHole's put method, it locks the CubbyHole, thereby preventing the Consumer from calling the CubbyHole's get method:

```
public synchronized void put(int value) {  
    // CubbyHole locked by the Producer ..  
    // CubbyHole unlocked by the Producer  
}
```

- When the put method returns, the Producer unlocks the CubbyHole. Similarly, when the Consumer calls CubbyHole's get method, it locks the CubbyHole, thereby preventing the Producer from calling put:

```
public synchronized int get() {  
    // CubbyHole locked by the Consumer ...  
    // CubbyHole unlocked by the Consumer  
}
```

# Locking an Object

---

- The acquisition and release of a lock is done automatically and atomically by the Java runtime system.
  - No race conditions
  - Ensures data integrity.
- Now let's take a look at how threads communicate.

# Using notify and wait

- The CubbyHole stores its value in a private member variable called contents.
- CubbyHole has another private member variable, available, that is a boolean.
- available is true when the value has just been put but not yet gotten and is false when the value has been gotten but not yet put.
- So, here's one possible implementation for the put and get methods:

```
public synchronized int get() {      // won't work!
    if (available == true) {
        available = false;
        return contents;
    }
}
public synchronized void put(int value) {      // won't work!
    if (available == false) {
        available = true;
        contents = value;
    }
}
```

# Using notify and wait (2)

---

- As implemented, these two methods won't work.
- Look at the `get` method.
  - What happens if the Producer hasn't put anything in the CubbyHole and available isn't true? (`get` does nothing.)
  - Similarly, if the Producer calls `put` before the Consumer got the value... (`put` doesn't do anything.)
- You really want the Consumer to wait until the Producer puts something in the CubbyHole and the Producer must notify the Consumer when it's done so.
- Similarly, the Producer must wait until the Consumer takes a value (and notifies the Producer of its activities) before replacing it with a new value.
- The two threads must coordinate more fully and can use Object's `wait` and `notifyAll` methods to do so.

# Using notify and wait (3)

```
public synchronized int get() {  
    while (available == false) {  
        try {  
            // wait for Producer to put value  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    available = false;  
    // notify Producer that value has been retrieved  
    notifyAll();  
    return contents;  
}
```

# Using notify and wait (4)

---

- The code in the get method loops until the Producer has produced a new value.
- Each time through the loop, get calls the wait method.
- The wait method relinquishes the lock held by the Consumer on the CubbyHole (thereby allowing the Producer to get the lock and update the CubbyHole) and then waits for notification from the Producer.
- When the Producer puts something in the CubbyHole, it notifies the Consumer by calling notifyAll.
- The Consumer then comes out of the wait state, available is now true, the loop exits, and the get method returns the value in the CubbyHole.

## Using notify and wait (5)

```
public synchronized void put(int value) {  
    while (available == true) {  
        try {  
            // wait for Consumer to get value  
            wait();  
        } catch (InterruptedException e) {  
        }  
    }  
    contents = value;  
    available = true;  
    // notify Consumer that value has been set  
    notifyAll();  
}
```

# Using notify and wait (6)

---

- The `put` method waits for the Consumer to consume the current value before the Producer can produce a new one.
- The `notifyAll` method wakes up all threads waiting on the CubbyHole.
- The awakened threads compete for the lock.
- One thread gets it, and the others go back to waiting.
- The `Object` class also defines the `notify` method, which arbitrarily wakes up one of the threads waiting on this object

# Producer/Consumer...

- So, now our code should work the way we want it to...
- Here's a small stand-alone Java application that creates a CubbyHole object, a Producer, a Consumer, and then starts both the Producer and the Consumer.

```
public class ProducerConsumerTest {  
    public static void main(String[] args) {  
        CubbyHole c = new CubbyHole();  
        Producer p1 = new Producer(c, 1);  
        Consumer c1 = new Consumer(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```

# Producer/Consumer... (2)

---

- Here's the output of ProducerConsumerTest:

```
Producer #1 put: 0
```

```
Consumer #1 got: 0
```

```
Producer #1 put: 1
```

```
Consumer #1 got: 1
```

```
Producer #1 put: 2
```

```
Consumer #1 got: 2
```

```
Producer #1 put: 3
```

```
Consumer #1 got: 3
```

```
...
```

# Signalling

---

## Object.wait()

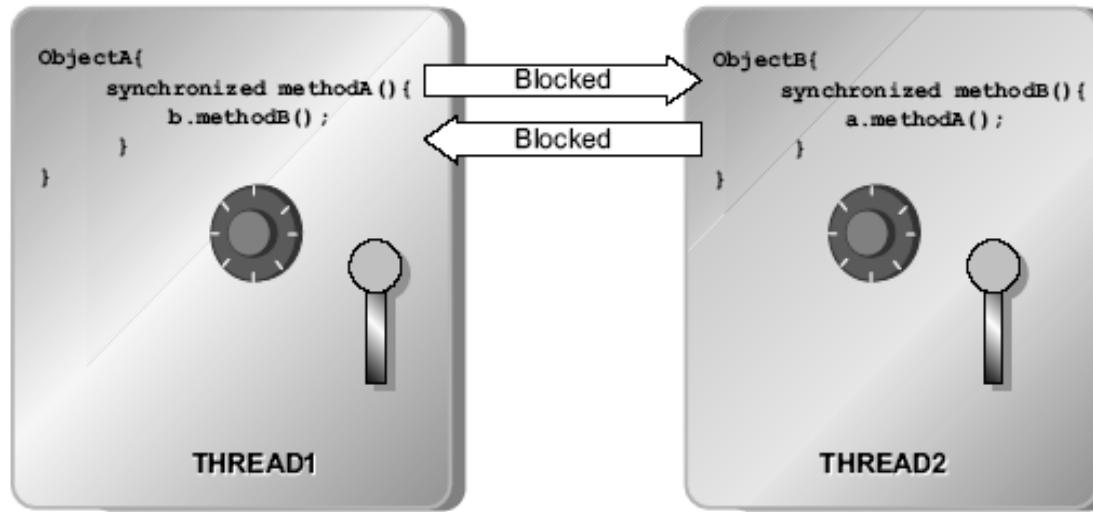
- Gives up ownership of the monitor.
- Blocks until timeout, interruption, or notification.
- On waking, the thread enters the monitor acquisition phase.

## Object.notify()

## Object.notifyAll()

- Does not give up ownership of the monitor.
- Wakes an arbitrary, or all, thread(s) blocked on the monitor.
- No thread preference!

# Deadlock



- Whenever two concurrently running threads try to access the same monitors in reverse order, it can lead to deadlock.

# Exception Handling

---

- When an exception is emitted out of the run() method, the thread is terminated.
- The exception is consumed by the JVM because once Thread.start() is called, the created thread is split from the caller.
- Java provides a means for dispatching a copy of the exception: ThreadGroup.

# Unit Summary

---

- Java thread model
- Creating a thread
- Thread properties & synchronization
- Suspending, Resuming & stopping Threads

*Welcome to*  
**JDBC**

# Unit Objective

---

- After completing this unit you should be able to
  - JDBC
  - The architecture of Java Database Connectivity
  - JDBC model
  - JDBC-ODBC bridge
  - JDBC programming
  - Result set processing
  - Other JDBC classes

# Introduction to Relational Databases

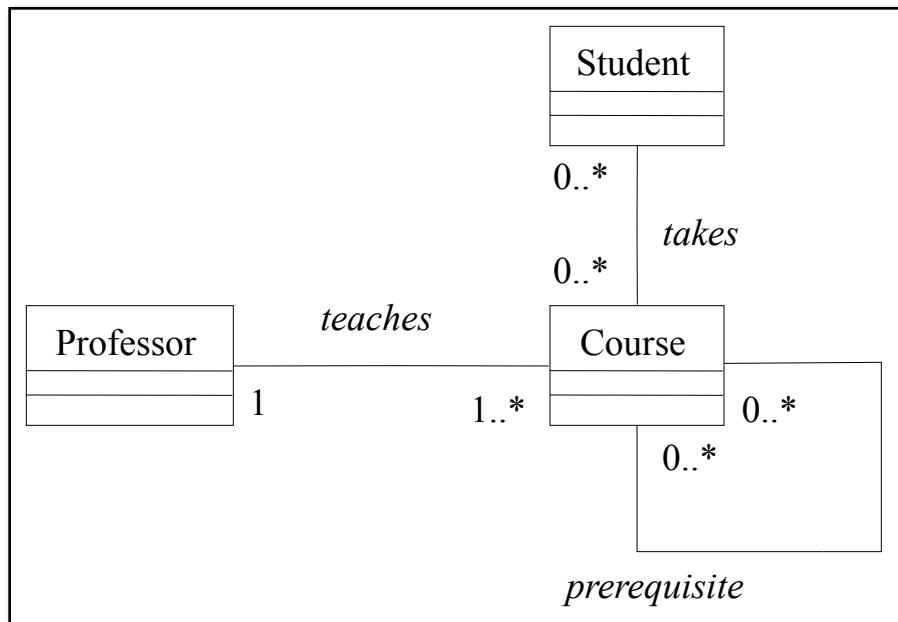
---

- Relational databases
  - Manage all information generated by businesses
  - Centric to business applications
  - Established and Mature
- Application developers require standard interface to relational databases
  - Change in database vendor should not require change in application programs

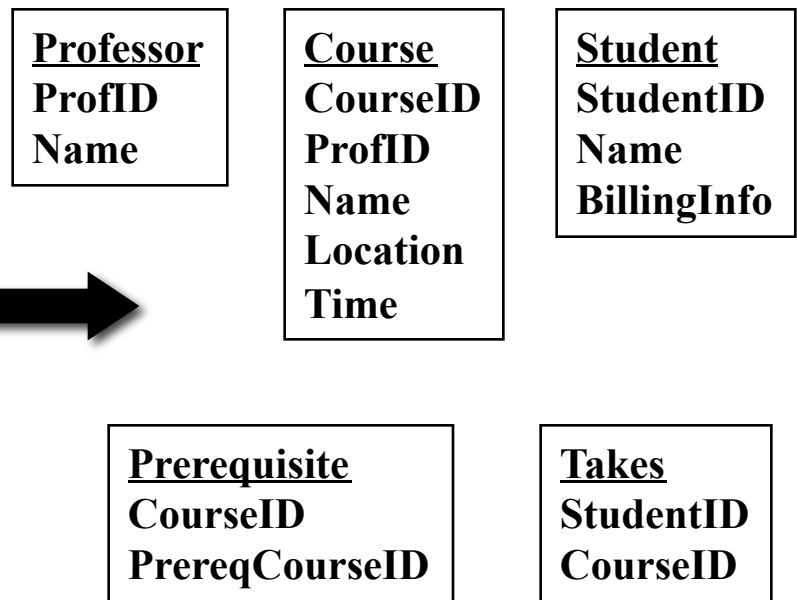
# Relational DBs

- Persistent objects map to relation tables

Class Diagram



Relation Tables

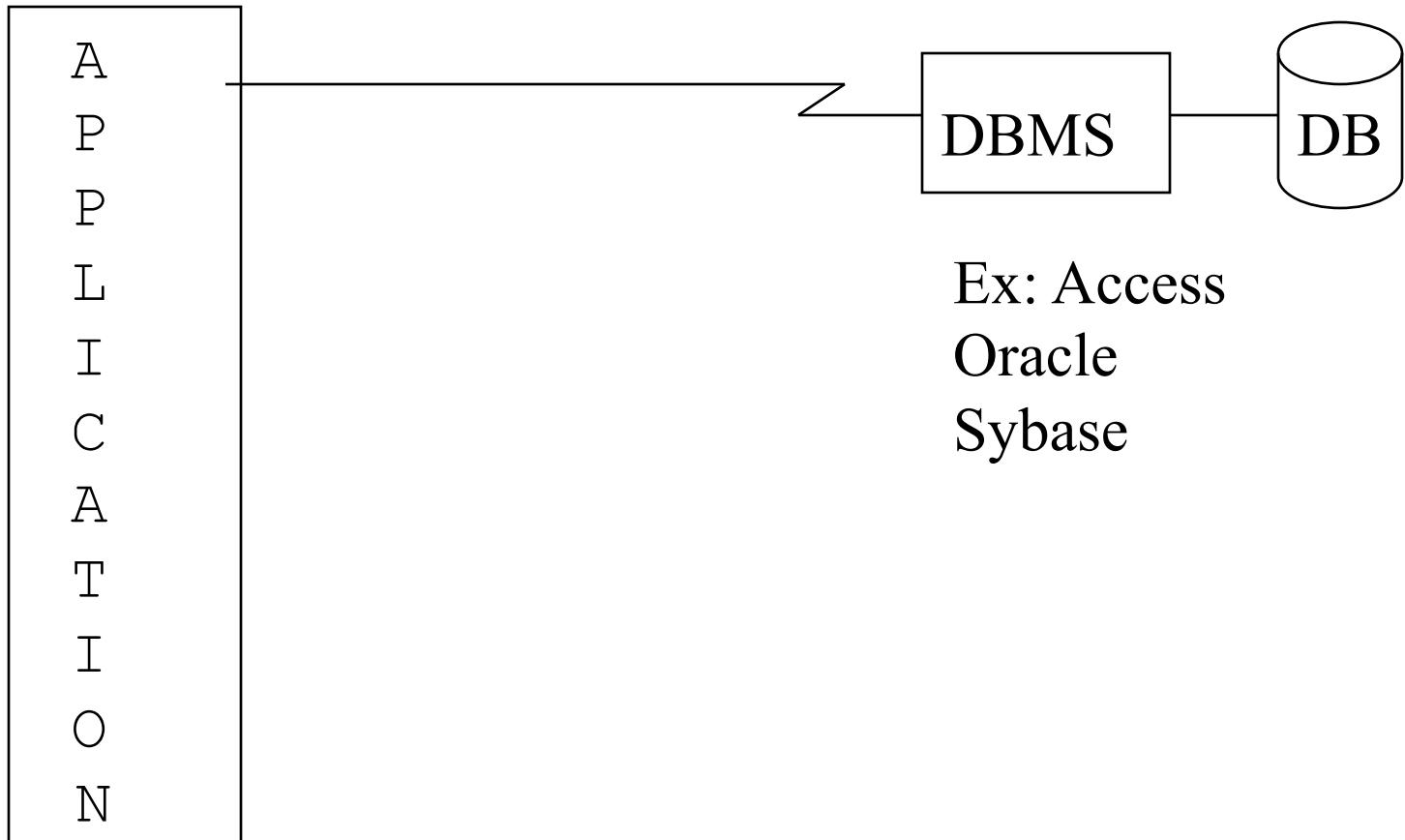


# Relational DBs

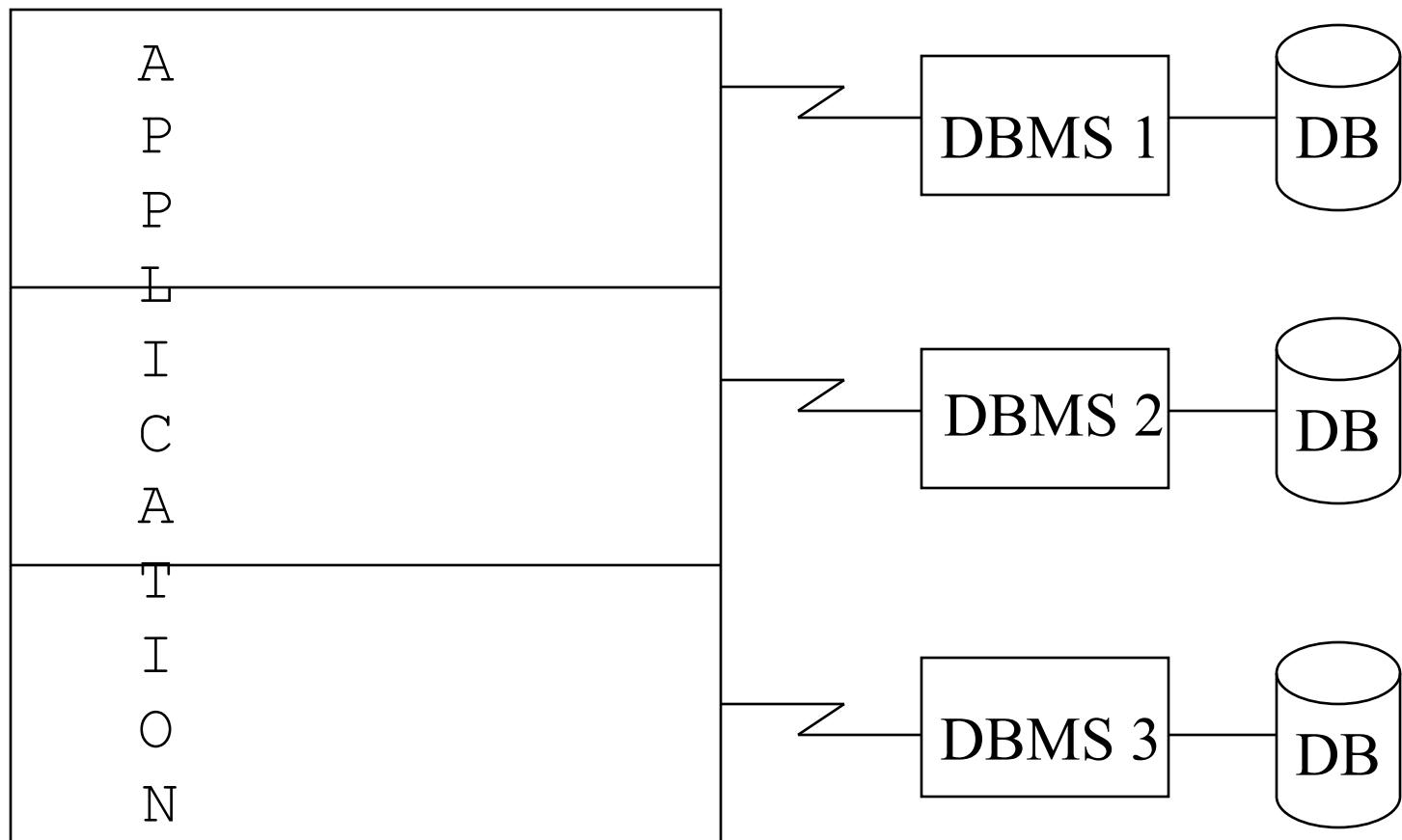
---

- Most popular form of database system is the relational database system.
- Examples: MS Access, Sybase, Oracle, MS Sequel Server.
- Structured Query Language (SQL) is used among relational databases to construct queries.
- These queries can be stand-alone or embedded within applications. This form of SQL is known as embedded SQL.

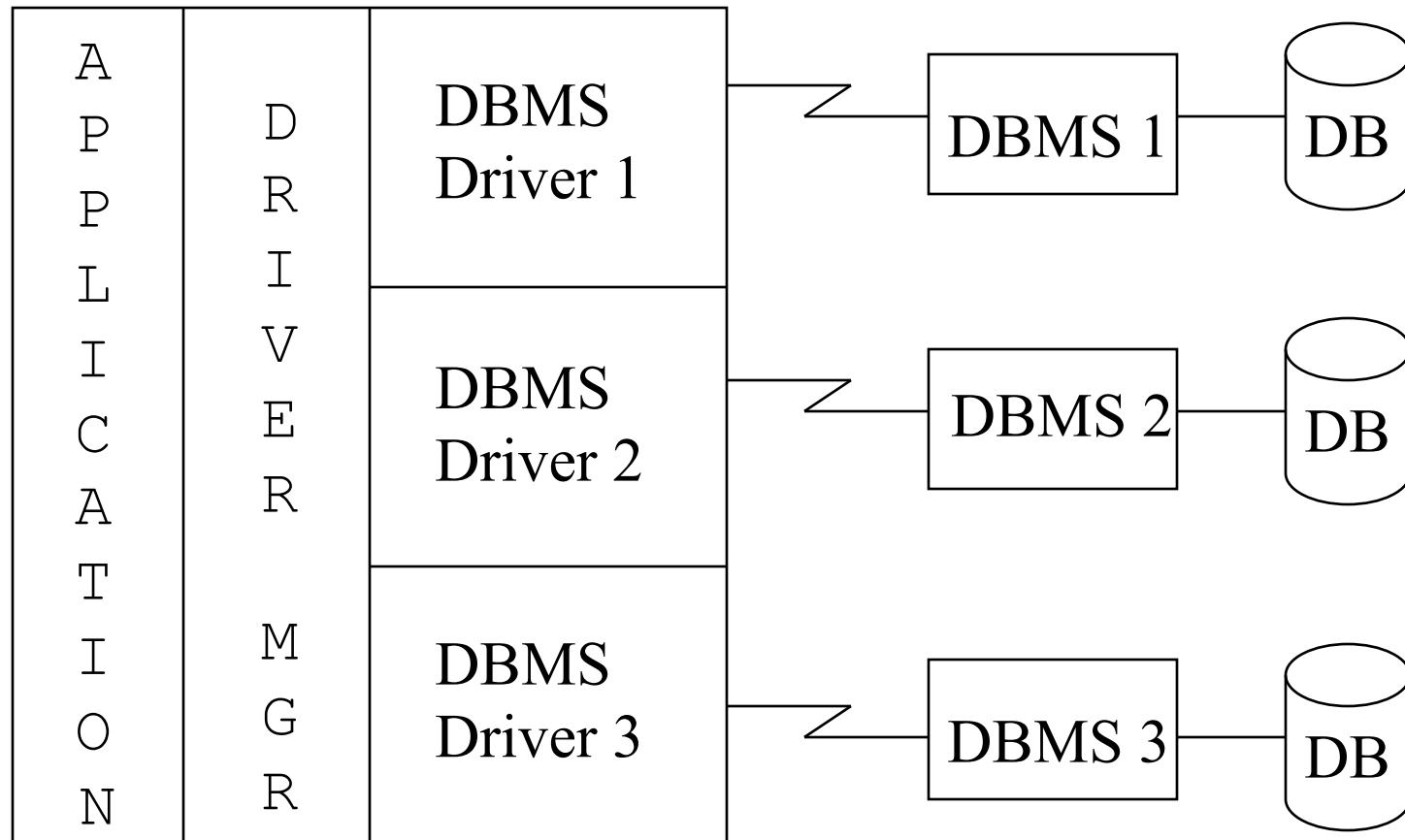
# Simple Database Application



# Multi-Databases



# Standard Access to DB



# Properties of a Relational Database

---

- Fundamental Properties
  - Tables, Indexes, Triggers, Stored Procedures, Transactions
- Data Definition (DDL)
  - Manage Database Objects
  - Create and delete tables, indexes, stored procs
- Queries (DML)
  - Data Manipulation Language
  - Read, write, or delete information between one or more related tables
  - SQL

# Queries (SQL)

---

- Insert

```
INSERT into Course(CourseID, Name, ProfessorID)  
values (162, 'Java101', 12);
```

- Update

```
UPDATE Course set NumStudents = 0;
```

- Delete

```
DELETE from Course where (ProfessorID = 100 );
```

- Select

```
SELECT Name from Course where CourseID = 162;
```

# Open Database Connectivity (ODBC) Standard

---

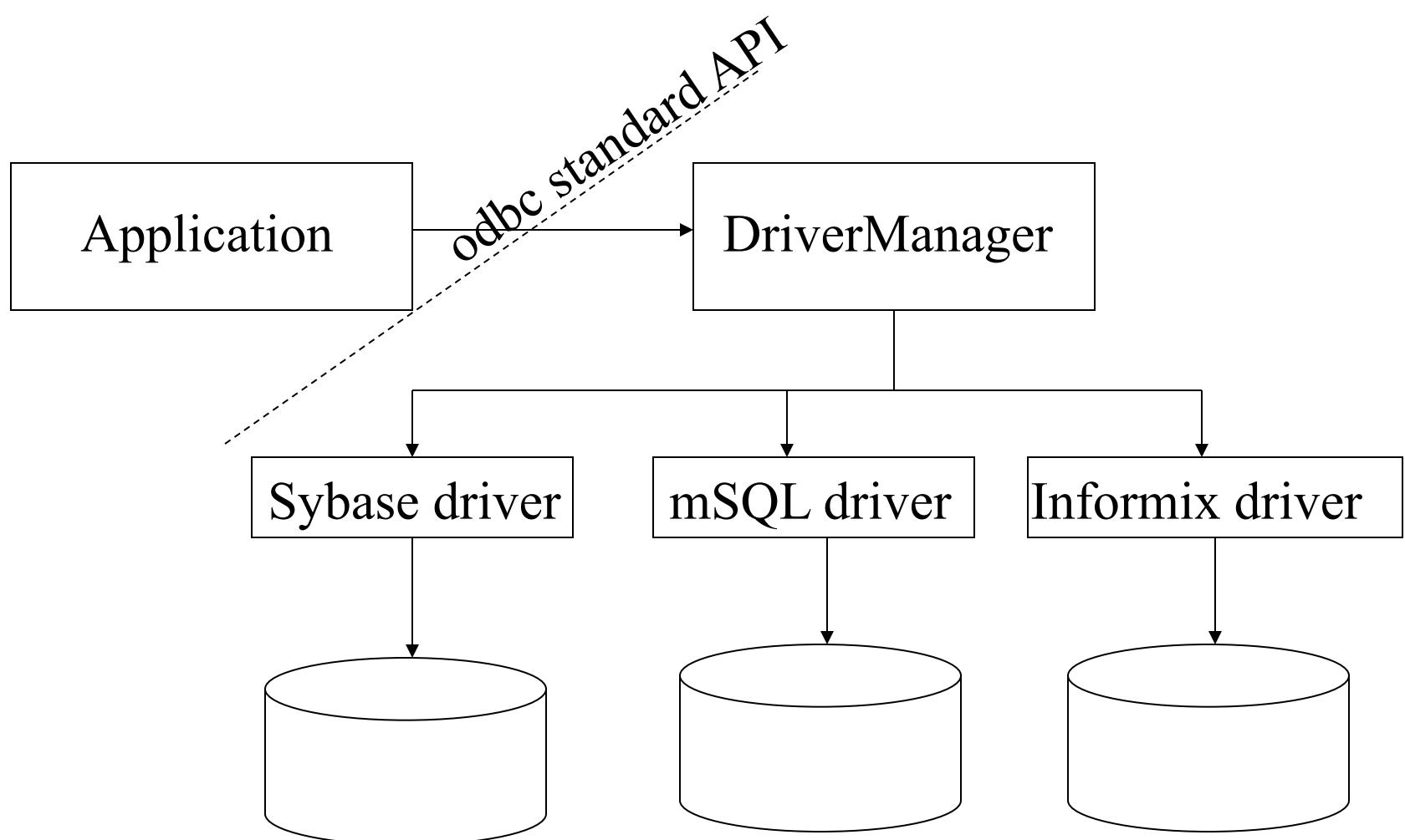
- ODBC standard is an interface by which application programs can access and process SQL databases in a DBMS-independent manner. It contains:
- A **Data Source** that is the database, its associated DBMS, operating system and network platform
- A **DBMS Driver** that is supplied by the DBMS vendor or independent software companies
- A **Driver Manager** that is supplied by the vendor of the O/S platform where the application is running

# ODBC Interface

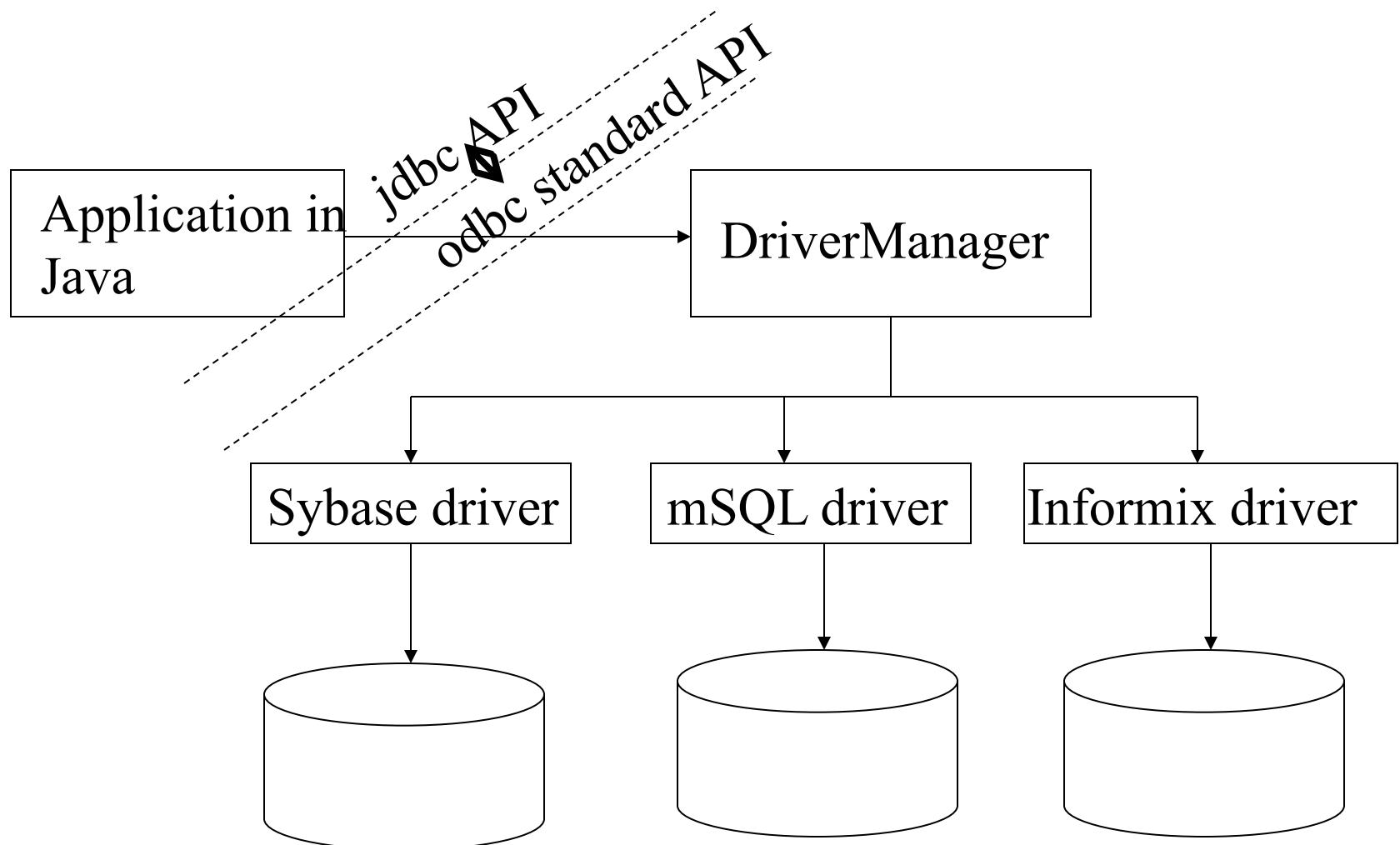
---

- It is a system independent interface to database environment that requires an ODBC driver to be provided for each database system from which you want to manipulate data.
- The database driver bridges the differences between your underlying system calls and the ODBC interface functionality.

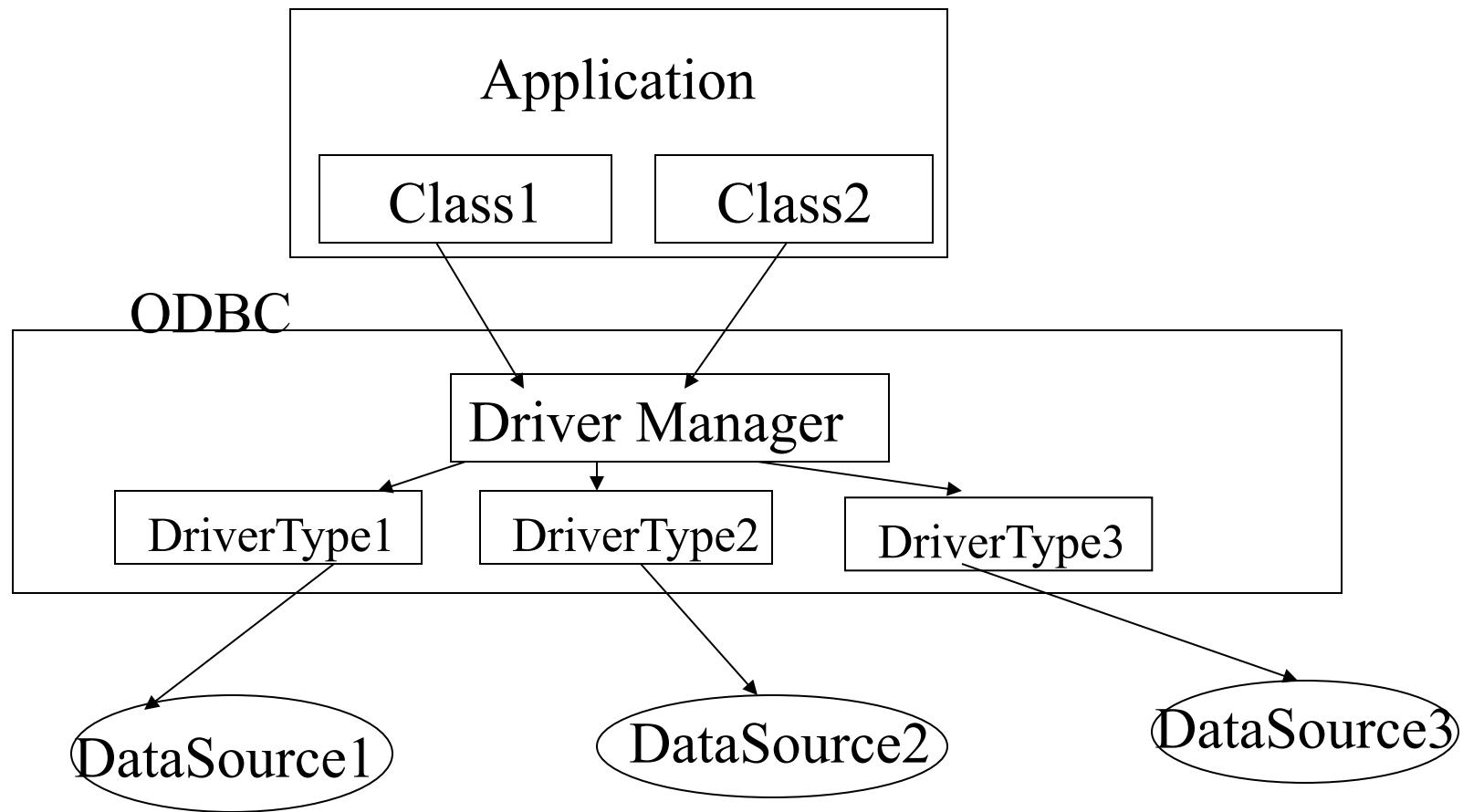
# An Example



# Application in Java



# ODBC Architecture



# What is JDBC?

---

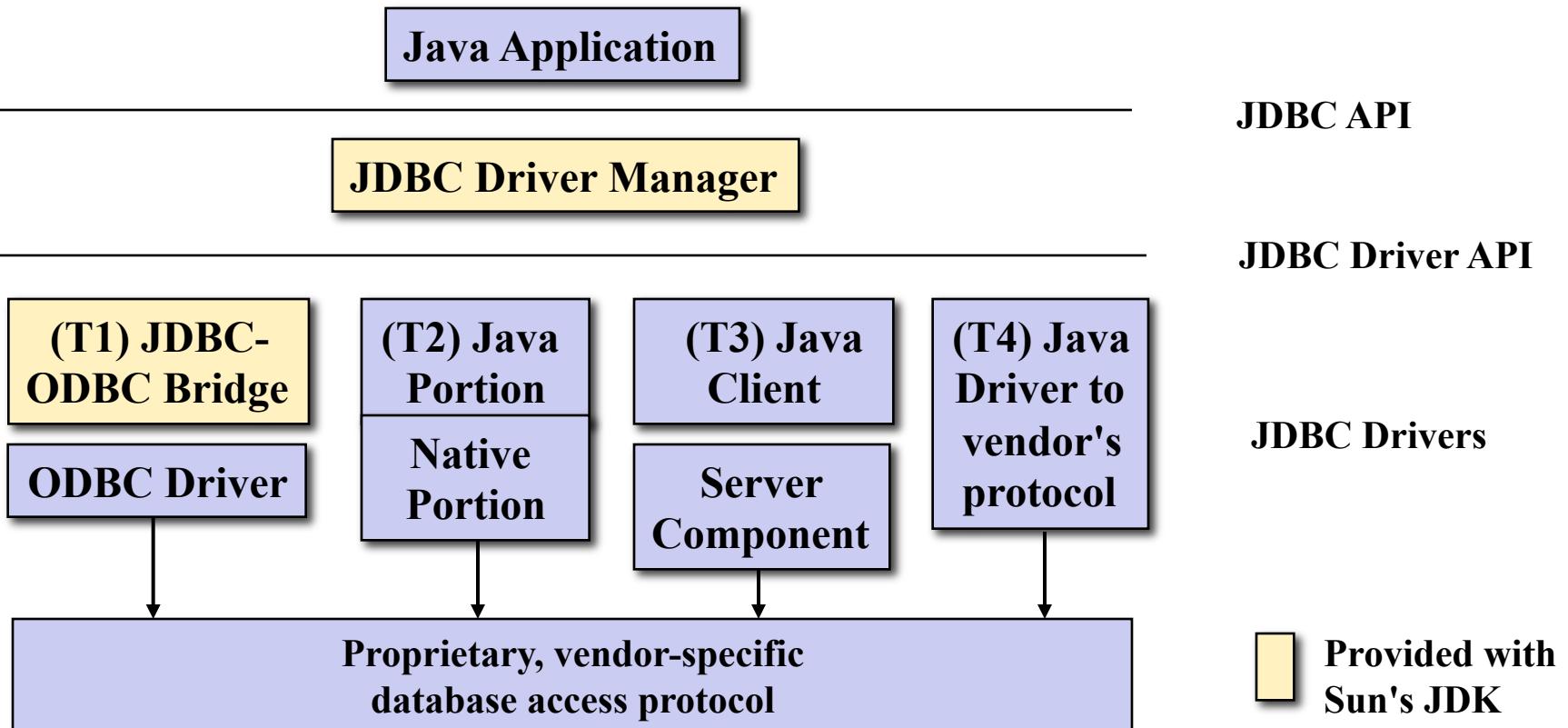
- A pure Java API for database communication similar to ODBC
  - JDBC and ODBC are based on the same standard: X/Open's SQL Call-Level Interface
- Created in 1995
- A set of classes that perform database transactions
  - Connect to relational databases
  - Send SQL commands
  - Process results
- JDBC 2 (1998) and JDBC 3.0 (2002) add more...
  - Additional features such as scrollable cursors, batch updates

# Benefits of JDBC

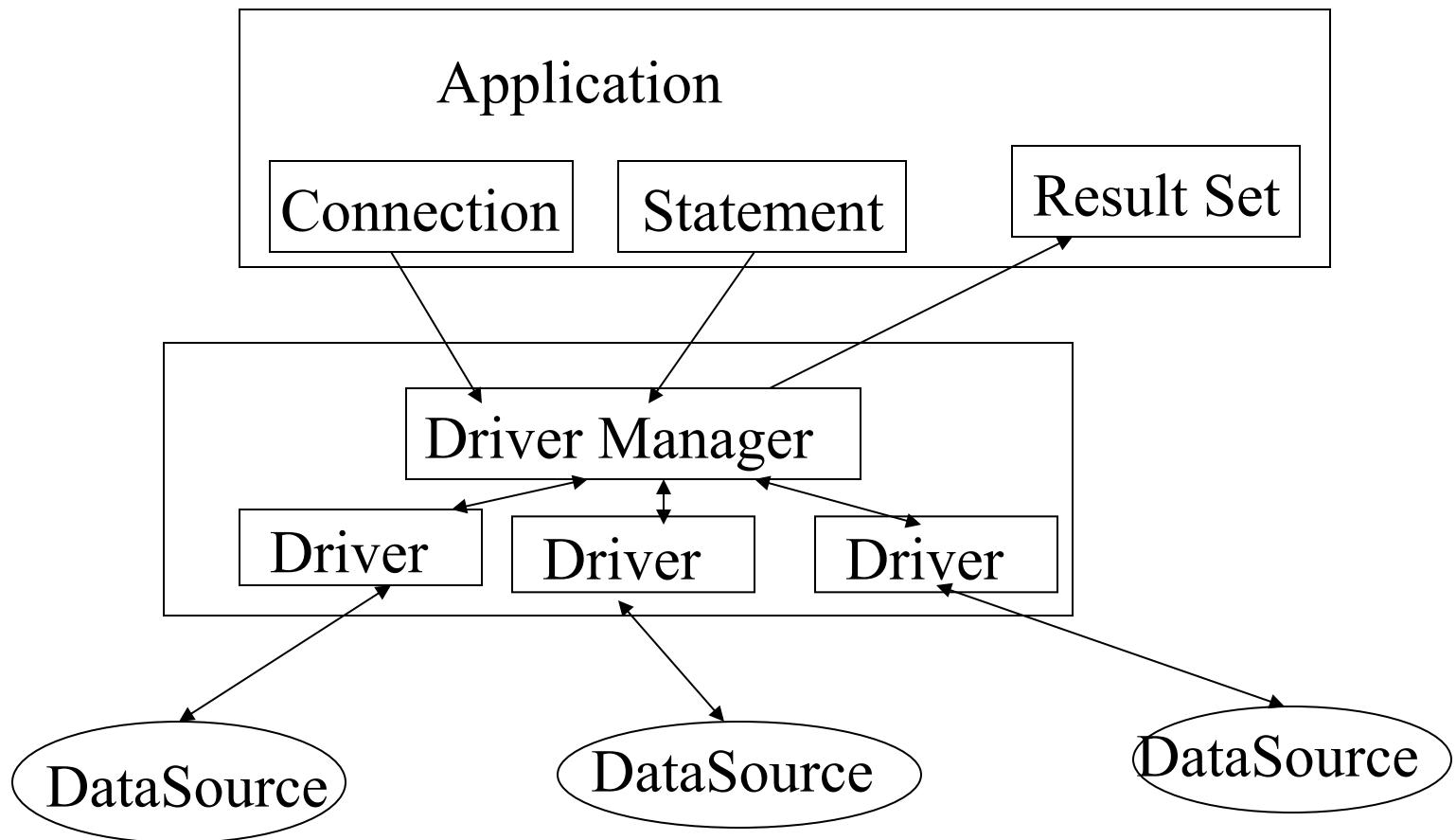
---

- No proprietary DB code
- Don't have to rely too much on single vendor
- Don't need to make DB vendor decision early
- Easier for DB vendors
  - Don't need to provide a query language, only implement API
  - *Only* low-level support

# JDBC Architecture



# JDBC Application Architecture



# **Java Support for ODBC : JDBC**

---

- When applications written in Java want to access data sources, they use classes and associated methods provided by Java DBC (JDBC) API.
- JDBC is specified an an “interface”.
- An interface in Java can have many “implementations”.
- So it provides a convenient way to realize many “drivers”

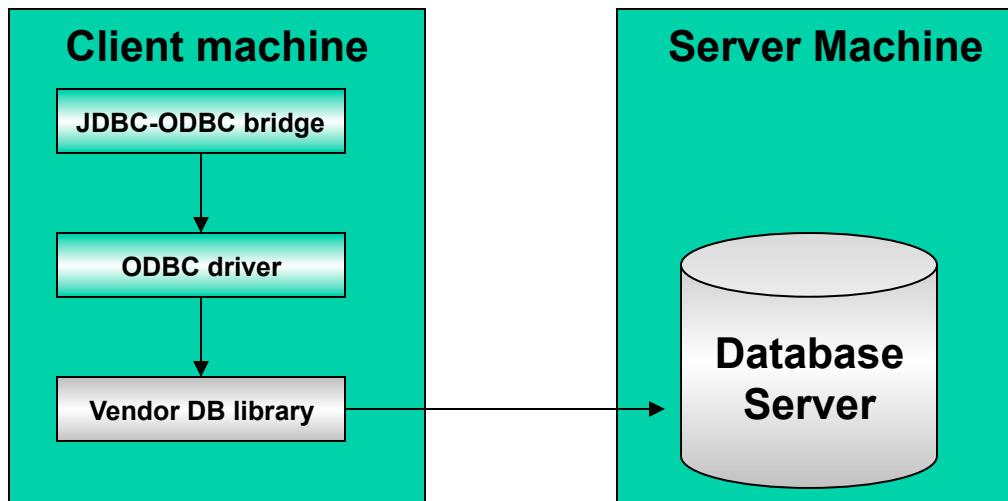
# JDBC Drivers Types

---

- Type 1: JDBC – ODBC bridge
  - Type 2: Native-API/partly Java driver
  - Type 3: Net-protocol/all-Java driver
  - Type 4: Native-protocol/all-Java driver
- 
- Types 1 & 2 rely on native binary modules
    - Platform specific
    - Cannot be used for applets
  - Types 3 & 4 are 100% Java

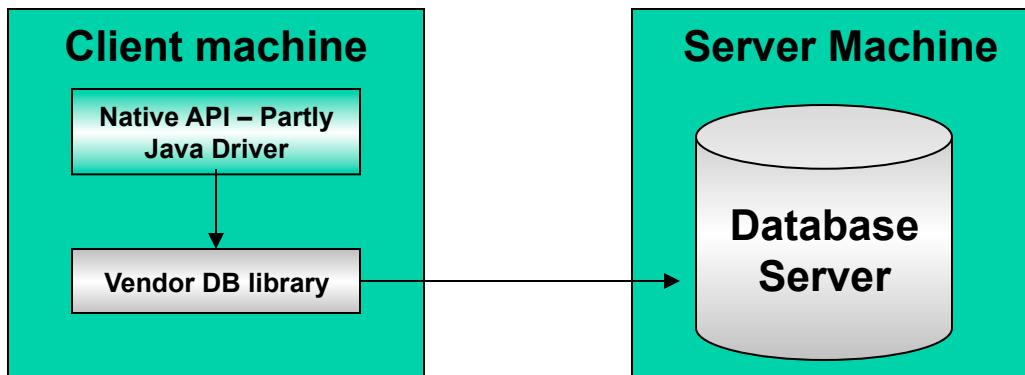
# Type 1 Driver

- JDBC – ODBC Bridge
  - Translates all JDBC calls into ODBC (Open Database Connectivity)
- Need to have ODBC client installed on the machine



# Type 2 Driver

- Native-API/Partly Java driver
  - Converts JDBC calls into db-specific calls
  - Communicates directly with the db server
  - Requires some binary code be present on the client machine.
  - Better performance than type 1 driver



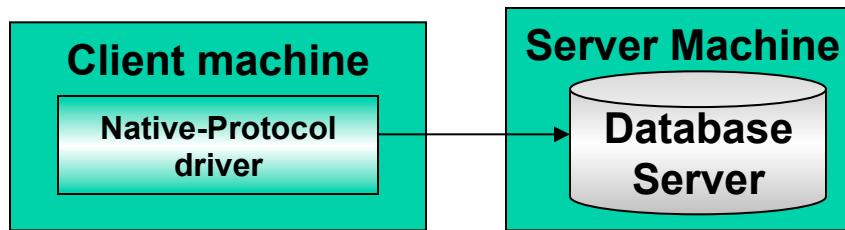
# Type 3 Driver

- Net-protocol – 100% Java driver
- Follows a three-tiered approach
  - JDBC database requests passed to the middle-tier server
  - Middle-tier server translates the request to the database-specific native-connectivity interface
    - May use a type 1 or type 2 JDBC driver
  - Request forwarded to the database server

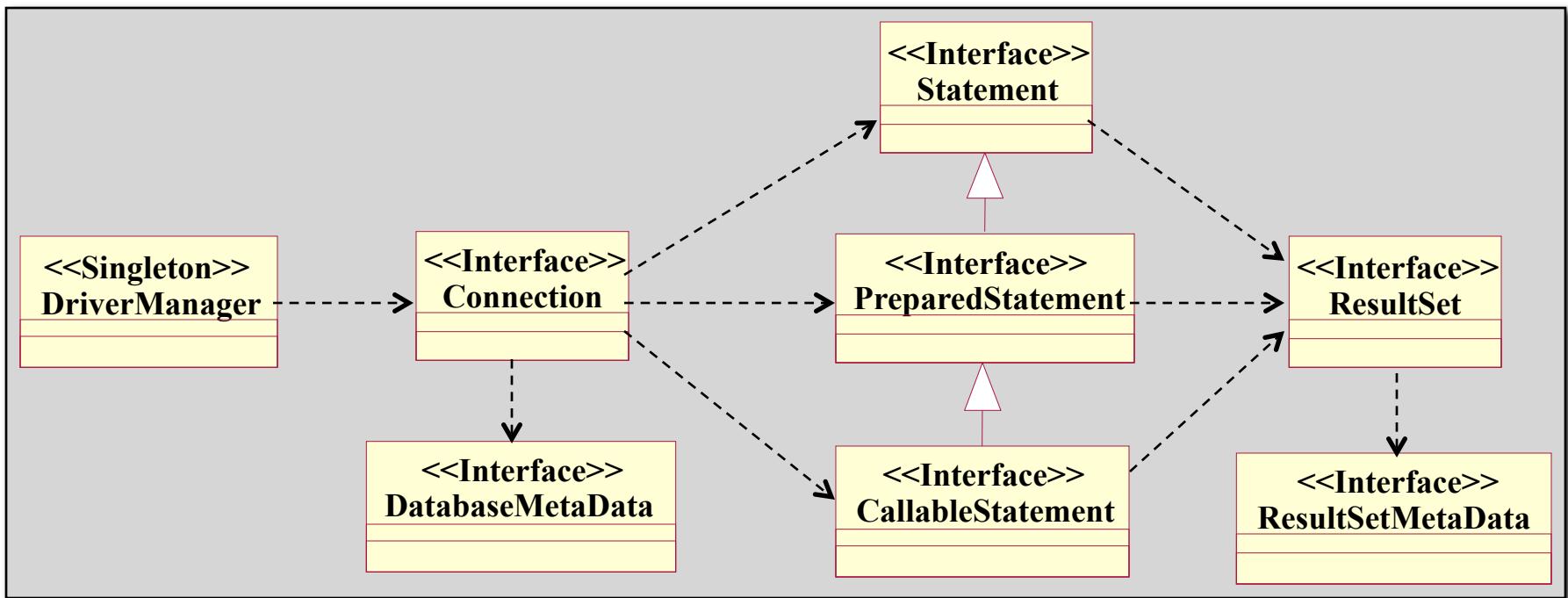


# Type 4 Driver

- Native protocol - 100% Java
- Converts JDBC calls into the vendor-specific DBMS protocol
- Client applications communicate directly with the database server
- Best performance
- Need a different driver for each database



# JDBC Classes



# JDBC Classes : java.sql package

---

- **Interfaces**
- **Classes**

- CallableStatement
- Connection
- DatabaseMetaData
- Driver
- PreparedStatement
- ResultSet
- ResultSetMetaData
- Statement

- Date
- DriverManager
- DriverPropertyInfo
- Time
- Timestamp
- Types

# Java Support for SQL

---

- Java supports embedded SQL.
- Also it provides an JDBC API as a standard way to connect to common relational databases.
- You need a JDBC:ODBC bridge for using the embedded SQL in Java.
- `Java.sql` package and an extensive exception hierarchy.
- We will examine incorporating this bridge using sample code.

# Data Source

---

- Local relational database; Ex: Oracle
- Remote relational database on a server; Ex: SQLserver
- On-line information service; Ex: Dow Jones, Customer database

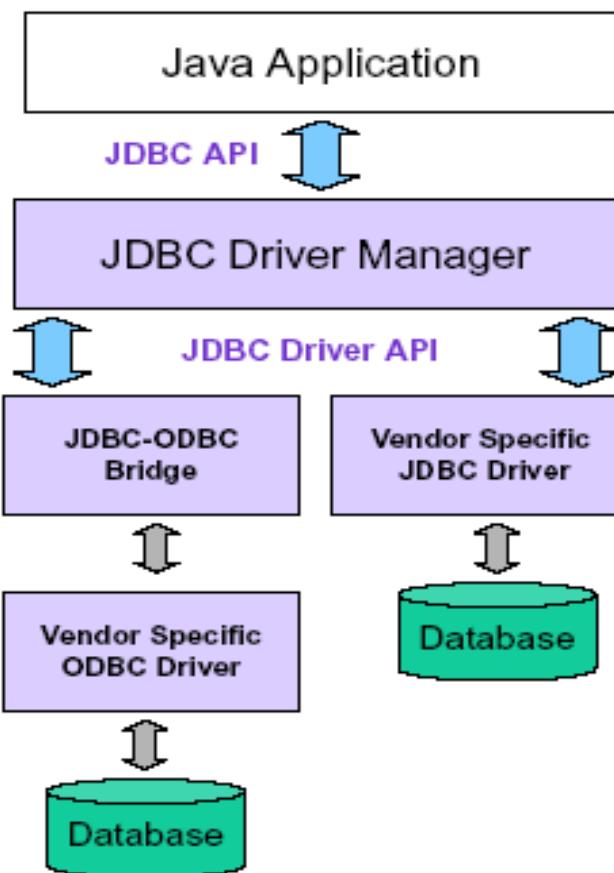
# Data Source and Driver

---

- Data source is the data base created using any of the common database applications available.
- Your system should have the driver for the database you will be using.
- For example your Windows system should have the MS Access Driver.
- There are a number of JDBC drivers available.

## JDBC consists of two parts:

- JDBC API, a purely Java-based API
- JDBC Driver Manager, which communicates with vendor-specific drivers that perform the real communication with the database.



# JDBC Components

---

- **Driver Manager:** Loads database drivers, and manages the connection between application & driver.
- **Driver:** Translates API calls to operations for a specific data source.
- **Connection:** A session between an application and a driver.
- **Statement:** A SQL statement to perform a query or an update operation.
- **Metadata:** Information about the returned data, driver and the database.
- **Result Set :** Logical set of columns and rows returned by executing a statement.

# JDBC Classes

---

- Java supports DB facilities by providing classes and interfaces for its components
- **DriverManager** class
- **Connection** interface (abstract class)
- **Statement** interface (to be instantiated with values from the actual SQL statement)
- **ResultSet** interface

# Driver Manager Class

---

- Provides static, “factory” methods for creating objects implementing the **connection** interface.
  - Factory methods create objects on demand
- when a connection is needed to a DB driver, DriverManager does it using its factory methods.

# Connection interface

---

- Connection class represents a session with a specific data source.
- Connection object establishes connection to a data source, allocates statement objects, which define and execute SQL statements.
- Connection can also get info (metadata) about the data source.

# Statement interface

---

- Statement interface is implemented by the connection object.
- Statement object provides the workspace for SQL query, executing it, and retrieving returned data.
- SELECT {what} FROM {table name} WHERE {criteria} ORDER BY {field}
- Queries are embedded as strings in a Statement object.
- Types: Statement, PreparedStatement, CallableStatement

# ResultSet interface

---

- Results are returned in the form of an object implementing the ResultSet interface.
- You may extract individual columns, rows or cell from the ResultSet using the metadata.

# Seven Basic Steps in Using JDBC

---

- Load the driver
- Define the Connection URL
- Establish the Connection
- Create a Statement object
- Execute a query
- Process the results
- Close the connection

# JDBC: Details of Process

---

## 1) Load the driver

```
try {  
    Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
}  
catch (ClassNotFoundException cnfe) {  
    System.out.println("Error loading driver: " + cnfe);  
}
```

# JDBC: Details of Process

---

## 2) Define the Connection URL

```
String host = "dbhost.yourcompany.com";  
String dbName = "someName";  
int port = 1521;  
String oracleURL = "jdbc:oracle:thin:@  
" + host +  
":" + port + ":" + dbName;
```

# JDBC: Details of Process

---

## 3) Establish the Connection

```
String username = "scott";
```

```
String password = "tiger";
```

```
Connection connection =
```

```
DriverManager.getConnection(oracleURL,  
                           username, password);
```

# JDBC: Details of Process

---

- Optionally, look up information about the database

```
DatabaseMetaData dbMetaData =  
    connection.getMetaData();  
  
String productName =  
    dbMetaData.getDatabaseProductName();  
  
System.out.println("Database: " + productName);  
  
String productVersion =  
    dbMetaData.getDatabaseProductVersion();  
  
System.out.println("Version: " + productVersion);
```

# JDBC: Details of Process

---

## 4) Create a Statement

```
Statement statement =  
    connection.createStatement();
```

## 5) Execute a Query

```
String query =  
"SELECT col1, col2, col3 FROM sometable";  
  
ResultSet resultSet =  
    statement.executeQuery(query);
```

# JDBC: Details of Process

---

## 6) Process the Result

```
while(resultSet.next()) {  
    System.out.println(resultSet.getString(1) + " " +  
        resultSet.getString(2) + " " + resultSet.getString(3));  
}
```

First column has index 1, not 0

- ResultSet provides various getXxx methods that take a column index or *column name* and returns the data
- You can also access result meta data (column names, etc.)

# JDBC: Details of Process

---

## 7) Close the Connection

```
connection.close();
```

Since opening a connection is expensive, postpone this step if additional database operations are expected

# Inserting values into Database

```
try{
    Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
    Connection con = DriverManager.getConnection("jdbc:db2:ntg", "db2admin", "db2admin");
    Statement st = con.createStatement();
    con.setAutoCommit(false);
    int i = st.executeUpdate("insert into userinfo values(001, 'admin', 'manager')");
    con.commit();
} catch(Exception e)
{
    System.out.println(e);
}finally{
    try{
        con.close();
    }catch(Exception e)
    {
        System.out.println(e);
    }
}
```

# Updating data

```
try{
    Class.forName("oracle.jdbc.driver.OracleDriver");
    con= DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl","scott","tiger");
    System.out.println("Connected");
    con.setAutoCommit(false);
    Statement st = con.createStatement();
    st.executeUpdate("update userInfo set password= 'admin' where username= 'manager'");
    con.commit();
}catch(Exception e)
{
    System.out.println(e);
}finally{
    try{
        con.close();
    }catch(Exception e)
    {
        System.out.println(e);
    }
}
```

# ResultSet

---

- Useful Methods
  - All methods can throw SQLException.
  - close()
    - Release JDBC and Database resources
      - The resultset is automatically closed when associated statement object executes a new query
  - getMetaDataObject()
    - Returns a ResultSetMetaData object containing information about columns in the ResultSet.
  - next()
    - Attempts to move to the next row in the ResultSet
      - If successful true is returned else false.
      - The first call to the next position, the cursor is at first row
      - Calling next clears the SQLWarning chain

# ResultSet

---

- getWarnings()
  - Returns the first SQLWarning or null if no warnings occurred.
- findColumn()
  - Returns the corresponding integer value corresponding to specified column name.
  - Column number in the resultset do not necessarily map to the same column no. in database.
- getXXX()
  - We use the getXXX method of the appropriate type to retrieve the value in each column.
  - Returns null, if the value is SQL NULL
  - Legal getXXX types:
    - Double      Flooat      String      Int      Date
    - Short      Long      Time      Object
- wasNull()
  - Used to check if the last getXXX read was a SQL NULL

# Result Set

---

```
try{  
    Class.forName("oracle.jdbc.driver.OracleDriver");  
    String oracleURL = "jdbc:oracle:thin:@localhost:1521:orcl";  
    con = DriverManager.getConnection(oracleURL,"scott","tiger");  
    System.out.println("connected");  
    Statement st = con.createStatement();  
    ResultSet rs = st.executeQuery("select *from userinfo");  
    while(rs.next()){  
        System.out.println(rs.getString(1));  
        System.out.println(rs.getInt(2));  
    } catch(ClassNotFoundException cnfe){  
        System.out.println("Can not load oracle Driver");  
    }catch(SQLException e){  
    }  
}
```

# Other ResultSet features

---

- Move the cursor (pointing to the current row) backwards and forwards, or position it anywhere within the ResultSet
- Update/delete the database row corresponding to the current result row
  - Analogous to the view update problem
- Insert a row into the database
  - Analogous to the view update problem

# Using MetaData

---

- Idea
  - From a ResultSet( the return type) of executeQuery), derive a RssultSetMetaData object.
  - Use that object to look up the number,names, and types of columns.
- ResultSetMetaData answers the follwing questions
  - How many columns are in result set?
  - What is the name of the given column?
  - Are the column name case sensitive?
  - What is the data type of specific column?
  - What is the maximum chracter size of column?

# Useful Metadata methods

---

- `getColumnCount`
  - Returns the number of columns in this ResultSet object.
- `getColumnDisplaySize`
  - Indicates the designated column's normal maximum width in characters.
- `getColumnLabel`
  - Gets the designated column's suggested title for use in printouts and displays.
- ◆ `getColumnName`
  - ◆ Get the designated column's name.

# Useful Metadata methods

---

- **isReadOnly**

- public boolean isReadOnly(int column)

- **isWritable**

- Indicates whether it is possible for a write on the designated column to succeed.

# Statements

---

- Through Statement object SQL statements are send to the data base.
- Three types of statement objects are available.
  - Statement
    - For executing simple SSQl statement.
  - PreparedStatement
    - For executing precompiled SQL statements passing in parameter.
  - CallableStatement
    - For executing database stored procedure.

# Useful statement methods

---

- **executeQuery**
  - Executes the given SQL statement, which returns a single ResultSet object.
    - a ResultSet object that contains the data produced by the given query; never null
- **executeUpdate**
  - Executes the given SQL statement, which may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement.
- **getMaxRows**
  - Retrieves the maximum number of rows that a ResultSet object produced by this Statement object can contain. If this limit is exceeded, the excess rows are silently dropped.

# Useful statement methods

---

## •**setMaxRows**

- Sets the limit for the maximum number of rows that any ResultSet object can contain to the given number. If the limit is exceeded, the excess rows are silently dropped.

## •**getQueryTimeout**

- Retrieves the number of seconds the driver will wait for a Statement object to execute. If the limit is exceeded, a SQLException is thrown.

# Prepared statements: motivation

```
Statement stmt = con.createStatement();
for (int age=0; age<100; age+=10) {
    ResultSet rs = stmt.executeQuery
        (.SELECT AVG(GPA) FROM Student. +
         . WHERE age >= . + age + . AND age < . + (age+10));
    // Work on the results:
}
```

- ❖ Every time an SQL string is sent to the DBMS, the DBMS must perform parsing, semantic analysis, optimization, compilation, and then finally execution
- ❖ These costs are incurred 10 times in the above example, even though all strings are essentially the same query (with different parameter values)

# Methods

---

- `setXxx`
  - Set indicated parameter(?) in SQL statement to value.
- `clearParameter`
  - Clears the current parameter values immediately.
- `setString`
  - `public void setString(int parameterIndex, String x)`
    - Sets the designated parameter to the given Java String value.

# Prepared statements: syntax

```
// Prepare the statement, using ? as placeholders for actual parameters:  
PreparedStatement stmt = con.prepareStatement  
("SELECT AVG(GPA) FROM Student WHERE age >= ? AND age < ?");  
for (int age=0; age<100; age+=10) {  
    // Set actual parameter values:  
    stmt.setInt(1, age);  
    stmt.setInt(2, age+10);  
    ResultSet rs = stmt.executeQuery();  
    // Work on the results:  
    .  
}
```

- ❖ The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it prepares the statement
- ❖ At execution time, the DBMS only needs to check parameter types and validate the compiled execution plan

# Callable statement

---

- The interface used to execute SQL **stored procedures**.
- Adavantage
  - Stored procedures execute much faster than dynamic SQL.
  - The programmer need to know only about the input and output parameters for the stored procedure, not the table structure or internal details of the stored procedure.

# CallableStatement

---

- Stored procedure syntax

- Procedure with no parameters

```
{call procedure_name}
```

- Procedure with input parameters

```
{call procedure_name(?, ?, .....)}
```

- Procedure with output parameters

```
{ ? = call procedure_name(?, ?, .....)}
```

- CallableStatement cstatement = connection.prepareCall("{call procedure(?, ?)}");

# CallableStatement

---

- Output parameters
  - Register the jdbc type of each parameter through registerOutParameter before calling execute.

Statement.registerOutParameter(n,Types.Float);

Use getXxx to access stored procedure return value.

# Unit Summary

---

- JDBC
- The architecture of Java Database Connectivity
- JDBC model
- JDBC-ODBC bridge
- JDBC programming
- Result set processing
- JDBC classes

*Welcome to*

# NetWork Programming

# Unit Objective

---

- After completing this unit you should be able to
  - Networking
  - Networking basics
  - Working with URLs
    - Creating and Parsing URLs
    - Reading from URLs
    - Connecting to a URL
    - Reading from and writing to URL connection.
  - Sockets
    - Reading from and writing to sockets
    - Writing server side sockets

# **URL and URL Connection**

# URL Class

---

- Provides high-level access to network data
- Abstracts the notion of a connection
- Constructor opens network connection
  - To resource named by URL

# The Class URL

---

- The class URL is used for parsing URLs
- Constructing URLs:
  - URL w3c1 = new URL("http://www.w3.org/TR/");
  - URL w3c2 = new URL("http","www.w3.org",80,"TR/");
  - URL w3c3 = new URL(w3c2, "xhtml1/");
- If the string is not an absolute URL, then it is considered relative to the URL

# URL Constructors

---

- URL( fullURL )

- URL( "http://www.cs.umd.edu/class/index.html" )

- URL( baseURL, relativeURL )

- URL base = new URL("http://www.cs.umd.edu/");
  - URL class = new URL( base, "/class/index.html" );

- URL( protocol, baseURL, relativeURL )

- URL( "http", www.cs.umd.edu, "/class/index.html" )

- URL( protocol, baseURL, port, relativeURL )

- URL( "http", www.cs.umd.edu, 80, "/class/index.html" )

# URL Methods

---

- The following methods of URL can be used for parsing URLs
  - getProtocol( )
  - getHost( )
  - getPort( )
  - getFile( )
  - getContent( )
  - openStream()
  - openConnection()

# Parsing a URL

---

```
import java.net.*;
import java.io.*;
public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL aURL = new URL("http://java.sun.com:80/docs/books/"
                            + "tutorial/index.html#DOWNLOADING");
        System.out.println("protocol = " + aURL.getProtocol()); System.out.println("host = " +
        aURL.getHost()); System.out.println("filename = " + aURL.getFile());
        System.out.println("port = " + aURL.getPort()); System.out.println("ref = " +
        aURL.getRef()); } }
```

Here's the output displayed by the program:

```
protocol = http
host = java.sun.com
filename = /docs/books/tutorial/index.html
port = 80
ref = DOWNLOADING
```

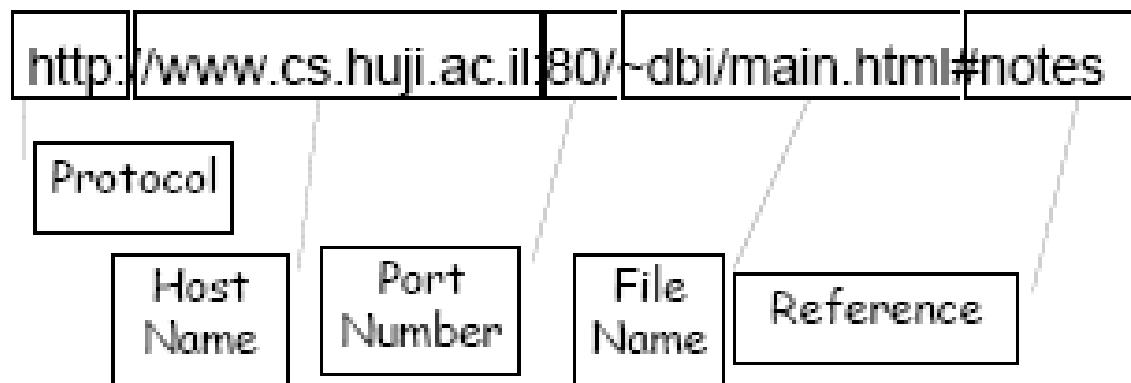
# URL Connection Classes

---

- High level description of network service
- Access resource named by URL
- Can define own protocols
- Examples
  - `URLConnection`      ⇒ Reads resource
  - `HttpURLConnection`   ⇒ Handles web page
  - `JarURLConnection`   ⇒ Manipulates Java Archives
  - `URLClassLoader`      ⇒ Loads class file into JVM

# Working with URLs

- **URL** (Uniform Resource Locator):
  - a reference (an address) to a resource on the Internet



# The Class URL

---

- The class URL is used for parsing URLs
- Constructing URLs:
  - URL w3c1 = new URL("http://www.w3.org/TR/");
  - URL w3c2 = new URL("http","www.w3.org",80,"TR/");
  - URL w3c3 = new URL(w3c2, "xhtml1/");
- If the string is not an absolute URL, then it is considered relative to the URL

# The class URLConnection

---

- To establish the actual resource, we can use the object `URLConnection` obtained by `url.openConnection()`
- If the protocol of the URL is HTTP, the returned object is of class `HttpURLConnection`
- This class encapsulates all socket management and HTTP directions required to obtain the resource

# Connecting to a URL

---

```
try {  
    URL yahoo = new URL("http://www.yahoo.com/");  
    URLConnection yahooConnection = yahoo.openConnection();  
    } catch (MalformedURLException e) {          // new URL() failed  
        . . .  
    } catch (IOException e) {      // openConnection()failed  
        . . .  
    }  
}
```

# Reading Directly from a URL

---

```
import java.net.*;
import java.io.*;

public class URLReader {

    public static void main(String[] args) throws Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        BufferedReader in = new BufferedReader( new
            InputStreamReader( yahoo.openStream())); String
        inputLine;
        while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
        in.close();
    }
}
```

# Reading from a URLConnection

---

- This program explicitly opens a connection to a URL and gets an input stream from the connection.

```
import java.net.*;  
import java.io.*;  
  
public class URLConnectionReader {  
    public static void main(String[] args) throws Exception {  
        URL yahoo = new URL("http://www.yahoo.com/");  
        URLConnection yc = yahoo.openConnection();  
        BufferedReader in = new BufferedReader( new  
            InputStreamReader(yc.getInputStream()));  
        String inputLine;  
        while ((inputLine = in.readLine()) != null)  
            System.out.println(inputLine);  
        in.close();  
    }  
}
```

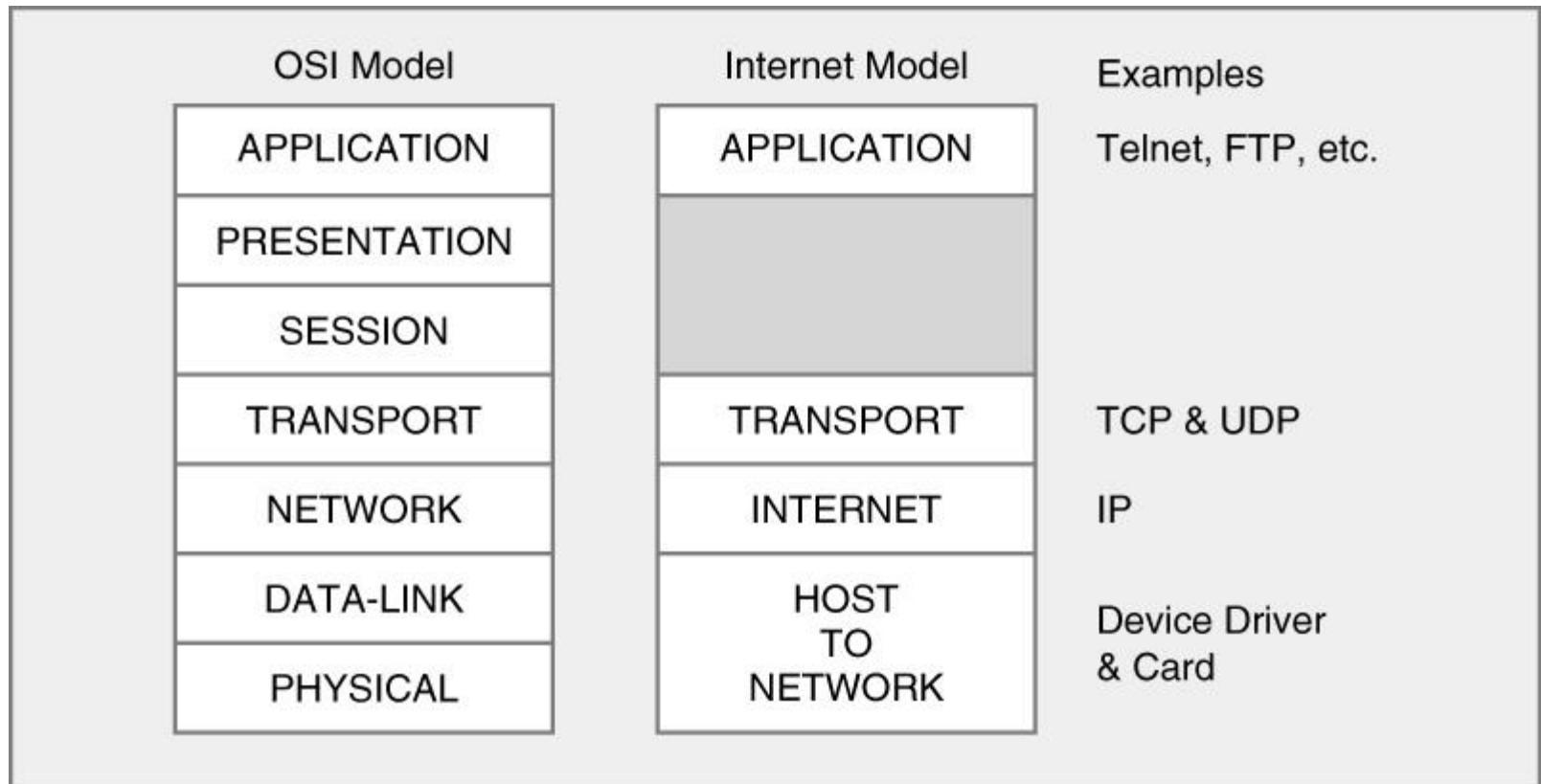
# **Network Programming**

# Overview

---

- Networking
  - Background
  - Concepts
  - Network applications
  - Java's networking API  
(Application Program Interface)

# Internet Design



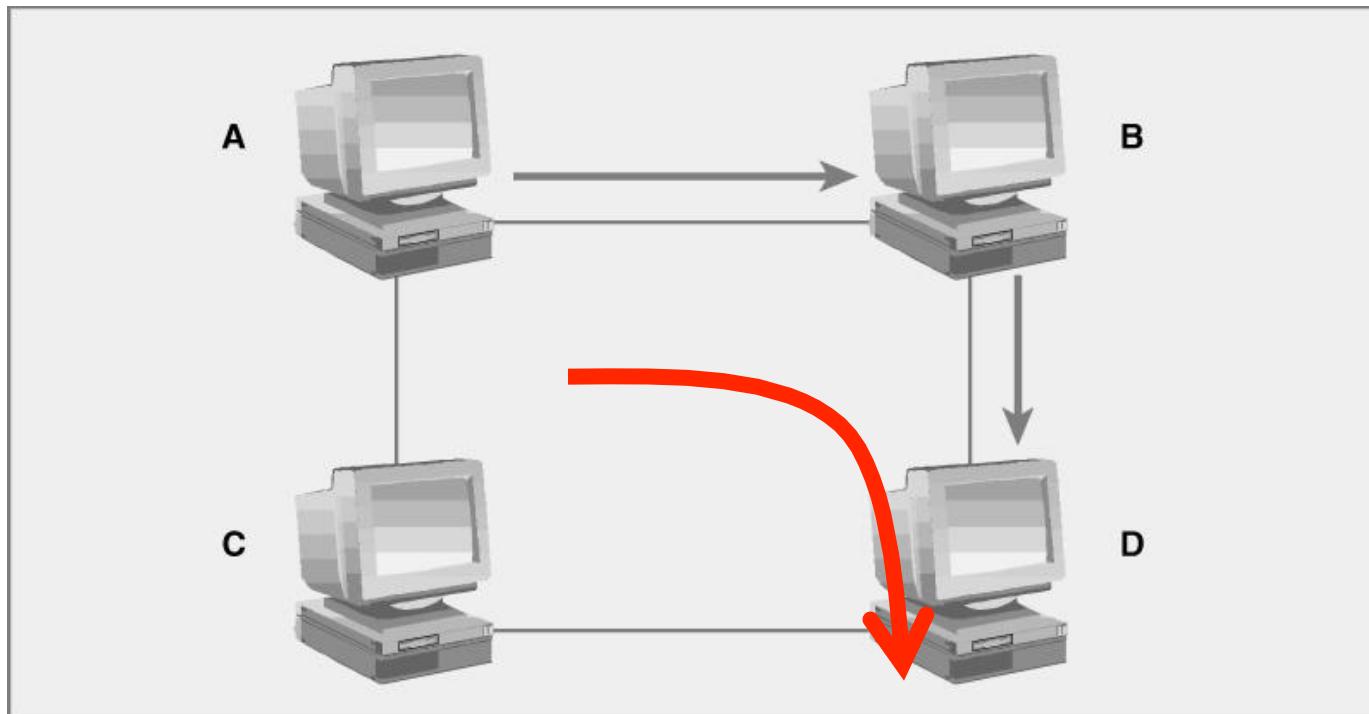
# Internet

---

- Network layer
  - Internet Protocol (**IP**)
- Transport layer
  - User Datagram Protocol (**UDP**)
  - Transmission Control Protocol (**TCP**)

# Internet Protocol (IP)

- Packet oriented
- Packets **routed** between computers
- Unreliable



# **TCP (Transmission-Control Protocol)**

---

- Enables symmetric byte-stream transmission between two endpoints (applications)
  - Reliable communication channel
  - TCP performs these tasks:
    - connection establishment by handshake (relatively slow)
    - division to numbered packets (transferred by IP)
    - error correction of packets (checksum)
    - acknowledgement and retransmission of packets
    - connection termination by handshake

# Transmission Control Protocol (**TCP**)

---

- Reliable but slower
- Application can treat as reliable connection
  - Despite unreliability of underlying IP (network)
- Examples
  - ftp (file transfer)
  - telnet (remote terminal)
  - http (web)

# UDP (User Datagram Protocol)

---

- Enables direct datagram (packet) transmission from one endpoint to another
- No reliability (*except for data correction*)
  - sender does not wait for acknowledgements
  - arrival order is not guaranteed
  - arrival is not guaranteed
- Used when speed is essential, even in cost of reliability
  - e.g., streaming media, games, Internet telephony, etc.

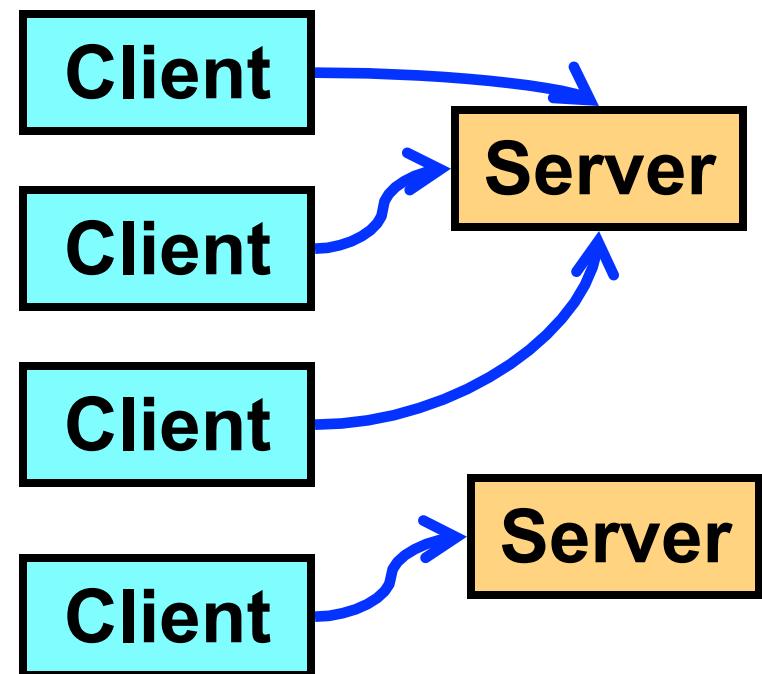
# User Datagram Protocol (**UDP**)

---

- Packet oriented
- Message split into datagrams
- Send datagrams as packets on network layer
- Unreliable but fast
- Application must deal with lost packets
- Examples
  - Ping
  - Streaming multimedia
  - Online games

# Client / Server Model

- Relationship between two computer programs
- Client
  - Initiates communication
  - Requests services
- Server
  - Receives communication
  - Provides services



# Client Programming

---

- Basic steps
  1. Determine server location – IP address & port
  2. Open network connection to server
  3. Write data to server (request)
  4. Read data from server (response)
  5. Close network connection
  6. Stop client

# Server Programming

---

- Basic steps
  1. Determine server location - port (& IP address)
  2. Create server to listen for connections
  3. Open network connection to client
  4. Read data from client (request)
  5. Write data to client (response)
  6. Close network connection to client
  7. Stop server

# Server Programming

---

- Can support multiple connections / clients
- Loop
  - Handles multiple connections in order
- Multithreading
  - Allows multiple simultaneous connections

# Networking in Java

---

- Packages
  - java.net ⇒ Networking
  - java.io ⇒ I/O streams & utilities
  - java.rmi ⇒ Remote Method Invocation
  - java.security ⇒ Security policies
  - java.lang ⇒ Threading classes
- Support at multiple levels
  - Data transport ⇒ Socket classes
  - Network services ⇒ URL classes
  - Utilities & security

# Java Networking API

---

- Application Program Interface
  - Set of routines, protocols, tools
  - For building software applications
- Java networking API
  - Helps build network applications
  - Interfaces to sockets, network resources
  - Code implementing useful functionality
  - Includes classes for
    - Sockets
    - URLs

# Java Networking Classes

---

- IP addresses
  - InetAddress
- Packets
  - DatagramPacket
- Sockets
  - Socket
  - ServerSocket
  - DatagramSocket
- URLs
  - URL

# InetAddress Class

---

- Represents an IP address
- Can convert domain name to IP address
  - Performs DNS lookup
- Getting an InetAddress object
  - getLocalHost()
  - getByName(String host)
  - getByAddress(byte[] addr)

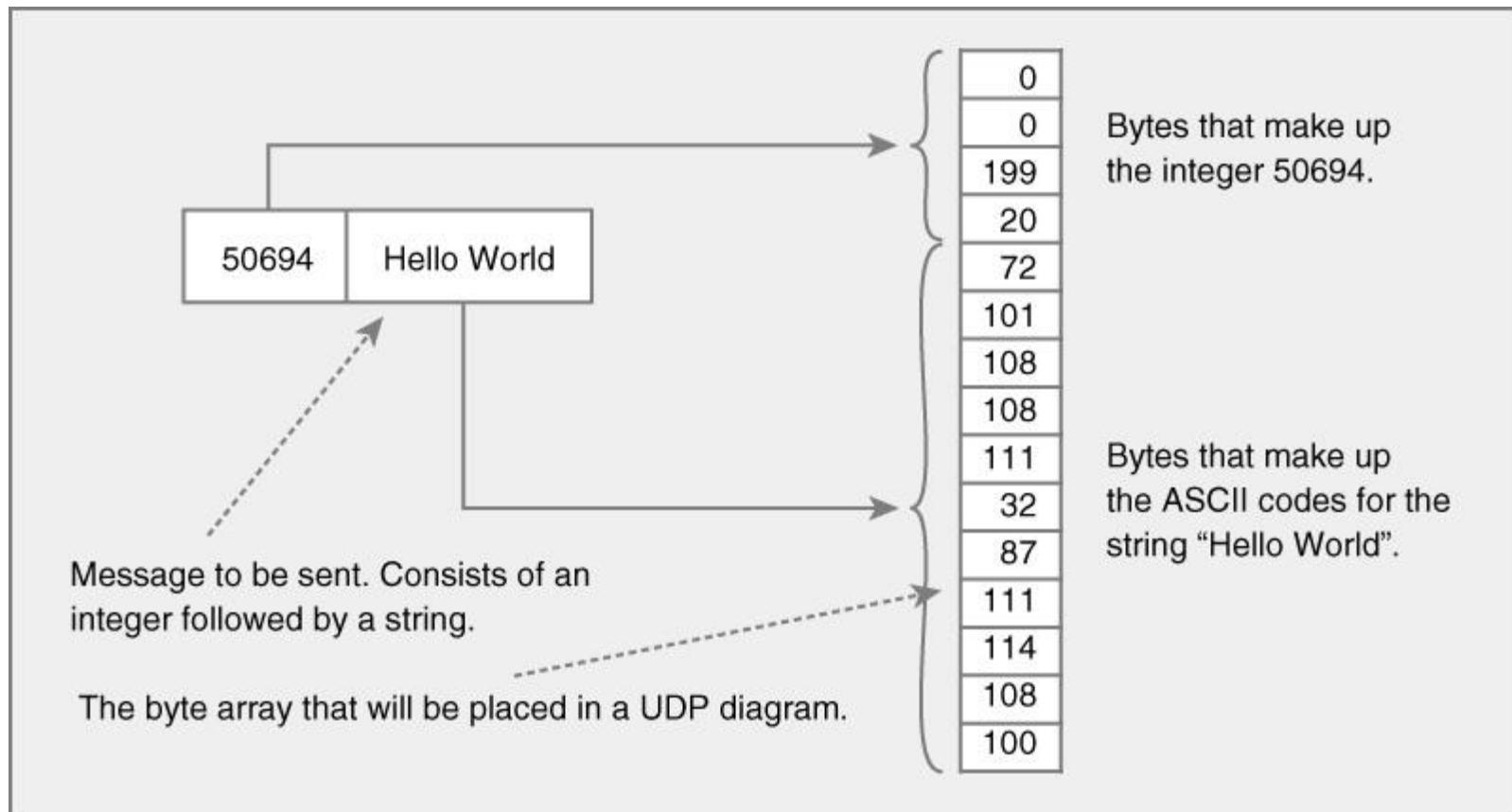
# DatagramPacket Class

---

- Each packet contains
  - InetAddress
  - Port of destination
  - Data

# DatagramPacket Class

- Data in packet represented as byte array



# DatagramPacket Methods

---

- getAddress()
- getData()
- getLength()
- getPort()
- setAddress()
- setData()
- setLength()
- setPort()

# Clients

---

- Code for handling client side is completely standard, the code examples in books on TCP/IP programming in C/C++ can easily be adapted for Java.
  - client
    - *build address structure for server*
    - *get socket*
    - *connect to server*
    - *while not finished*
    - *get command from user*
    - *write to socket*
    - *read reply*
    - *show reply to user*

# Servers

---

- Server may have to deal with many clients
  - Concurrency strategies:
    - Serial:
      - deal with clients one at a time, handling all requests until client disconnects; then start handling next client.
      - OS helps by maintaining a queue of clients who wish to connect;
    - Forking
      - Start a new process for each client – just as was described for WebServer (this strategy rarely used with Java servers)
    - Threaded
      - Each client gets a thread to serve them

# Server provides a service

---

- Typically, a server is a bit like an instance of a class that has a public interface advertising a few operations it can perform for a client.
  - Example 1: HTTP server
    - GET, POST, OPTIONS, ...
  - Example 2: FTP server
    - List directory, change directory, get file, put file, ...
- Sometimes do have a service with only one operation – echo, ping, ...

# Ports

---

- A computer may have several applications that communicate with applications on remote computers through the same physical connection to the network
  - When receiving a packet, how can the computer tell which application is the destination?
  - Solution: each channel endpoint is assigned a unique port that is known to both the computer and the other end point

# Ports (cont)

---

- Thus, an endpoint application on the Internet is identified by
  - A host name
  - 32 bits IP-address
  - A 16 bits port

# Known Ports

---

- Some known ports are

- 20, 21: FTP

- 23: TELNET

- 25: SMTP

- 110: POP3

- 80: HTTP

- 19: NNTP

# Client-Server Model

---

- A common paradigm for distributed applications
- Asymmetry in connection establishment:
  - Server waits for client requests (daemon) at a well known address (IP+port)
  - Connection is established upon client request
- Once the connection is made, it can be either symmetric (TELNET) or asymmetric (HTTP)
- For example: Web servers and browsers

# Accepting Connections

---

- Usually, the accept() method is executed within an infinite loop
  - i.e., `while(true){...}`
- Whenever accept() returns, a new *thread* is launched to handle that interaction
- Hence, the server can handle several requests concurrently

# Timeout

---

- You can set timeout values to these blocking methods:
  - `read()` of `Socket`
  - - `accept()` of `ServerSocket`
- Use the method `setSoTimeout(milliseconds)`
- If timeout is reached before the method returns,  
`java.net.SocketTimeoutException` is thrown

# Reading HTTP Messages

---

- Several ways to interpret the bytes of the body
  - Binary: images, compressed files, class files, ...
  - - Text: ASCII, Latin-1, UTF-8, ...
- Commonly, applications parse the headers of the message, and process the body according to the information supplied by the headers
  - E.g., Content-Type, Content-Encoding, Transfer-Encoding

# Sockets

---

- A socket is a construct that represents one *end-point* of a *two-way* communication channel between two programs running on the network
- Using sockets, the OS provides processes a file-like access to the channel
  - i.e., sockets are allocated a file descriptor, and processes can access (read/write) the socket by specifying that descriptor
- A specific socket is identified by the machine's IP and a port within that machine

# Sockets (cont)

---

- A socket stores the IP and port number of the other endpoint computer of the channel
- When writing to a socket, the written bytes are sent to the other computer and port (e.g., over TCP/IP)
  - That is, remote IP and port are attached to the packets
- When OS receives packets on the network, it uses their destination port to decide which socket should get the received bytes

# Socket Classes

---

- Provides interface to TCP, UDP sockets
- Socket
  - TCP client sockets
- ServerSocket
  - TCP server sockets
- DatagramSocket
  - UDP sockets (server or client)

# Socket Class

---

- Creates socket for client
- Constructor connects to
  - Machine name or IP address
  - Port number
- Transfer data via **streams**
  - Similar to standard Java I/O streams

# Socket Methods

---

- `getInputStream()`
- `getOutputStream()`
- `close()`
- `getInetAddress()`
- `getPort()`
- `getLocalPort()`

# ServerSocket Class

---

- Create socket on server
- Constructor specifies local port
  - Server listens to port
- Usage
  - Begin waiting after invoking accept()
  - Listen for connection (from client socket)
  - Returns **Socket** for connection

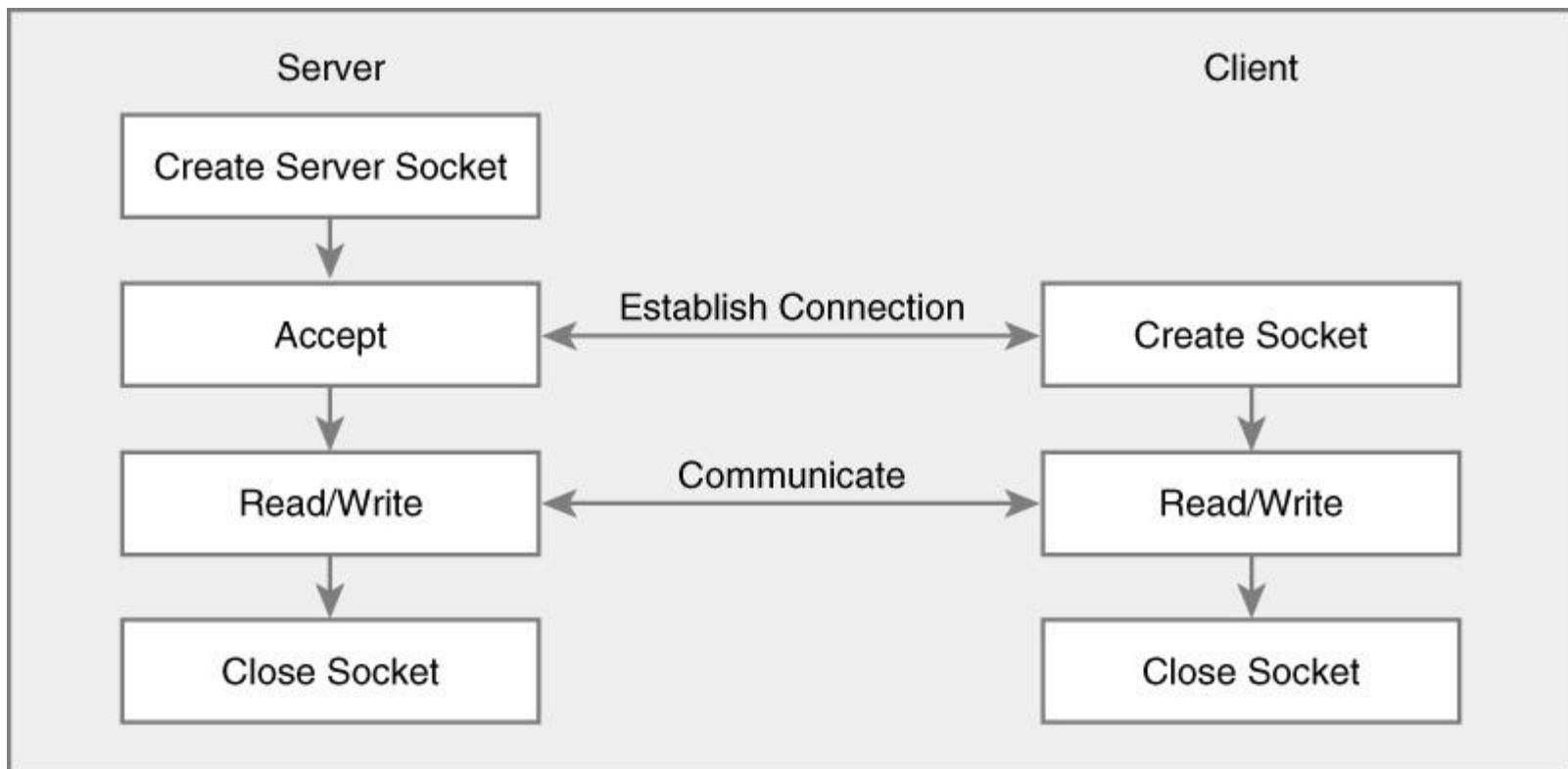
# ServerSocket Methods

---

- accept()
- close()
- getInetAddress()
- getLocalPort()

# Connection Oriented

- TCP Protocol



# DatagramSocket Class

---

- Create UDP socket
  - Does not distinguish server / client sockets
- Constructor specifies InetAddress, port
- Set up UDP socket connection
- Send / receive DatagramPacket

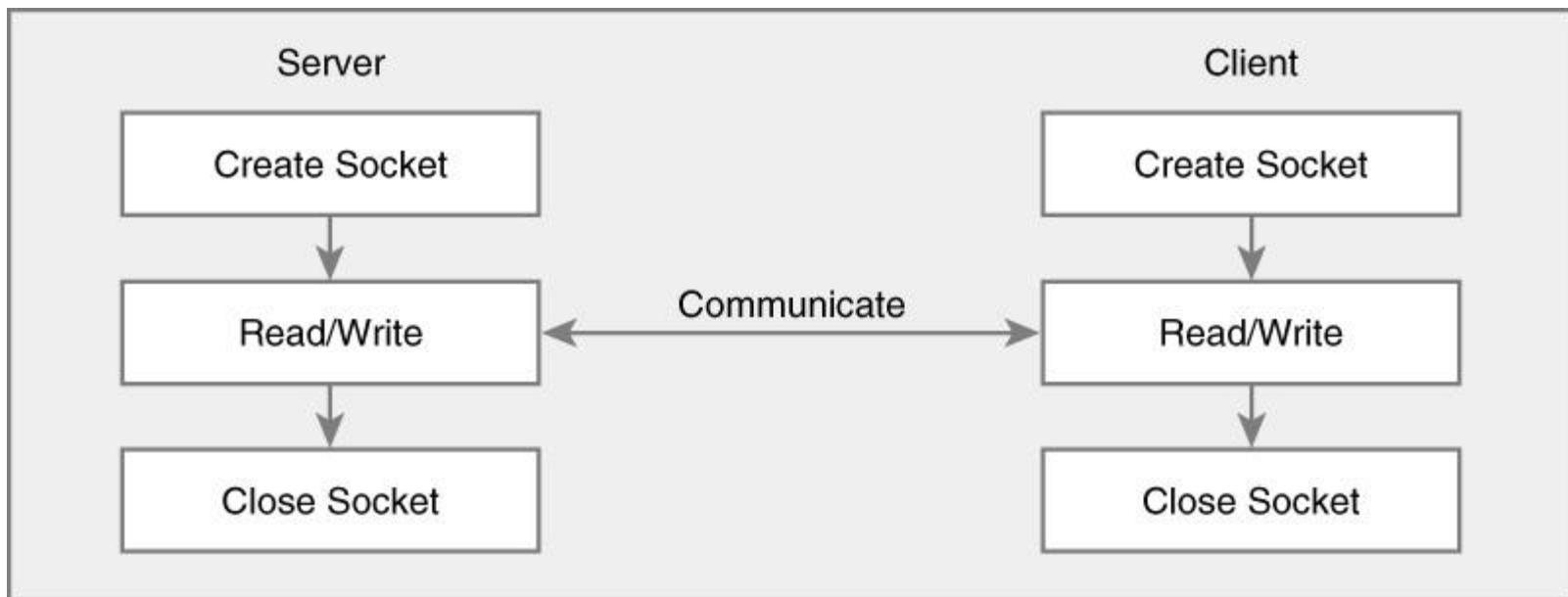
# DatagramSocket Methods

---

- close()
- getLocalAddress()
- getLocalPort()
- receive(DatagramPacket p)
- send(DatagramPacket p)
- setSoTimeout(int t)
- getSoTimeout()

# Packet Oriented

- UDP Protocol



# Java Applets

---

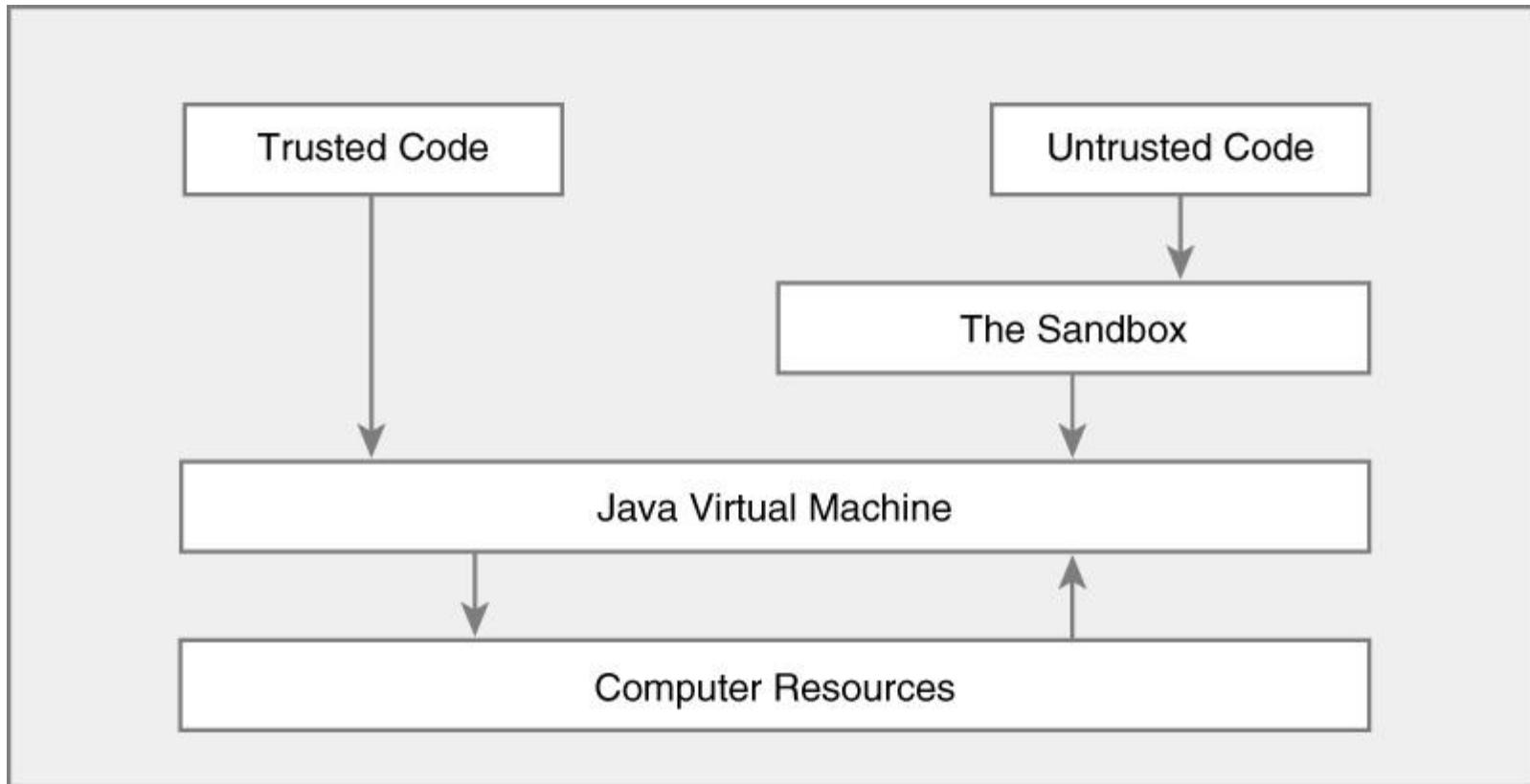
- Applets are Java programs
  - Classes downloaded from network
  - Run in browser on client
- Applets have special security restrictions
  - Executed in **applet sandbox**
  - Controlled by **java.lang.SecurityManager**

# Applet Sandbox

---

- Prevents
  - Loading libraries
  - Defining native methods
  - Accessing local host file system
  - Running other programs (`Runtime.exec()`)
  - Listening for connections
  - Opening sockets to new machines
    - Except for originating host
- Restricted access to system properties

# Applet Sandbox



# Network Summary

---

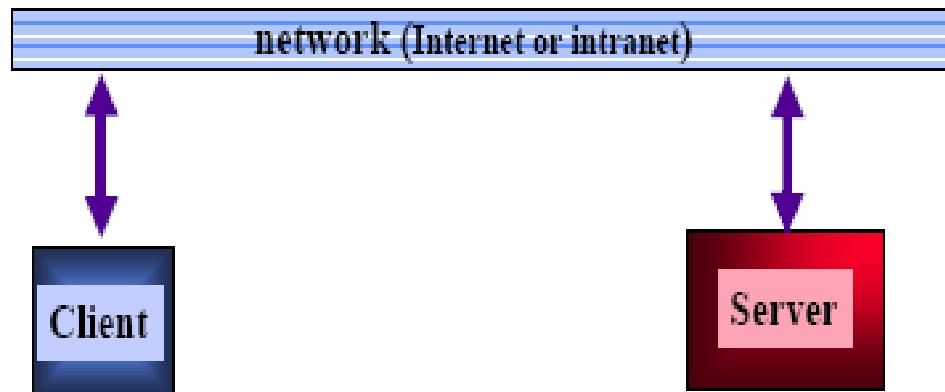
- Internet
  - Designed with multiple layers of abstraction
  - Underlying medium is unreliable, packet oriented
  - Provides two views
    - Reliable, connection oriented (TCP)
    - Unreliable, packet oriented (UDP)
- Java
  - Object-oriented classes & API
    - Sockets, URLs
    - Extensive networking support

# Java Sockets

---

- The `java.net.Socket` class is Java's fundamental class for performing client-side TCP/IP operations
  - Other classes, such as `URL`, `URLConnection`, `Applet`, and `JEditorPane`, all ultimately end up invoking the methods of the `Socket` class
- The constructor is
  - `public Socket(String host, int port)`
  - It attempts to connect to the foreign host at the specified port number

# clients and servers



**Java Applet in Web page  
Java application  
C++ or other application**

**Java server  
C++ server**

# Sockets

---

- The specifics of the TCP handshake process and so on, are covered in the section on the transport layer
- In a nutshell, each port is assigned to a specific process in the computer's memory
  - So if I'm sending a packet to a specific port on a machine, I'm sending it to a particular process that is running on that machine
  - The ports provide a way of multiplexing the different processes to allow multiple processes to be able to utilize the network resources

# Sockets

---

- A socket is the combination of the computer's IP address and port number
  - The computer's IP address uniquely identifies it in the network space
  - The process' port number uniquely identifies it in the computer
  - So the combination of IP address and port number should uniquely identify a process within a network

# Sockets

---

- Functionally, any socket library will have to perform the same, since the protocols all work according to standard
  - How they do things internally, is up to the developers of the operating system and socket library
  - So long as they adhere to the internet standards, when they put information on the network, no one is likely to complain

# Sockets

---

- So any communications between client and server are functionally communications between the two sockets
- In order to uniquely identify that communication stream, you need a 5-tuple that Stevens calls an *association*
  - Local IP address
  - Local Port Number
  - Foreign IP address
  - Foreign Port Number
  - Protocol
- The protocol is necessary because it is possible to have simultaneous TCP and UDP streams between the same sockets

# Unit Summary

---

–Networking

–Networking basics

–Working with URLs

- Creating and Parsing URLs

- Reading from URLs

- Connecting to a URL

- Reading from and writing to URL connection.

–Sockets

- Reading from and writing to sockets

- Writing server side sockets

**END**

# Break Time – 15 minutes

---

