ADVANCED JAVA PROGRAMMING

SUBJECT CODE:18PCS2

Java and its Evolution

Java - An Introduction

- Java - The new programming language developed by Sun Microsystems in 1991.
- Originally called Oak by James Gosling, one of the inventors of the Java Language.
- Java -The name that survived a patent search
- Java Authors: James , Arthur Van , and others
- Java is really “C++ -- ++ “

Java Introduction

- Originally created for consumer electronics (TV, VCR, Freeze, Washing Machine, Mobile Phone).
- Java - CPU Independent language
- Internet and Web was just emerging, so Sun turned it into a language of Internet Programming.
- It allows you to publish a webpage with Java code in it.

Java Milestones

Year	Development
1990	Sun decided to developed special software that could be used for electronic devices. A project called Green Project created and head by James Gosling.
1991	Explored possibility of using C++, with some updates announced a new language named "Oak"
1992	The team demonstrated the application of their new language to control a list of home appliances using a hand held device.
1993	The World Wide Web appeared on the Internet and transformed the text-based interface to a graphical rich environment. The team developed Web applets (time programs) that could run on all types of computers connected to the Internet.

Java Milestones

Year	Development
1994	The team developed a new Web browsed called “Hot Java” to locate and run Applets. HotJava gained instance success.
1995	Oak was renamed to Java, as it did not survive “legal” registration. Many companies such as Netscape and Microsoft announced their support for Java
1996	Java established itself it self as both 1. “the language for Internet programming” 2. a general purpose OO language.
1997-	A class libraries, Community effort and standardization, Enterprise Java, Clustering, etc..

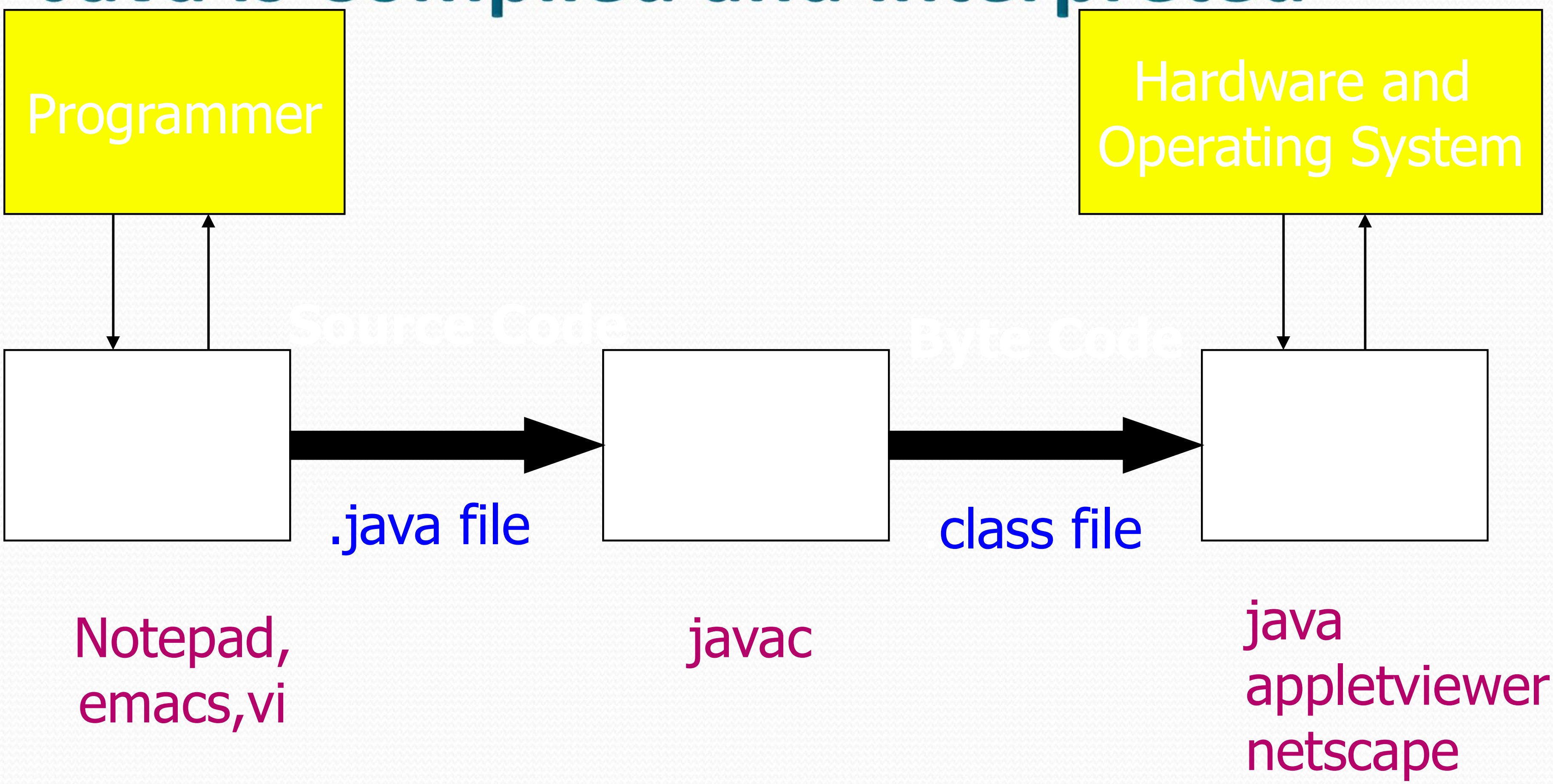
Sun white paper defines Java as:

- Simple and Powerful
- Safe
- Object Oriented
- Robust
- Architecture Neutral and Portable
- Interpreted and High Performance
- Threaded
- Dynamic

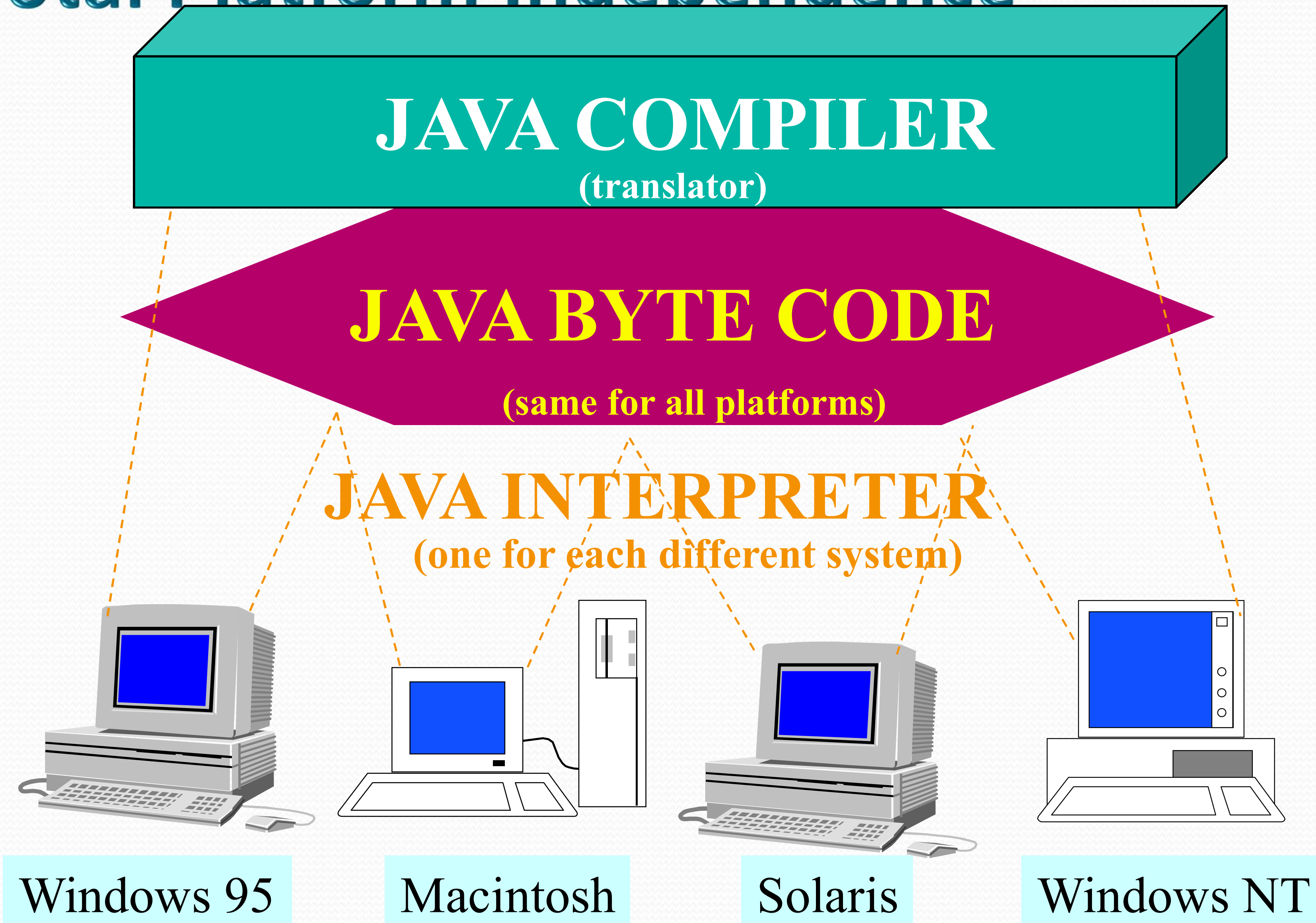
Java Attributes

- Familiar, Simple, Small
- Compiled and Interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Multithreaded and Interactive
- High Performance
- Dynamic and Extensible

Java is Compiled and Interpreted



Total Platform Independence



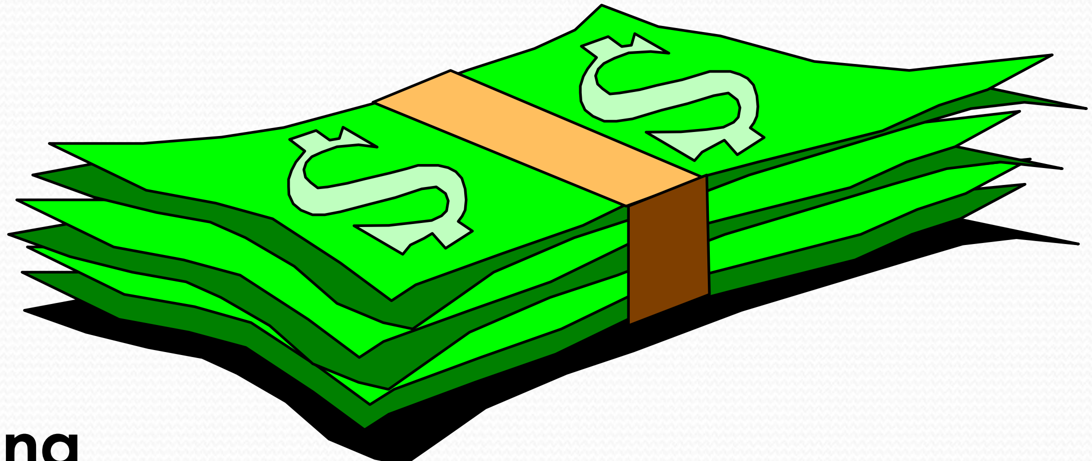
Architecture Neutral & Portable

- Java Compiler - Java *source code* (file with extension .java) to *bytecode* (file with extension .class)

- *Bytecode* - an intermediate form, closer to machine representation

- A interpreter (virtual machine) on any target platform interprets the bytecode.

Rich Class Environment



- Core Classes

  language

  Utilities

  Input/Output

  Low-Level Networking

  Abstract Graphical User Interface

- Internet Classes

  TCP/IP Networking

  WWW and HTML

  Distributed Programs

- A programming language is a set of commands, instructions, and other syntax use to create a software program. Languages that programmers use to write code are called "high-level languages." This code can be compiled into a "low-level language," which is recognized directly by the computer hardware.

- High-level languages are designed to be easy to read and understand. This allows programmers to write source code in a natural fashion, using logical words and symbols. For example, reserved words like function, while, if, and else are used in most major programming languages. Symbols like <, >, ==, and != are common operators. Many high-level languages are similar enough that programmers can easily understand source code written in multiple languages.

- Examples of high-level languages include C++, Java, Perl, and PHP. Languages like C++ and Java are called "compiled languages" since the source code must first be compiled in order to run. Languages like Perl and PHP are called "interpreted languages" since the source code can be run through an interpreter without being compiled. Generally, compiled languages are used to create software applications, while interpreted languages are used for running scripts, such as those used to generate content for dynamic websites.
- Low-level languages include assembly and machine languages. An assembly language contains a list of basic instructions and is much more difficult to read than a high-level language. In rare cases, a programmer may decide to code a basic program in an assembly language to ensure it operates as efficiently as possible. An assembler can be used to translate the assembly code into machine code. The machine code, or machine language, contains a series of binary codes that are understood directly by a computer's CPU. Needless to say, machine language is not designed to be human readable.

- Programmers write their programs in a high level programming language such as Java, C++.

- A computer only understands its own language called “machine language”.

- A compiler is needed to translate high level program code into machine language code that will be understood by the computer.

- After a program is compiled, the machine code can be executed on the computer, say Windows, for which it was compiled. If the program is to be executed on another platform, say Mac, the program will first have to be compiled for that platform and can then be executed.
- Java is a very popular high level programming language
- Java has been used widely to create various types of computer applications such as database applications, desktop applications, Web based applications, mobile applications, and games among others.

Datatypes and Variables

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Literals and Variables

- Literals
  - 456—a literal numerical constant
  - System.out.println(456); // Java
  - “A Literal String Constant”
  - System.out.println(“My First Java”); // Java

- Variables
  - It is a named computer location in memory that holds values that might vary
  - That location have an address

Data type Declarations

- Specify the type of data and the length of the data item in bytes
- int, short, long
- float, double
- boolean
- Char

There are eight primitive data types

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

PRIMITIVE	SIZE IN BITS	RANGE
int	32	-2 to the 31st to 2 to the 31st
int	4 bytes	2147483648
long	64 -- 8 bytes	-2 to the 63rd to 2 to the 63rd
float	32	+- 1.5 x 10^45
double	64	+- 5.0 x 10^324
decimal	128	28 significant figures
string	16 bits per char	Not applicable
char	16	One character
boolean	8	True or false

Numeric data types in Java: integers

Data type name	Minimum value	Maximum value
byte	-128	127
short	-32,768	32,767
int	-2,147,483,648	2,147,483,647
long	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Numeric data types in Java: floating-point numbers

Data type name	Minimum value	Maximum value
float	$-3.40282347 \times 10^{38}$	$3.40282347 \times 10^{38}$
double	$-1.79769313486231570 \times 10^{308}$	$1.79769313486231570 \times 10^{308}$

The assignment operator =

- int A = 36;
  - Sets a = to the constant 36 at execution time
- Int A =36;
  - Sets A = to the constant 36 at compile time
  - Initializes A to 36 at the time memory is set aside for it

Conversion Between Primitive Data Types

- Java is known as a strongly typed language.
  - In a Strongly Typed Language before a value is assigned to a variable, Java checks the types of the variable and the value being assigned to it to determine if they are compatible.
- For example:

```
int x;
double y = 2.5;
x = y;
```

This will cause an error

```
int x;
short y;
x = y;
```

This will NOT cause an error

…but, why?

Conversion Between Primitive Data Types

- Types in Java have “ranks”.
  - Ranks here means that if a type has a higher rank than another, it can hold more numbers, and thus, will not lose any precision.
  - Ranks (Highest to Lowest):
    1. `double`
    2. `float`
    3. `long`
    4. `int`
    5. `short`
    6. `byte`

Variables

- Variables are containers for storing data values.
- In Java, there are different **types** of variables, for example:
- String - stores text, such as "Hello". String values are surrounded by double quotes
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

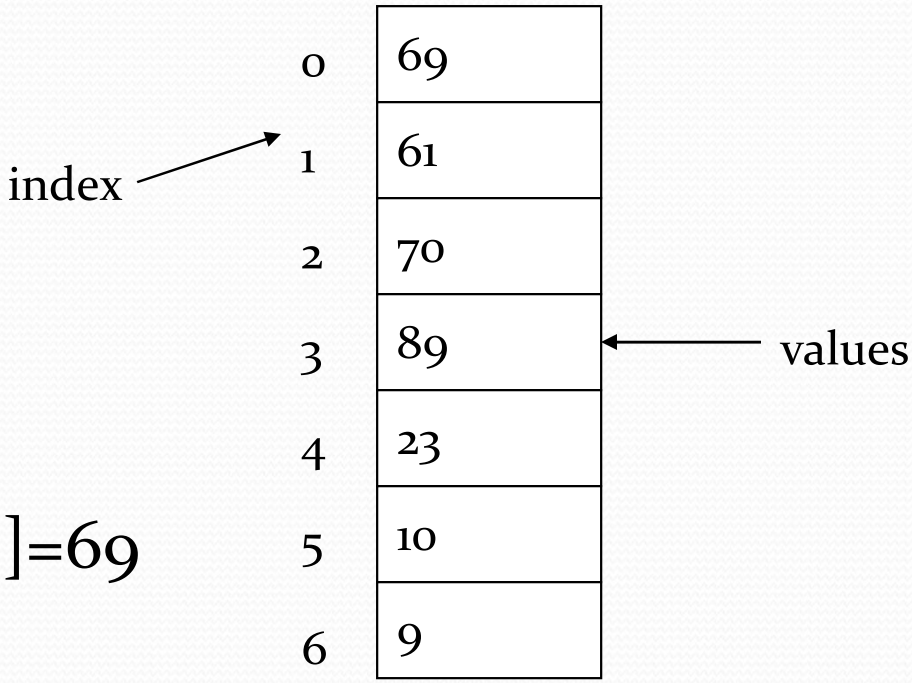
Arrays

An array is a data structure that contains a group of elements. Typically these elements are all of the same data type, such as an integer or string. Arrays are commonly used in computer programs to organize data so that a related set of values can be easily sorted or searched.

Arrays - Introduction

- An array is a group of contiguous or related data items that share a common name.
- Used when programs have to handle large amount of data
- Each value is stored at a specific position
- Position is called a index or superscript. Base index = 0
- The ability to use a single name to represent a collection of items and refer to an item by specifying the item number enables us to develop concise and efficient programs.

Arrays - Introduction



- A[0]=69

Declaration of Arrays

- Like any other variables, arrays must declared and created before they can be used. Creation of arrays involve three steps:
  - Declare the array
  - Create storage area in primary memory.
  - Put values into the array (i.e., Memory location)
- Declaration of Arrays:
  - Form 1:

    Type arrayname[]
  - Form 2:
    - Type [] arrayname;

  - Examples:

    ```
    int[] students;
    int students[];
    ```
  - Note: we don't specify the size of arrays in the declaration.

Creation of Arrays

- After declaring arrays, we need to allocate memory for storage array items.
- In Java, this is carried out by using “new” operator, as follows:
  - Arrayname = **new** type[size];
- Examples:
  - students = new int[7];

Initialisation of Arrays

- Once arrays are created, they need to be initialised with some values before access their content. A general form of initialisation is:
  - Arrayname [index/subscript] = value;
- Example:
  - students[0] = 50;
  - students[1] = 40;
- Java creates arrays starting with subscript 0 and ends with value one less than the size specified.
- Java protects arrays from overruns and under runs. Trying to access an array beyond its boundaries will generate an error message.

Arrays – Length

- Arrays are fixed length
- Length is specified at create time
- In java, all arrays store the allocated size in a variable named “length”.
- We can access the length of arrays as arrayName.length:

  e.g. `int x = students.length;      // x = 7`

- Accessed using the index

  e.g. `int x = students [1];      // x = 40`

Arrays – Example

```
// StudentArray.java: store integers in arrays and access
public class StudentArray{
    public static void main(String[] args) {
        int[] students;
        students = new int[7];
        System.out.println("Array Length = " + students.length);

        for ( int  i=0;  i < students.length;  i++)
            students[i] = 2*i;
        System.out.println("Values Stored in Array:");
        for ( int  i=0;  i < students.length;  i++)
            System.out.println(students[i]);
    }
}
```

Arrays – Initializing at Declaration

- Arrays can also be initialised like standard variables at the time of their declaration.
  - Type arrayname[] = {list of values};
- Example:

```
int[] students = {55, 69, 70, 30, 80};
```

- Creates and initializes the array of integers of length 5.
- In this case it is not necessary to use the *new* operator.

Arrays – Example

```
// StudentArray.java: store integers in arrays and access
public class StudentArray{
    public static void main(String[] args) {
    int[] students = {55, 69, 70, 30, 80};

    System.out.println("Array Length = " + students.length);
    System.out.println("Values Stored in Array:");
     for ( int  i=0;  i < students.length;  i++)
         System.out.println(students[i]);
    }
}
```

Two Dimensional Arrays

- Two dimensional arrays allows us to store data that are recorded in table. For example:
- Table contains 12 items, we can think of this as a matrix consisting of 4 rows and 3 columns.

Sold / Person	Item1	Item2	Item3
Salesgirl #1	10	15	30
Salesgirl #2	14	30	33
Salesgirl #3	200	32	1
Salesgirl #4	10	200	4

```c
int num[3][4] = {
  {1, 2,  3,  4},
  {5, 6,  7,  8},
  {9, 10, 11, 12}
};
```

col →

row ↓

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

2D arrays manipulations

- Declaration:
  - int myArray [][];
- Creation:
  - myArray = new int[4][3]; // OR
  - int myArray [][] = new int[4][3];
- Initialisation:
  - Single Value;
    - myArray[0][0] = 10;
  - Multiple values:
    - int tableA[2][3] = {{10, 15, 30}, {14, 30, 33}};
    - int tableA[][] = {{10, 15, 30}, {14, 30, 33}};

Example-Multidimensional Array

```
public class multiDimensional
{
    public static void main(String args[])
    {
        // declaring and initializing 2D array
        int arr[ ][ ] = { {2,7,9},{3,6,1},{7,4,2} };

        // printing 2D array
        for (int i=0; i< 3 ; i++)
        {
            for (int j=0; j < 3 ; j++)
                System.out.print(arr[i][j] + " ");

            System.out.println();
        }
    }
}
```

Java Operators

Assignment Operator (=)

lvalue = rvalue;

```
w = 10;
x = w;
z = (x - 2)/(2 + 2);
```

- Take the value of the **rvalue** and store it in the **lvalue**.
- The **rvalue** is any constant, variable or expression.
- The **lvalue** is named variable.

Mathematical Operators

- Addition +
- Subtraction -
- Multiplication *
- Division /
- Modulus %

Shorthand Operators

+=, -=, *=, /=, %=

Common	Shorthand
`a = a + b;`	`a += b;`
`a = a - b;`	`a -= b;`
`a = a * b;`	`a *= b;`
`a = a / b;`	`a /= b;`
`a = a % b;`	`a %= b;`

Shorthand Increment and Decrement `++ and --`

Common	Shorthand
`a = a + 1;`	**`a++; or ++a;`**
`a = a - 1;`	**`a--; or --a;`**

The Logical and Relational Operators

Relational Operators

`>`   `<`   `>=`   `<=`   `==`   `!=`

Primitives

- **Greater Than** `>`
- **Less Than** `<`
- **Greater Than or Equal** `>=`
- **Less Than or Equal** `<=`

Primitives or Object References

- **Equal (Equivalent)** `==`
- **Not Equal** `!=`

The Result is Always `true` or `false`

Logical Operators (boolean)

- **Logical AND** `&&`
- **Logical OR** `||`
- **Logical NOT** `!`

Logical (&&) Operator Examples

```
public class Example {
  public static void main(String[] args) {
        boolean t = true;
        boolean f = false;

        System.out.println("f && f " + (f && f));
        System.out.println("f && t " + (f && t));
        System.out.println("t && f " + (t && f));
        System.out.println("t && t " + (t && t));

  }
}
```

```
> java Example
f && f false
f && t false
t && f false
t && t true
>
```

Logical (||) Operator Examples

```
public class Example {
  public static void main(String[] args) {
       boolean t = true;
       boolean f = false;

       System.out.println("f || f " + (f || f));
       System.out.println("f || t " + (f || t));
       System.out.println("t || f " + (t || f));
       System.out.println("t || t " + (t || t));

  }
}
```

```
> java Example
f || f false
f || t true
t || f true
t || t true
>
```

Logical (!) Operator Examples

```
public class Example {
  public static void main(String[] args) {
        boolean t = true;
        boolean f = false;

        System.out.println("!f " + !f);
        System.out.println("!t " + !t);

  }
}
```

```
> java Example
!f true
!t false
>
```

Ternary Operator

? :

Any expression that evaluates to a boolean value.

boolean_expression **?** expression_1 **:** expression_2

If **true** this expression is evaluated and becomes the value entire expression.

If **false** this expression is evaluated and becomes the value entire expression.

Ternary (? :) Operator Examples

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



  }
}
```

```
> java Example
t?true:false true
t?1:2 1
f?true:false false
f?1:2 2
>
```

String (+) Operator

String Concatenation

```
"Now is " + "the time."
```

```
"Now is the time."
```

String (+) Operator

Automatic Conversion to a String

expression_1 **+** expression_2

If either **expression_1** or **expression_2** evaluates to a string the other will be converted to a string if needed. The result will be their concatenation.

String IsEmpty()

This method checks whether the String contains anything or not. If the java String is Empty, it returns true else false.

```
public class IsEmptyExample{
public static void main(String args[]){
String s1="";
String s2="hello";
System.out.println(s1.isEmpty());      // true
System.out.println(s2.isEmpty());      // false
}
}
```

String Trim()

The java string trim() method removes the leading and trailing spaces. It checks the unicode value of space character ('u0020') before and after the string. If it exists, then removes the spaces and return the omitted string.

```
public class StringTrimExample{
public static void main(String args[]){
String s1="  hello   ";
System.out.println(s1+"how are you");        // without trim()
System.out.println(s1.trim()+"how are you"); // with trim()
}
}
```

String toLowerCase()

The java string toLowerCase() method converts all the characters of the String to lower case.

```
public class StringLowerExample
{
public static void main(String args[]){
String s1="HELLO HOW Are You?";
String s1lower=s1.toLowerCase();
System.out.println(s1lower);}
}
```

String toUpper()

The Java String toUpperCase() method converts all the characters of the String to upper case.

```
public class StringUpperExample
{
public static void main(String args[])
{
String s1="hello how are you";
String s1upper=s1.toUpperCase();
System.out.println(s1upper);
}
}
```

Operator Precedence

<table>
<tr><td>Unary</td><td>+ - ++ -- ! ~ ()</td></tr>
<tr><td>Arithmetic</td><td>* / %</td></tr>
<tr><td>Shift</td><td>+ -</td></tr>
<tr><td>Comparison</td><td><< >> >>></td></tr>
<tr><td>Logical Bit</td><td>> < >= <= instanceof</td></tr>
<tr><td>Boolean</td><td>== !=</td></tr>
<tr><td>Ternary</td><td>& | ^</td></tr>
<tr><td>Assignment</td><td>&& ||</td></tr>
<tr><td></td><td>?:</td></tr>
<tr><td></td><td>= (and += etc.)</td></tr>
</table>

Control Statements

- Selection Statements
  - Using `if` and `if...else`
  - Nested `if` Statements
  - Using `switch` Statements
  - Conditional Operator
- Repetition Statements
  - Looping: `while`, `do-while`, and `for`
  - Nested loops
  - Using `break` and `continue`

Selection Statements

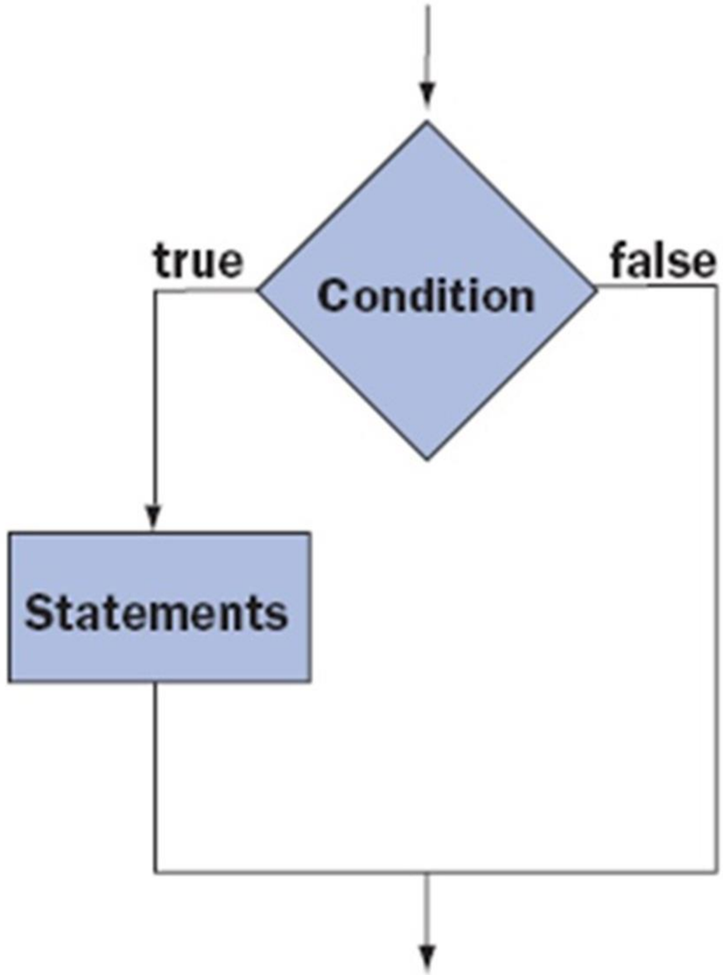
- `if` Statements
- `switch` Statements
- Conditional Operators

`if` Statements

```
if (booleanExpression) {
  statement(s);
}
```

Example:

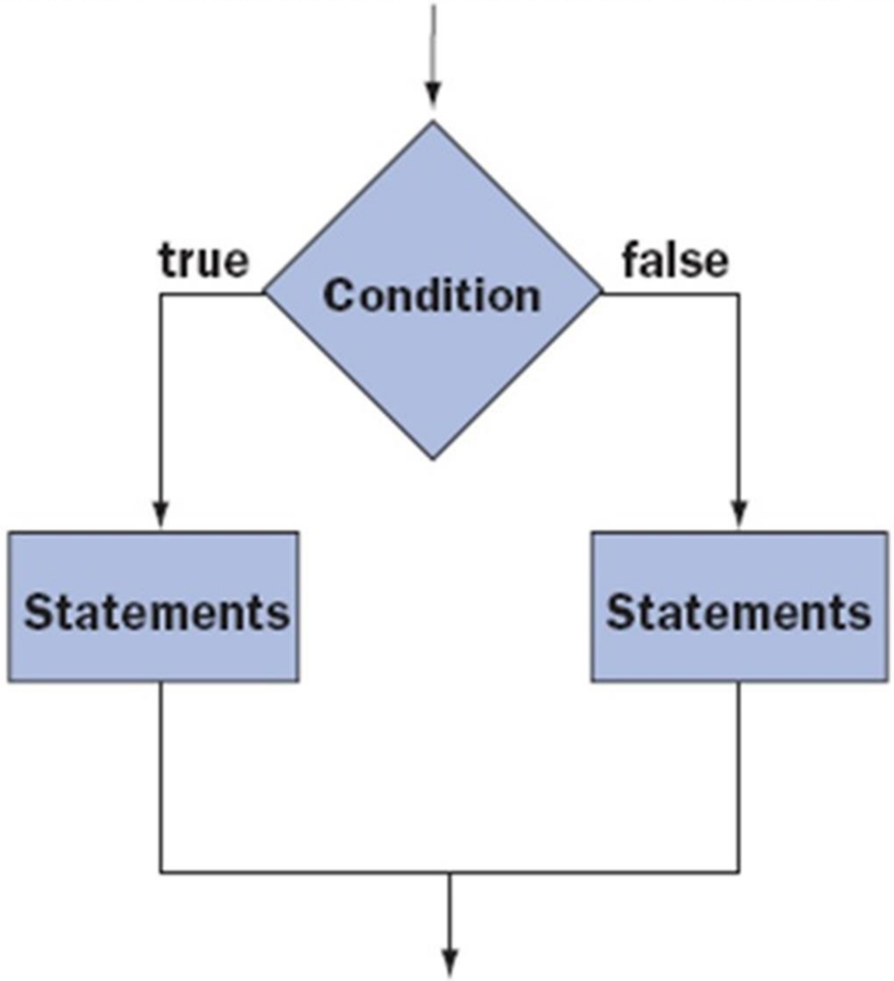
```
if ((i > 0) && (i < 10)) {
  System.out.println("i is an " +
    "integer between 0 and 10");
}
```



(a) `if` statement

The `if...else` Statement

```
if (booleanExpression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```

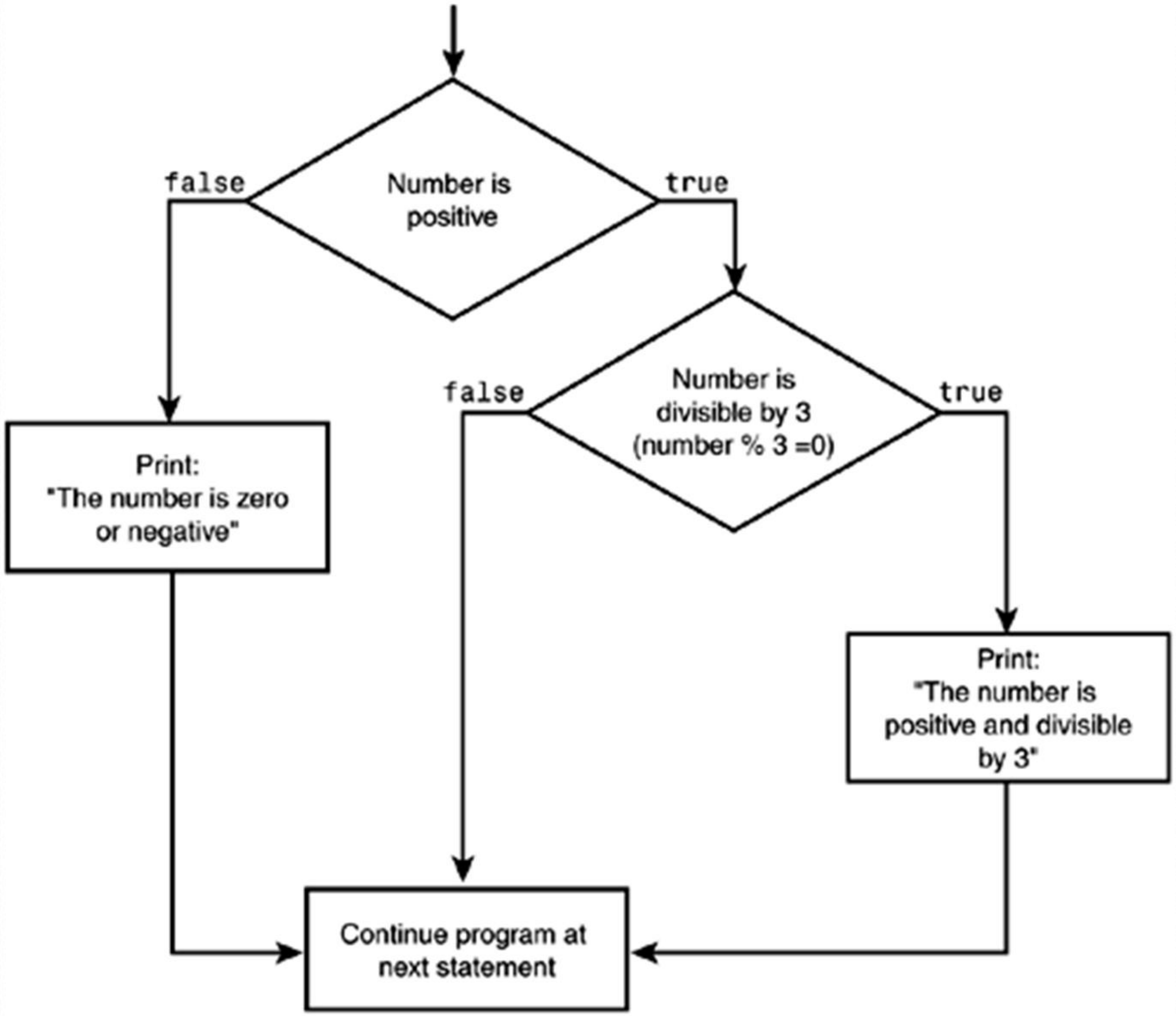


(b) `if-else` statement

`if...else` Example

```
if (radius >= 0) {
  area = radius*radius*PI;

 System.out.println("The area for the "
    + "circle of radius " + radius +
    " is " + area);
}
else {
  System.out.println("Negative input");
}
```

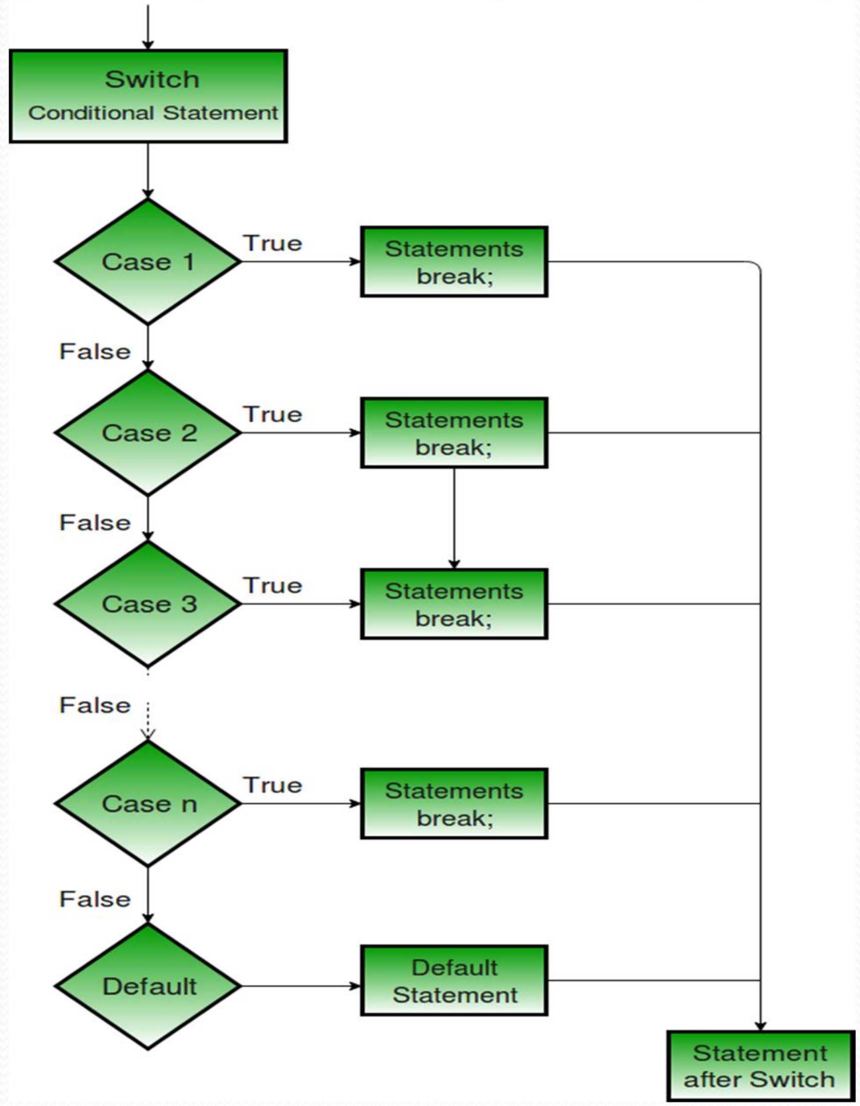
Nested If Statements



`switch` Statements

```
switch (year) {
  case 7:  annualInterestRate = 7.25;
           break;
  case 15: annualInterestRate = 8.50;
           break;
  case 30: annualInterestRate = 9.0;
           break;
  default: System.out.println(
   "Wrong number of years, enter 7, 15, or 30");
}
```

`switch` Statement Flow Chart



`switch` Statement Rules

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. (The case statements are executed in sequential order.)

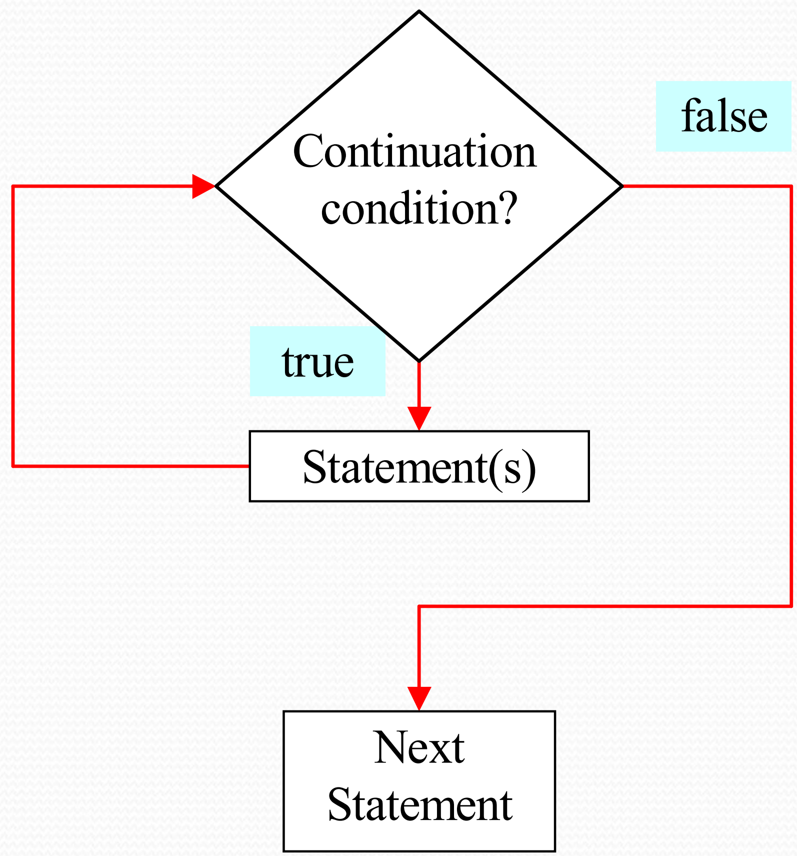
The keyword break is optional, but it should be used at the end of each case in order to terminate the remainder of the switch statement. If the break statement is not present, the next case statement will be executed.

Repetitions

- `while` Loops
- `do-while` Loops
- `for` Loops
- `break` and `continue`

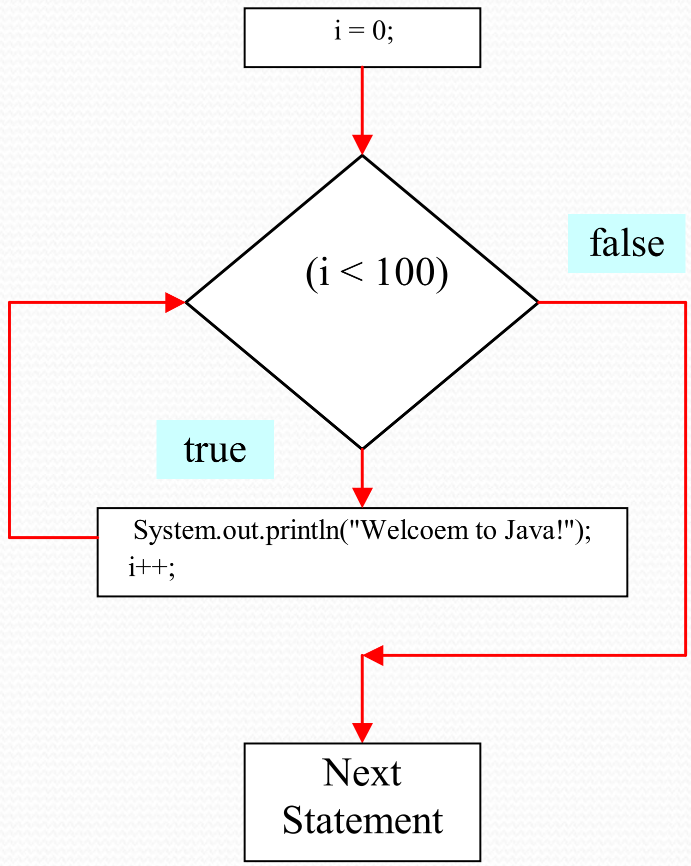
`while` Loop Flow Chart

```
while (continuation-condition) {
  // loop-body;
}
```



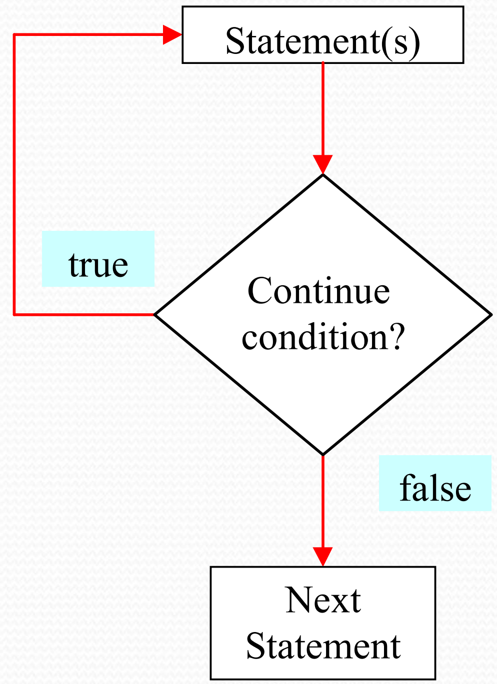
`while` Loop Flow Chart, cont.

```
int i = 0;
while (i < 100) {
  System.out.println(
    "Welcome to Java!");
  i++;
}
```



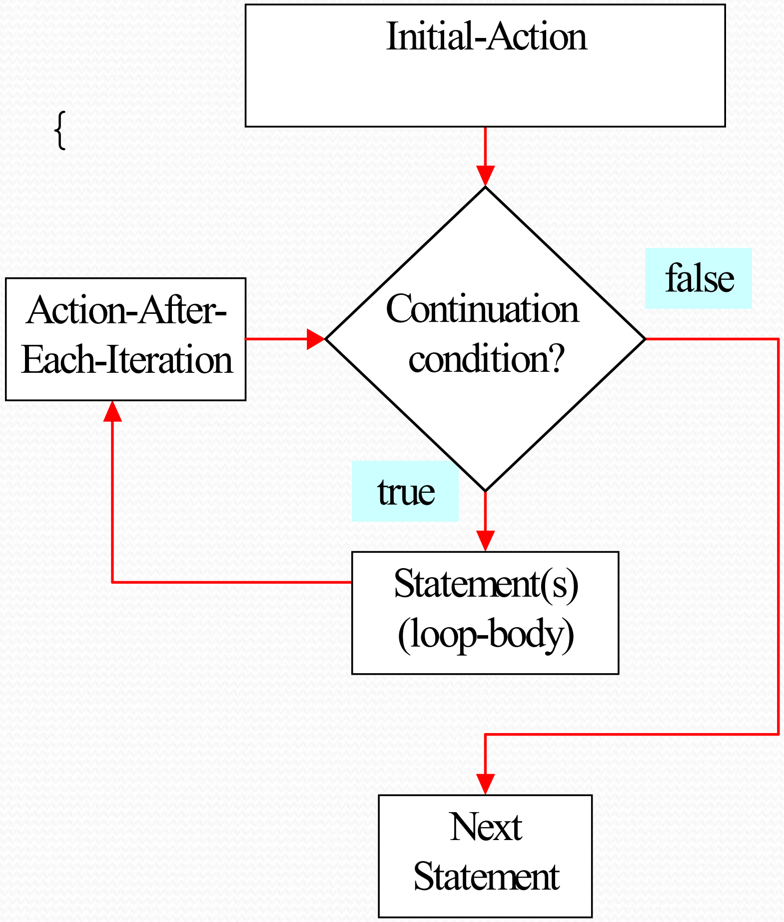
do-while Loop

```
do {
  // Loop body;
} while (continue-condition);
```

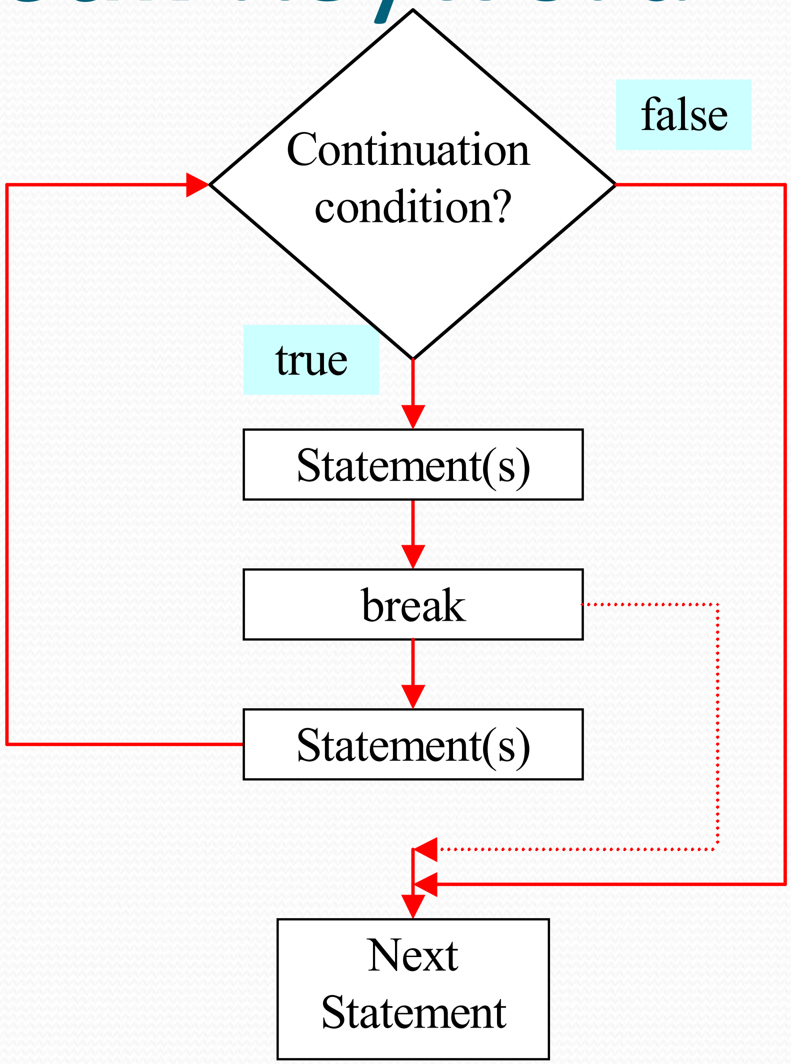


`for` Loop Flow Chart

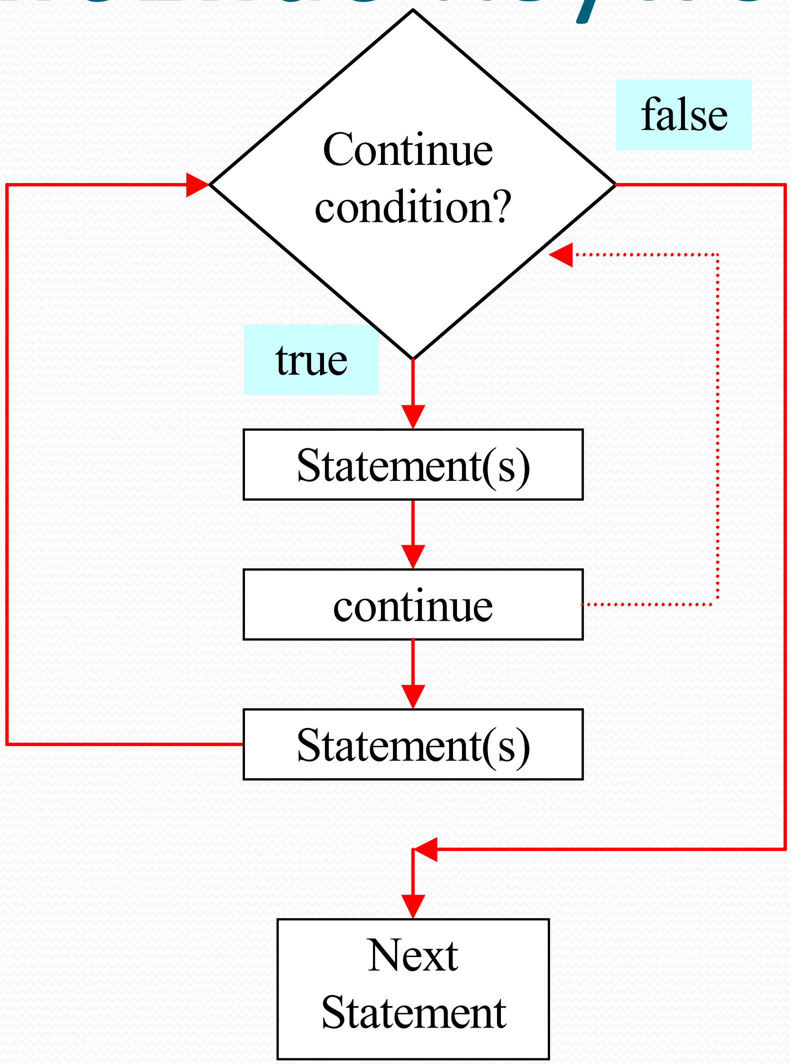
```
for (initial-action;
  loop-continuation-condition;
  action-after-each-iteration) {
   //loop body;
}
```

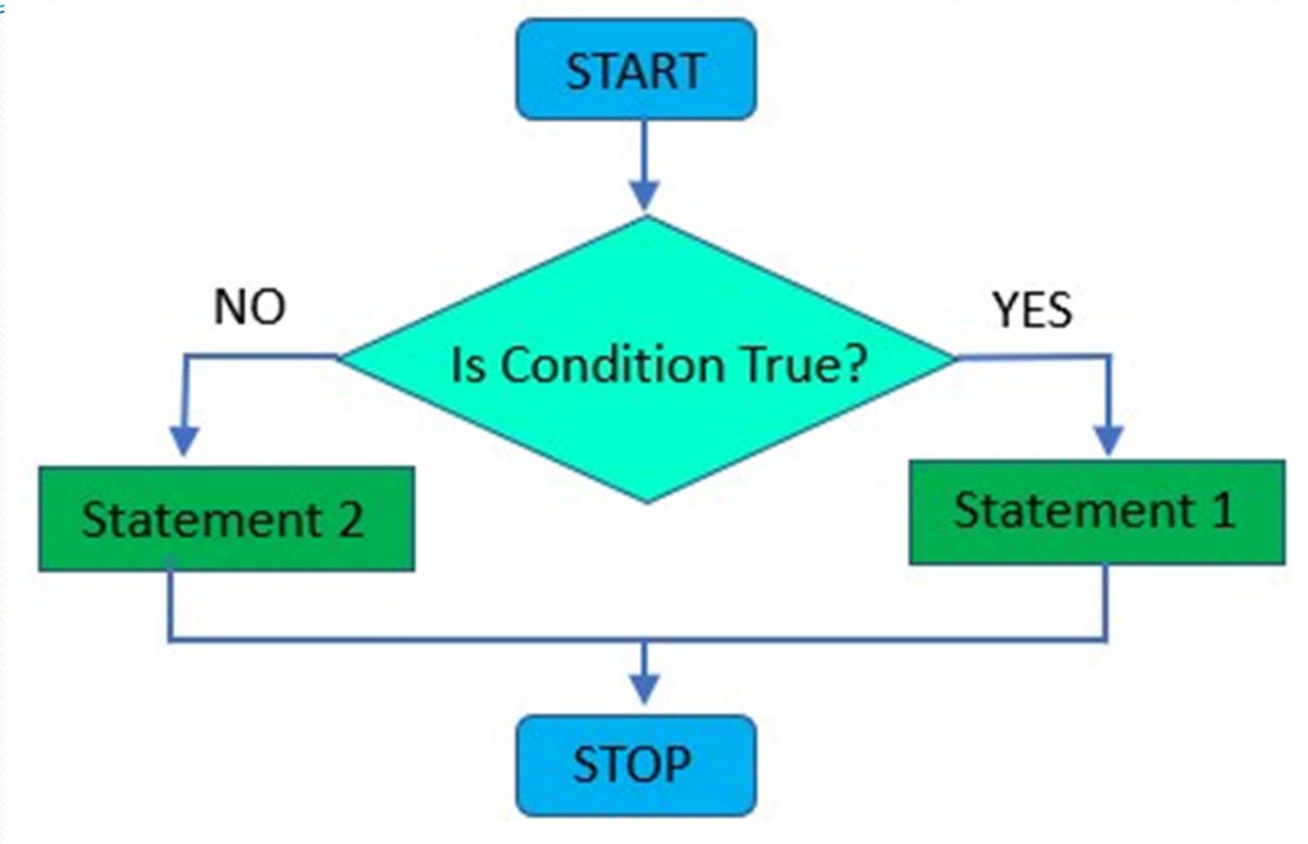


The `break` Keyword



The `continue` Keyword





Condn?Exp1:Exp2

Introduction to Class and Objects

Introduction

- Java is a true Object Oriented language and therefore the underlying structure of all Java programs is classes.

- Anything we wish to represent in Java must be encapsulated in a class that defines the “state” and “behaviour” of the basic program components known as objects.

- Classes create objects and objects use methods to communicate between them. They provide a convenient method for packaging a group of logically related data items and functions that work on them.

- A class essentially serves as a template for an object and behaves like a basic data type “int”.

- It is therefore important to understand how the fields and methods are defined in a class and how they are used to build a Java program that incorporates the basic Object Oriented concepts such as encapsulation, inheritance, and polymorphism.

Classes

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.

Circle
centre radius
circumference() area()

Classes

- The basic syntax for a class definition:

```
class  ClassName [extends
SuperClassName]
{
        [fields declaration]
        [methods declaration]
}
```

- Bare bone class – no fields, no methods

```
public class Circle {
    // my circle class
}
```

Adding Fields: Class Circle with fields

- Add *fields*

```
public class Circle {
    public double x, y;  // centre coordinate
    public double r;     //  radius of the circle

}
```

- The fields (data) are also called the *instance* varaibles.

Adding Methods

- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- The general form of a method declaration is:

```
type MethodName (parameter-list)
{
        Method-body;
}
```

Adding Methods to Class Circle

```
public class Circle {

    public double x, y; // centre of the circle
    public double r;    // radius of circle

    //Methods to return circumference and area
    public double circumference() {
                return 2*3.14*r;
    }
    public double area() {
                return 3.14 * r * r;
    }
}
```

Method Body

Accessing Object/Circle Data

- Similar to C syntax for accessing data defined in a structure.

ObjectName.VariableName
ObjectName.MethodName(parameter-list)

```
Circle aCircle = new Circle();

aCircle.x = 2.0 // initialize center and radius
aCircle.y = 2.0
aCircle.r = 1.0
```

Executing Methods in Object/Circle

- Using Object Methods:

sent ‘message’ to aCircle

```
Circle aCircle = new Circle();

double area;
aCircle.r = 1.0;
area = aCircle.area();
```

Using Circle Class

```
// Circle.java:  Contains both Circle class and its user class
//Add Circle class code here
class MyMain
{
	public static void main(String args[])
	{
		Circle aCircle;  // creating reference
		aCircle = new Circle(); // creating object
		aCircle.x = 10;  // assigning value to data field
		aCircle.y = 20;
		aCircle.r = 5;
		double area = aCircle.area(); // invoking method
		double circumf = aCircle.circumference();
		System.out.println("Radius="+aCircle.r+" Area="+area);
		System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
	}
}
```

```
class Circle {

      public double x, y; // centre of the circle
      public double r;    // radius of circle

      //Methods to return circumference and area
      public double circumference() {
                              return 2*3.14*r;
      }
      public double area() {
                              return 3.14 * r * r;
      }
}
public class MyMain
{
        public static void main(String args[])
        {
                Circle aCircle;  // creating reference
                aCircle = new Circle(); // creating object
                aCircle.x = 10;  // assigning value to data field
                aCircle.y = 20;
                aCircle.r = 5;
                double area = aCircle.area(); // invoking method
                double circumf = aCircle.circumference();
                System.out.println("Radius="+aCircle.r+" Area="+area);
                System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
        }
```

Constructor

- Java allows objects to initialize themselves when they are created.
- A constructor initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- Once defined, the constructor is automatically called immediately after the object is created.

- By implementing constructor, it would be simpler and more concise to have all of the setup done at the time the object is first created.
- It can be tedious to initialize all of the variables in a class each time an instance is created.
- This automatic initialization is performed through the use of a constructor.

- Constructors have no return type
- This is because the implicit return type of a class' constructor is the class type itself.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

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

```
class BoxDemo6
  {
public static void main(String args[])
          {
          Box mybox1 = new Box();
          Box mybox2 = new Box();
          double vol;
          vol = mybox1.volume();
          System.out.println("Volume is " + vol);
          vol = mybox2.volume();
          System.out.println("Volume is " + vol);
          }
  }
```

Constructor Overloading

What is Constructor overloading?

- **Constructor overloading in java** allows having *more than one constructor inside one Class.*
- overloading is not much different than method overloading. Just like in the case of method overloading you have multiple methods with the same name but different signature mean paramete,
- in Constructor overloading you have *multiple constructors with a different signature* with the only difference that Constructor doesn't have a return type in Java. That constructor will be called as an **overloaded constructor** . Overloading is also another form of polymorphism in Java which allows having multiple constructors with a different name in one Class in java.

Why do you overload Constructor in Java ?

When we talk about **Constructor overloading**, the first question comes to mind is **why do someone overload Constructors in Java** or why do we have overloaded constructor ? If you have been using framework or API(Application Programming Interface) like JDK(Java Development Kit) you must have seen a lot of method overloading and constructor overloading. Constructor overloading makes sense if you can Construct object via a different way.

How to overload Constructor in Java?

- **Constructor overloading** is not complex you just need to create another constructor, obviously same name as of class but different signature but there are certain rules related to Constructor overloading which needs to be remembered while *overloading constructor in Java*.
- e.g. One Constructor can only be called from inside of another Constructor and if called it must be the first statement of that Constructor. here is an example of correct and incorrect constructor overloading

Important points related to Constructor overloading

- 1. Constructor overloading is similar to method overloading in Java.
- 2. You can call overloaded constructor by using this() keyword in Java.
- 3. overloaded constructor must be called from another constructor only.
- 4. make sure you add no argument default constructor because once compiler will not add if you have added any constructor in Java.

Important points related to Constructor overloading

- 5. if an overloaded constructor called , it must be the first statement of constructor in java.
- 6. Its best practice to have one primary constructor and let overloaded constructor calls that. this way
- your initialization code will be centralized and easier to test and maintain.

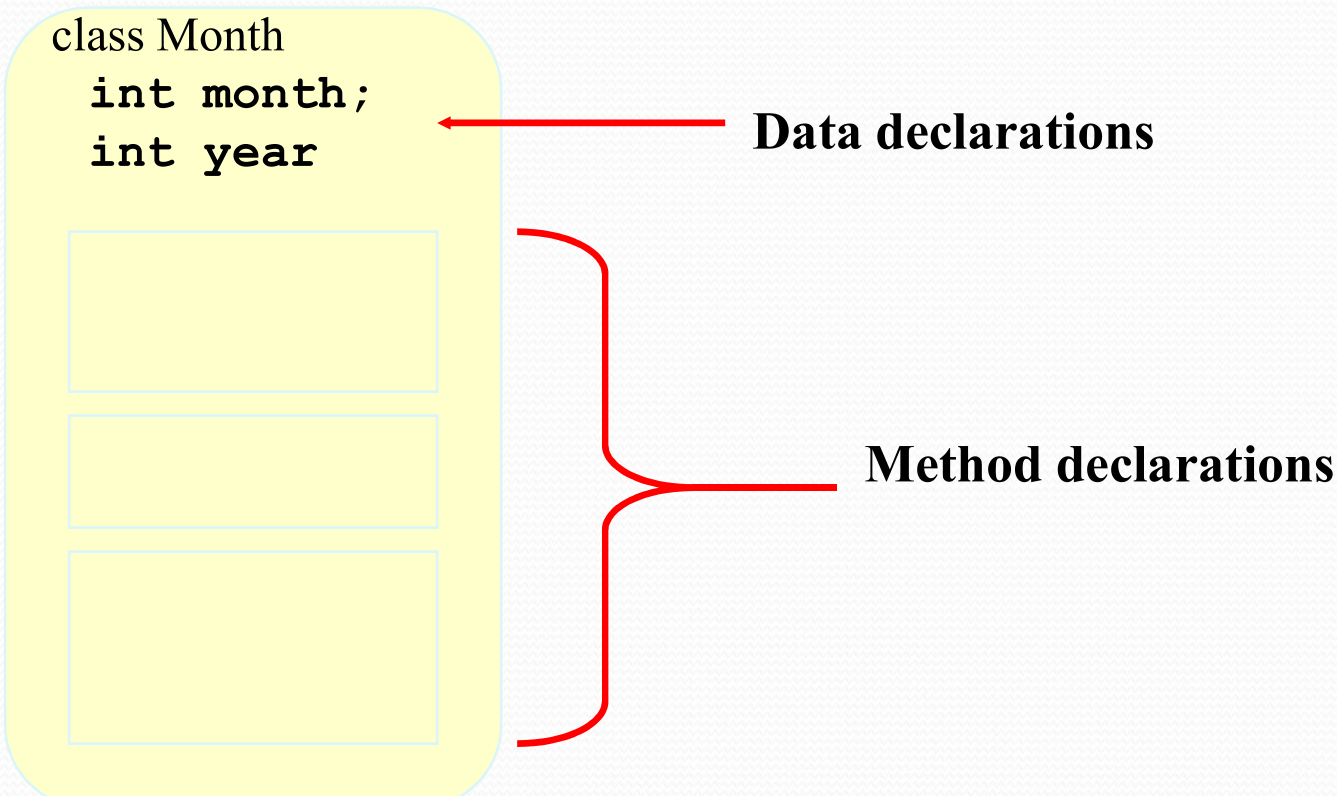
biggest advantage of Constructor overloading

- That's all on **Constructor overloading in java**. The biggest advantage of Constructor overloading is flexibility which allows you to create the object in a different way and classic examples are various Collection classes. Though you should remember that once you add a constructor, a compiler will not add default no argument constructor.

Methods

Defining Classes

- A class contains data declarations (static and instance variables) and method declarations (behaviors)



Methods

- A program that provides some functionality can be long and contains many statements
- A method groups a sequence of statements and should provide a well-defined, easy-to-understand functionality
  - a method takes input, performs actions, and produces output
- In Java, each method is defined within specific class

Method Declaration: Header

- A method declaration begins with a *method header*

```
class MyClass
{ …
  static  int   min ( int num1, int num2 )
```

method name

parameter list

The parameter list specifies the type and name of each parameter

return type

The name of a parameter in the method declaration is called a *formal argument*

properties

Java static method

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Method Declaration: Body

The header is followed by the *method body:*

```
class MyClass
{
    …
    static int min(int num1, int num2)
    {
        int minValue = num1 < num2 ? num1 : num2;
        return minValue;
    }


    …
 }
```

The `return` Statement

- The *return type* of a method indicates the type of value that the method sends back to the calling location
  - A method that does not return a value has a `void` return type

- The *return statement* specifies the value that will be returned
  - Its expression must conform to the return type

Calling a Method

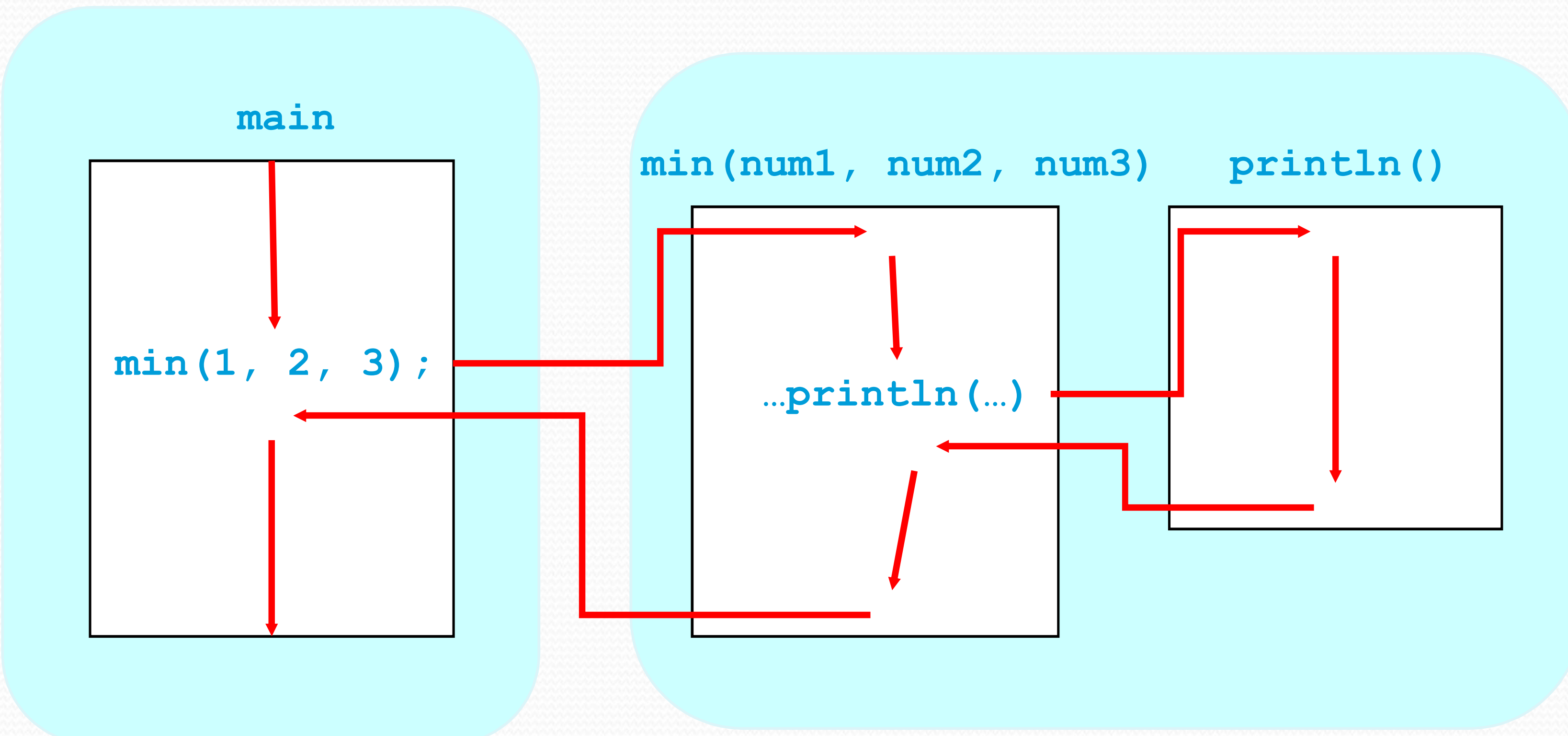
- Each time a method is called, the values of the *actual arguments* in the invocation are assigned to the *formal arguments*

```
int  num = min (2, 3);
```

```
static int min (int num1, int num2)
{
   int minValue = (num1 < num2 ? num1 : num2);
   return minValue;
}
```

Method Control Flow

- A method can call another method, who can call another method, …



```java
public class ExampleMinNumber {

  public static void main(String[] args) {
    int a = 11;
    int b = 6;
    int c = minFunction(a, b);
    System.out.println("Minimum Value = " + c);
  }

  /** returns the minimum of two numbers */
  public static int minFunction(int n1, int n2) {
    int min;
    if (n1 > n2)
      min = n2;
    else
      min = n1;

    return min;
  }
}
```

Method Overloading

Method Overloading

- Sometimes you want to have a multiple methods with the same name be able to do different operations on different parameters.
  - Java allows this through a process called overloading.
    - Overloading is having multiple methods in the same class with the same name, but accept different types of parameters.
  - For instance:

```
public double add(double num1, double num2) {
        return num1 + num2;
}

public String add(String str1, String str2) {
        return str1 + str2;
}
```

  - Even though both of these methods are named `add`, they perform different operations on different parameters.

Method Overloading

- When we call a method, the compiler must determine which of the methods to use through a process called <u>binding</u>.
  - Java binds methods by matching a method's <u>signature</u> to how it is called.
    - A method's signature consists of its name and the data types of its parameters.
    - The signatures of the two previous methods are:
      - add(double, double)
      - add(String, String)
    - So the java compiler can tell which method to used based on how it was called.

Method Overloading

- A class may define multiple methods with the same name---this is called method overloading
  - usually perform the same task on different data types

- Example: The `PrintStream` class defines multiple `println` methods, i.e., `println` is overloaded:

```
println (String s)
println (int i)
println (double d)
    …
```

- The following lines use the `System.out.print` method for different data types:

```
System.out.println ("The total is:");
double total = 0;
System.out.println (total);
```

Method Overloading: Signature

- The compiler must be able to determine which version of the method is being invoked
- This is by analyzing the parameters, which form the *signature* of a method
  - the signature includes the type and order of the parameters
    - if multiple methods match a method call, the compiler picks the best match
    - if none matches exactly but some implicit conversion can be done to match a method, then the method is invoke with implicit conversion.
  - the return type of the method is not part of the signature

```java
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

public class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```

Java & Inner Classes

Kinds of nested/inner classes

- Inner class
  - defined inside another class
  - but each instance of an inner class is transparently associated with an instance of the outer class
  - method invocations can be transparently redirected to outer instance
- Anonymous inner classes
  - unnamed inner classes
- Nested class
  - defined inside another class
  - has access to private members of enclosing class
  - But just a normal class

Inner Classes

- Description
  - Class defined in scope of another class
- Property
  - Can directly access all variables & methods of enclosing class (including private fields & methods)
- Example

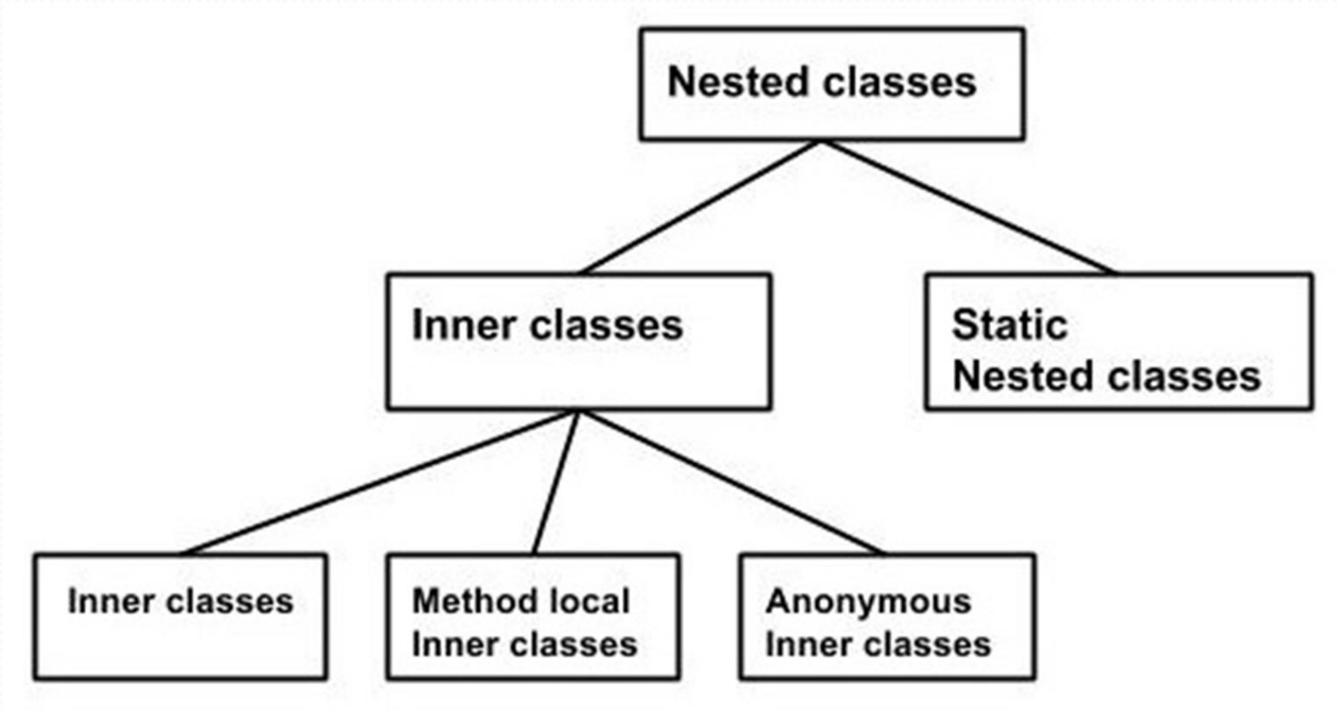
```
public class OuterClass {
    public class InnerClass {
        ...
    }
}
```

Inner Classes

- May be named or anonymous
- Useful for
  - Logical grouping of functionality
  - Data hiding
  - Linkage to outer class
- Examples
  - Iterator for Java Collections
  - ActionListener for Java GUI widgets

Inner Classes

- Inner class instance
  - Has association to an instance of outer class
  - Must be instantiated with an enclosing instance
  - Is tied to outer class object at moment of creation (can not be changed)



Anonymous Inner Class

- Doesn't name the class
- inner class defined at the place where you create an instance of it (in the middle of a method)
  - Useful if the only thing you want to do with an inner class is create instances of it in one location
- In addition to referring to fields/methods of the outer class, can refer to final local variables

Syntax for anonymous inner classes

- use

```
new Foo() {
    public int one() { return 1; }
    public int add(int x, int y) { return x+y; }
};
```

- to define an anonymous inner class that:
  - extends class Foo
  - defines methods one and add

MyList without anonymous inner class

- Code

```
public class MyList implements Iterable {
    private Object [ ] a;
    private int size;
    public Iterator iterator() {
     return new MyIterator();
     }
    public class MyIterator implements Iterator {
        private int pos = 0;
        public boolean hasNext() { return pos < size; }
        public Object next()   { return a[pos++]; }
    }
}
```

MyList with anonymous inner class

- Code

```
public class MyList implements Iterable {
   private Object [ ] a;
   private int size;
   public Iterator iterator() {
    return new Iterator () {
       private int pos = 0;
       public boolean hasNext() { return pos < size; }
       public Object next()   { return a[pos++]; }
   }
}
```

Nested class

- Declared like a standard inner class, except you say "static class" rather than "class".
- For example:

```
class LinkedList {
  static class Node {
    Object head;
    Node tail;
    }
  Node head;
  }
```

Nested classes

- An instance of an inner class does not contain an implicit reference to an instance of the outer class
- Still defined within outer class, has access to all the private fields
- Use if inner object might be associated with different outer objects, or survive longer than the outer object
  - Or just don't want the overhead of the extra pointer in each instance of the inner object

```java
class Outer_Demo {
  int num;

  // inner class
  private class Inner_Demo {
    public void print() {
      System.out.println("This is an inner class");
    }
  }

  // Accessing he inner class from the method within
  void display_Inner() {
    Inner_Demo inner = new Inner_Demo();
    inner.print();
  }
}

public class My_class {

  public static void main(String args[]) {
    // Instantiating the outer class
    Outer_Demo outer = new Outer_Demo();

    // Accessing the display_Inner() method.
    outer.display_Inner();
  }
}
```

The String Class

String class facts

- An object of the String class represents a string of characters.
- The String class belongs to the java.lang package, which does not require an import statement.
- Like other classes, String has constructors and methods.
- Unlike other classes, String has two operators, + and += (used for concatenation).

Literal Strings

- are anonymous objects of the String class
- are defined by enclosing text in double quotes.
  “This is a literal String”
- don’t have to be constructed.
- can be assigned to String variables.
- can be passed to methods and constructors as parameters.
- have methods you can call.

Literal String examples

```
//assign a literal to a String variable
String name = “Robert”;

//calling a method on a literal String
char firstInitial = “Robert”.charAt(0);

//calling a method on a String variable
char firstInitial = name.charAt(0);
```

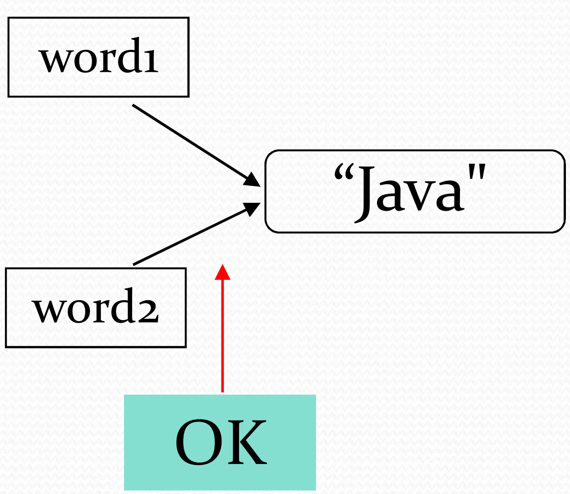
Immutability

- Once created, a string cannot be changed: none of its methods changes the string.
- Such objects are called *immutable*.
- Immutable objects are convenient because several references can point to the same object safely: there is no danger of changing an object through one reference without the others being aware of the change.

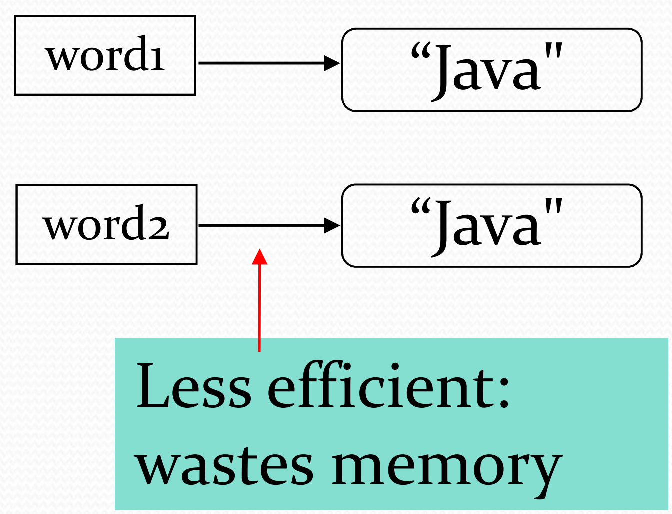
Advantages Of Immutability

Uses less memory.

```
String word1 = "Java";
String word2 = word1;
```



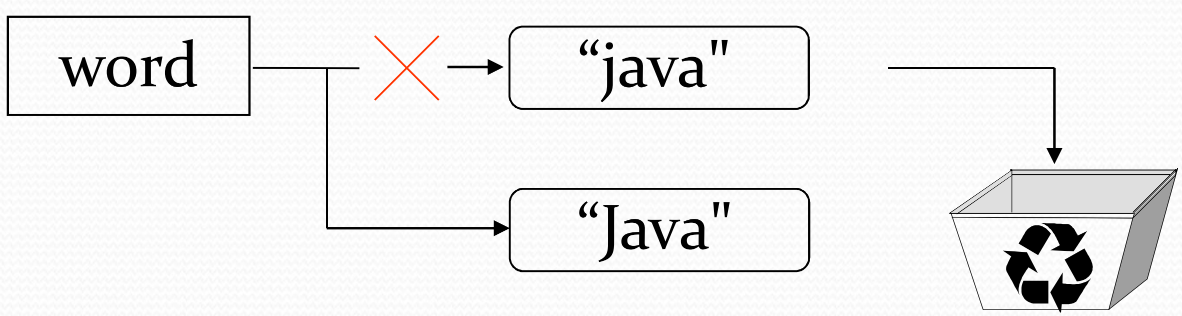

```
String word1 = “Java";
String word2 = new String(word1);
```



Disadvantages of Immutability

Less efficient — you need to create a new string and throw away the old one even for small changes.

```
String word = "Java";
char ch = Character.toUpperCase(word.charAt (0));
word =  ch + word.substring (1);
```



Empty Strings

- An empty String has no characters. It's length is 0.

```
String word1 = "";
String word2 = new String();
```

Empty strings

- Not the same as an uninitialized String.

```
private String errorMsg;
```

errorMsg is **null**

No Argument Constructors

- No-argument constructor creates an empty String. Rarely used.
- A more common approach is to reassign the variable to an empty literal String. (Often done to reinitialize a variable used to store input.)

```
String empty = new String();
```

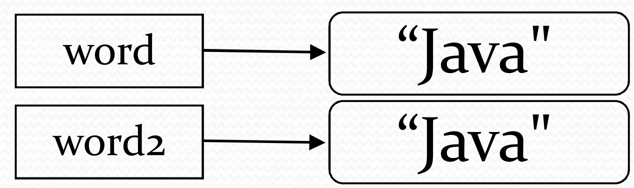
```
String empty = “”;//nothing between quotes
```

Copy Constructors

- Copy constructor creates a copy of an existing String. Also rarely used.
- Not the same as an assignment.

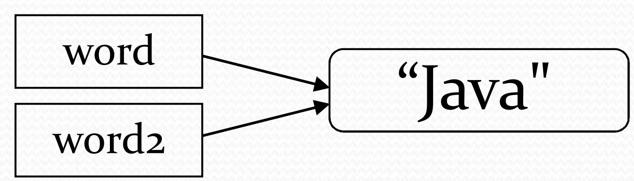
Copy Constructor: Each variable points to a different copy of the String.

```
String word = new String("Java");
String word2 = new String(word);
```



Assignment: Both variables point to the same String.

```
String word = "Java";
String word2 = word;
```



Other Constructors

Most other constructors take an array as a parameter to create a String.

```
char[] letters = {‘J’, ‘a’, ‘v’, ‘a’};
String word = new String(letters);//”Java”
```

Methods — length, charAt

int length(); — Returns the number of characters in the string

char charAt(i); — Returns the char at position i.

Character positions in strings are numbered starting from 0 – just like arrays.

Returns:

```
"Problem".length();      → 7
"Window".charAt (2);     → 'n'
```

Methods — substring

Returns a new String by copying characters from an existing String.

- String subs = word.**substring** (i, k);
  - returns the substring of chars in positions from **i** to **k-1**

`television`

i *k*

- String subs = word.**substring** (i);
  - returns the substring from the **i**-th char to the end

`television`

i

Returns:

”television".substring (2,5);	“lev"
“immutable".substring (2);	“mutable"
“bob".substring (9);	"" (empty string)

Methods — Concatenation

String word1 = “re”, word2 = “think”; word3 = “ing”;

int num = 2;

- String result = word1 + word2;

  //concatenates word1 and word2 “rethink“

- String result = word1.**concat** (word2);

  //the same as word1 + word2 “rethink“

- **result** += word3;

  //concatenates word3 to result “rethinking”

- **result** += num; //converts num to String
  //and concatenates it to result “rethinking2”

Methods — Find (indexOf)

0 2 6 10 15

String name ="President George Washington";

	Returns:
date.indexOf ('P');	0
date.indexOf ('e');	2
date.indexOf ("George");	10
date.indexOf ('e', 3);	6
date.indexOf ("Bob");	−1
date.lastIndexOf ('e');	15

(starts searching at position 3)

(not found)

Methods — Equality

boolean b = word1.**equals**(word2);

returns **true** if the string **word1** is equal to **word2**

boolean b = word1.**equalsIgnoreCase**(word2);

returns **true** if the string **word1** matches **word2**, case-blind

```
b = “Raiders”.equals(“Raiders”);//true
b = “Raiders”.equals(“raiders”);//false
b = “Raiders”.equalsIgnoreCase(“raiders”);//true
```

```
if(team.equalsIgnoreCase(“raiders”))
      System.out.println(“Go You “ + team);
```

Methods — Comparisons

int diff = word1.**compareTo**(word2);
returns the "difference" **word1 – word2**

int diff = word1.**compareToIgnoreCase**(word2);
returns the "difference" **word1 – word2**, case-blind

Usually programmers don't care what the numerical "difference" of **word1 - word2** is, just whether the difference is negative (word1 comes before word2), zero (word1 and word2 are equal) or positive (word1 comes after word2). Often used in conditional statements.

```
if(word1.compareTo(word2) > 0){
        //word1 comes after word2…
}
```

Comparison Examples

```
//negative differences
diff = “apple”.compareTo(“berry”);//a before b
diff = “Zebra”.compareTo(“apple”);//Z before a
diff = “dig”.compareTo(“dug”);//i before u
diff = “dig”.compareTo(“digs”);//dig is shorter
```

```
//zero differences
diff = “apple”.compareTo(“apple”);//equal
diff = “dig”.compareToIgnoreCase(“DIG”);//equal
```

```
//positive differences
diff = “berry”.compareTo(“apple”);//b after a
diff = “apple”.compareTo(“Apple”);//a after A
diff = “BIT”.compareTo(“BIG”);//T after G
diff = “huge”.compareTo(“hug”);//huge is longer
```

Methods — trim

String word2 = **word1.trim** ();

returns a new string formed from **word1** by removing white space at both ends

does not affect whites space in the middle

```
String word1 = " Hi Bob ";
String word2 = word1.trim();
//word2 is "Hi Bob" – no spaces on either end
//word1 is still " Hi Bob " – with spaces
```

Methods — replace

String word2 = word1.**replace**(oldCh, newCh);

returns a new string formed from **word1** by replacing all occurrences of **oldCh** with **newCh**

```
String word1 = “rare“;
String word2 = “rare“.replace(‘r’, ‘d’);
//word2 is “dade”, but word1 is still “rare“
```

Methods — Changing Case

String word2 = word1.**toUpperCase**();
String word3 = word1.**toLowerCase**();

returns a new string formed from **word1** by converting its characters to upper (lower) case

```
String word1 = “HeLLo“;
String word2 = word1.toUpperCase();//”HELLO”
String word3 = word1.toLowerCase();//”hello”
//word1 is still “HeLLo“
```

Replacements

- Example: to “convert” word1 to upper case, replace the reference with a new reference.

```
word1 = word1.toUpperCase();
```

- A common bug:

```
word1.toUpperCase();
```

word1 remains unchanged

Numbers to Strings

Three ways to convert a number into a string:

1. String s = "" + num;

```
s = “” + 123;//”123”
```

2. String s = Integer.toString (i); ←

   String s = Double.toString (d);←

```
s = Integer.toString(123);//”123”
s = Double.toString(3.14); //”3.14”
```

3. String s = String.valueOf (num);

```
s = String.valueOf(123);//”123”
```

Commandline Arguments

Command-line arguments in Java are **used** to pass **arguments** to the main program. If you look at the **Java** main method syntax, it accepts String array as an **argument**. When we pass **command-line arguments**, they are treated as strings and passed to the main function in the string array **argument**.

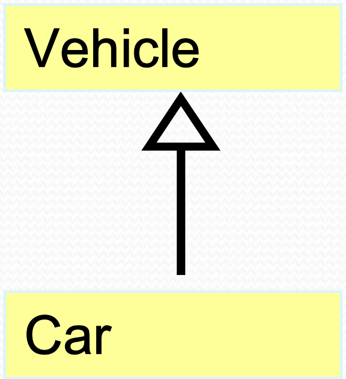
Inheritance

Inheritance

- *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*
- 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

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



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

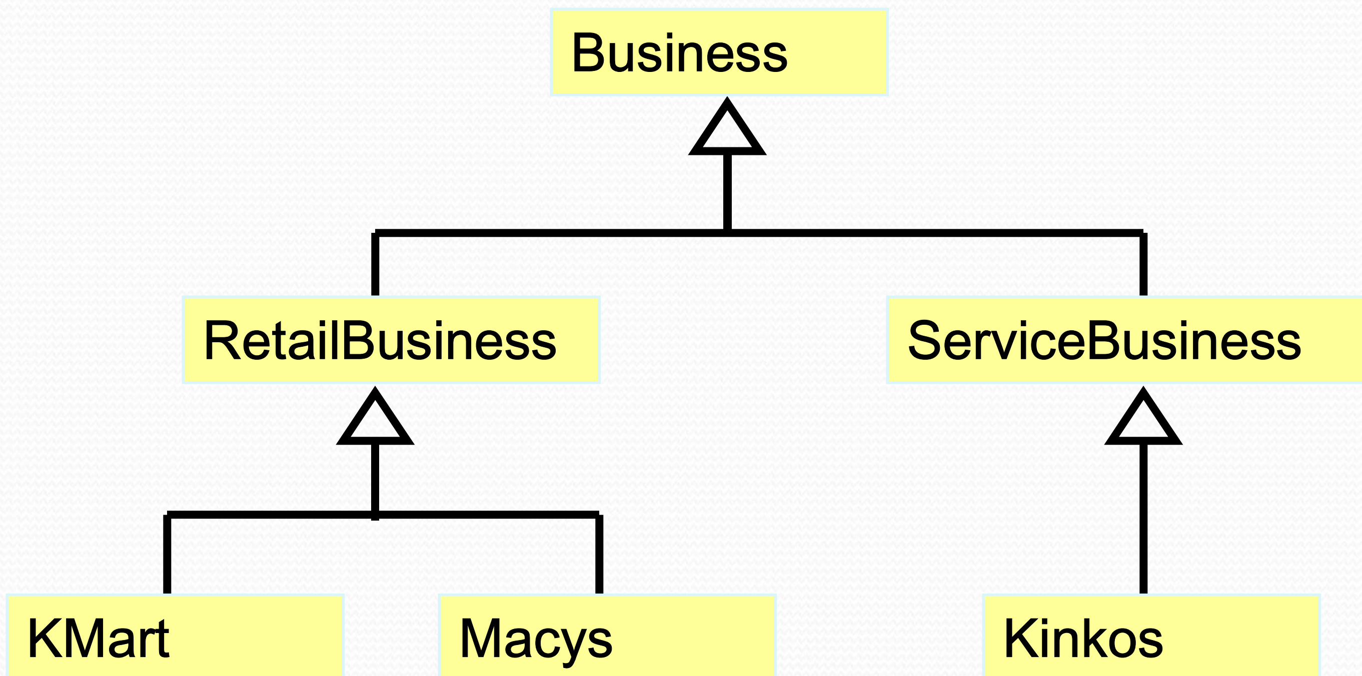
Deriving Subclasses

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

```
class Car extends Vehicle
  {
     // class contents
  }
```

Class Hierarchies

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



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its ancestor classes
- There is no single class hierarchy that is appropriate for all situations

```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
            System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
            PhysicsTeacher obj = new PhysicsTeacher();
            System.out.println(obj.collegeName);
            System.out.println(obj.designation);
            System.out.println(obj.mainSubject);
            obj.does();
   }
}
```

The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child

- Protected visibility provides more encapsulation than public visibility 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 constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

The super Reference

- A child’s constructor is responsible for calling the parent’s constructor
- The first line of a child’s constructor should use the `super` reference to call the parent’s constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent’s class

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

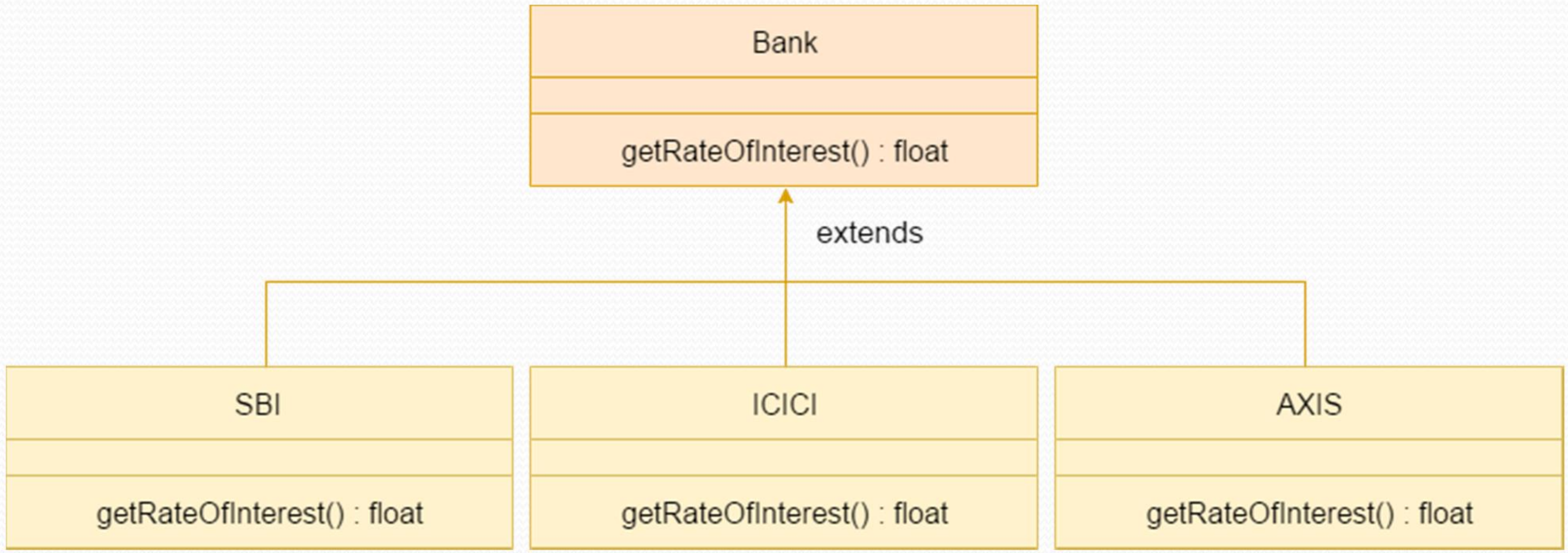
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.

- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

```java
//Creating a parent class.
class Vehicle{
  //defining a method
  void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```


Bank
getRateOfInterest() : float
extends
SBI
getRateOfInterest() : float
ICICI
getRateOfInterest() : float
AXIS
getRateOfInterest() : float

```
class Bank
{
int getRateOfInterest()
{
return 0;
}
}

class SBI extends Bank
{
int getRateOfInterest(){
return 8;
}
}
class ICICI extends Bank
{
int getRateOfInterest(){
return 7;
}
}
class AXIS extends Bank
{
int getRateOfInterest(){
return 9;
}
}
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

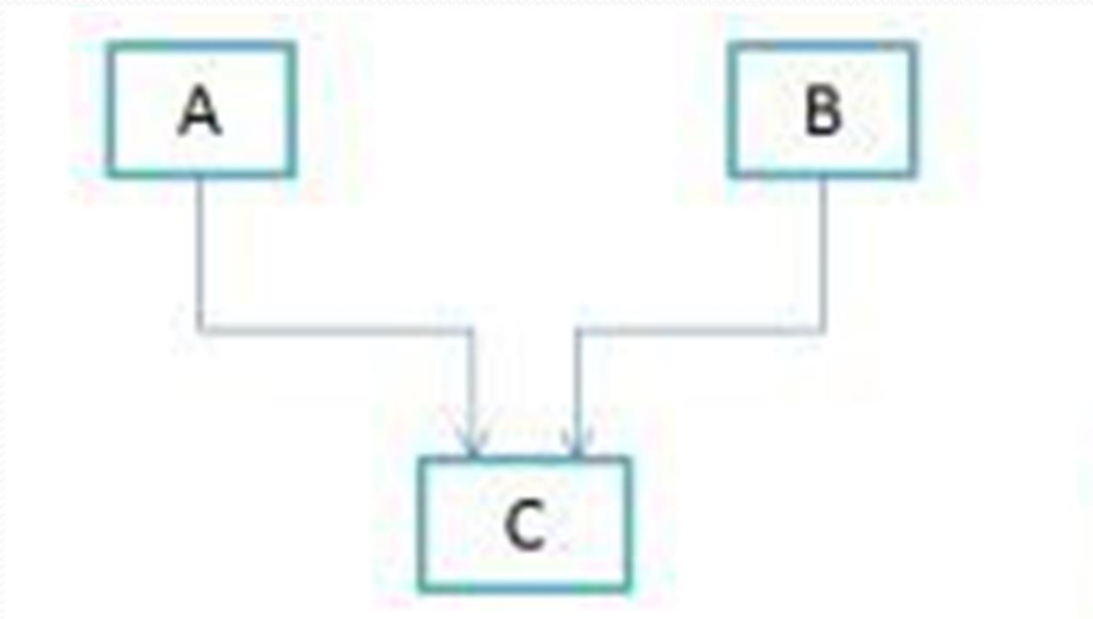
No.	**Method Overloading**	**Method Overriding**
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Multiple and Multilevel Inheritance

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
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

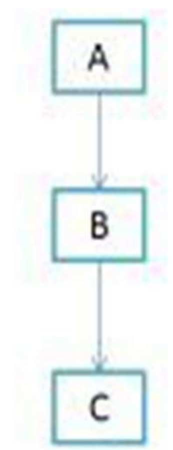
"Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class.



Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Multilevel Inheritance

- **Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.
- In the flow diagram C is subclass or child class of B and B is a child class of A.



```
class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
class Y extends X
{
public void methodY()
{
System.out.println("Class Y method");
}
}
class Z extends Y
{
  public void methodZ()
  {
    System.out.println("Class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
 }
}
```

The Object Class

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
- Therefore, the `Object` class is the ultimate root of all class hierarchies

- 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 an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information
- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Whatever
{
   // contents
}
```

- An abstract class often contains abstract methods with no definitions (like an interface does)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- An abstract class typically contains non-abstract methods (with bodies), further distinguishing abstract classes from interfaces
- A class declared as abstract does not need to contain abstract methods

- 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` (because it must be overridden) or `static` (because it has no definition yet)
- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate

```
//abstract parent class
abstract class Animal{
   //abstract method
   public abstract void sound();
}
//Dog class extends Animal class
public class Dog extends Animal{

   public void sound(){
		System.out.println("Woof");
   }
   public static void main(String args[]){
		Animal obj = new Dog();
		obj.sound();
   }
}
```

Packages

Introduction

- The main feature of OOP is its ability to support the reuse of code:
  - Extending the classes (via inheritance)
  - Extending interfaces
- The features in basic form limited to reusing the classes within a program.
- What if we need to use classes from other programs without physically copying them into the program under development ?
- In Java, this is achieved by using what is known as "packages", a concept similar to "class libraries" in other languages.

Packages

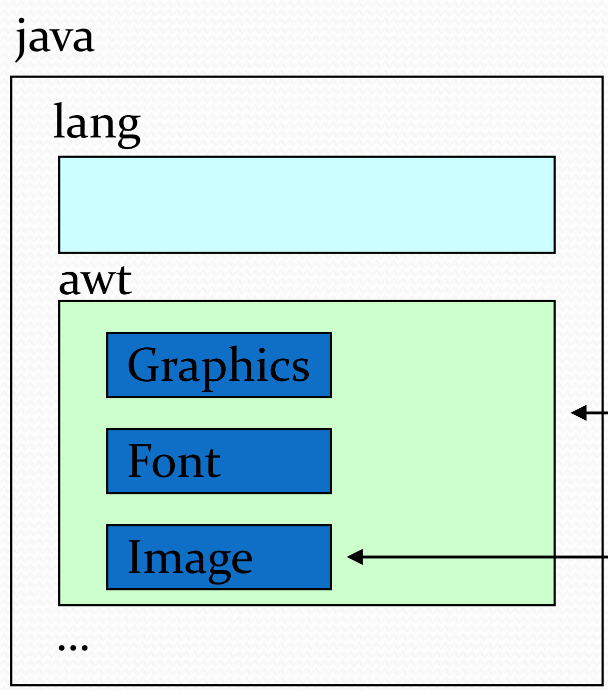
- Packages are Java's way of grouping a number of related classes and/or interfaces together into a single unit. That means, packages act as "containers" for classes.
- The benefits of organising classes into packages are:
  - The classes contained in the packages of other programs/applications can be reused.
  - In packages classes can be unique compared with classes in other packages. That two classes in two different packages can have the same name. If there is a naming clash, then classes can be accessed with their fully qualified name.
  - Classes in packages can be hidden if we don't want other packages to access them.
  - Packages also provide a way for separating "design" from coding.

Java Foundation Packages

- Java provides a large number of classes grouped into different packages based on their functionality.
- The six foundation Java packages are:
  - java.lang
    - Contains classes for primitive types, strings, math functions, threads, and exception
  - java.util
    - Contains classes such as vectors, hash tables, date etc.
  - java.io
    - Stream classes for I/O
  - java.awt
    - Classes for implementing GUI – windows, buttons, menus etc.
  - java.net
    - Classes for networking
  - java.applet
    - Classes for creating and implementing applets

Using System Packages

- The packages are organised in a hierarchical structure. For example, a package named "java" contains the package "awt", which in turn contains various classes required for implementing GUI (graphical user interface).



"java" Package containing "lang", "awt",.. packages; Can also contain classes.

awt Package containing classes

Classes containing methods

Accessing Classes from Packages

- There are two ways of accessing the classes stored in packages:
  - Using fully qualified class name
    - java.lang.Math.sqrt(x);
  - Import package and use class name directly.
    - import java.lang.Math
    - Math.sqrt(x);
- Selected or all classes in packages can be imported:

```
import package.class;
import package.*;
```

- Implicit in all programs: import java.lang.*;
- package statement(s) must appear first

Creating Packages

- Java supports a keyword called “package” for creating user-defined packages. The package statement must be the first statement in a Java source file (except comments and white spaces) followed by one or more classes.

```
package myPackage;
public class ClassA {
        // class body
}
class ClassB {
 // class body
}
```

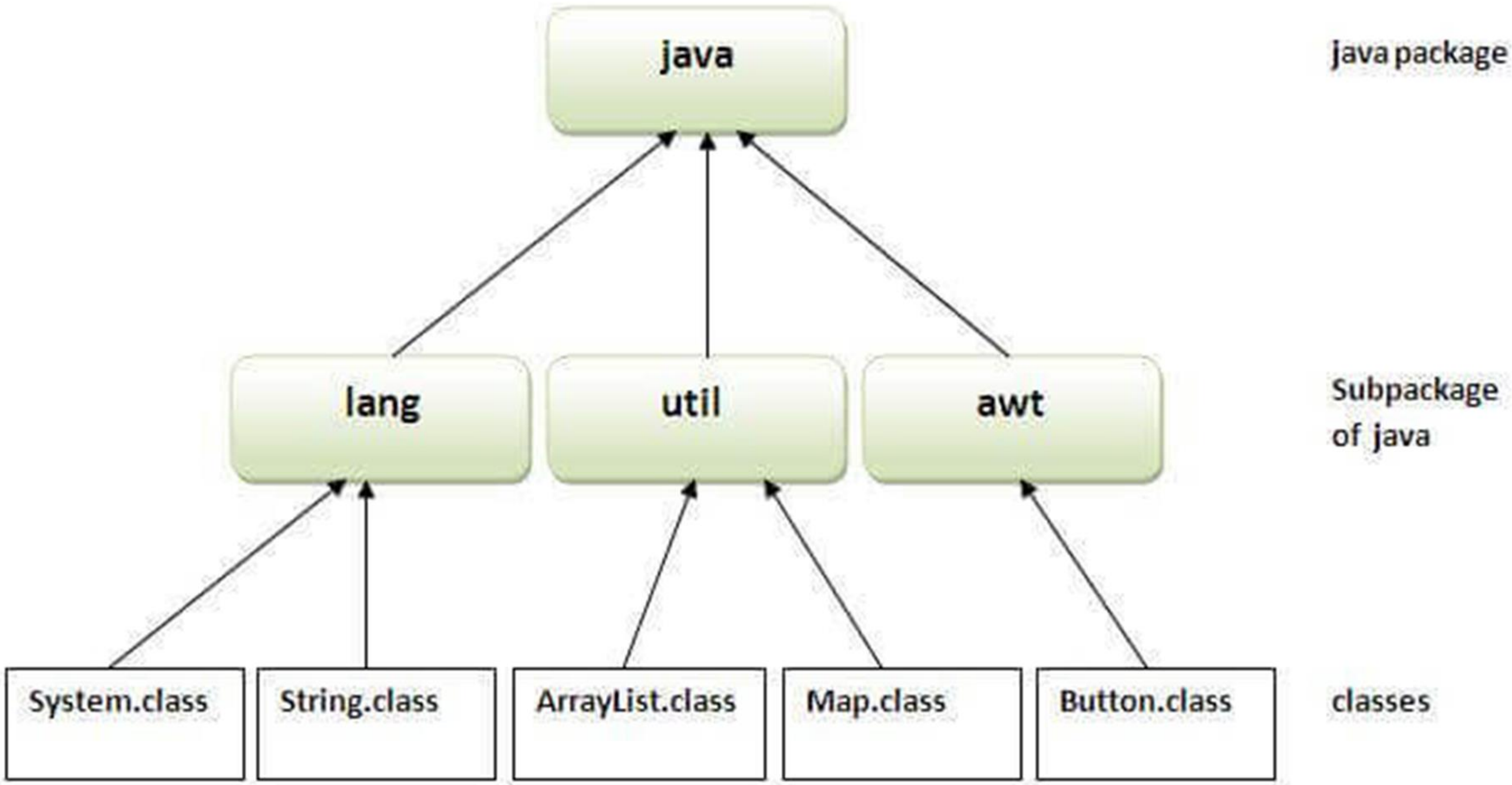
- Package name is “myPackage” and classes are considred as part of this package; The code is saved in a file called “ClassA.java” and located in a directory called “myPackage”.

Creating Sub Packages

- Classes in one ore more source files can be part of the same packages.
- As packages in Java are organised hierarchically, sub-packages can be created as follows:
  - package myPackage.Math
  - package myPackage.secondPakage.thirdPackage
- Store “thirdPackage” in a subdirectory named “myPackage\secondPackage”. Store “secondPackage” and “Math” class in a subdirectory “myPackage”.

Accessing a Package

- As indicated earlier, classes in packages can be accessed using a fully qualified name or using a short-cut as long as we import a corresponding package.
- The general form of importing package is:
  - import package1[.package2][…].classname
  - Example:
    - import myPackage.ClassA;
    - import myPackage.secondPackage
  - All classes/packages from higher-level package can be imported as follows:
    - import myPackage.*;


java
java package
lang
util
awt
Subpackage
of java
System.class
String.class
ArrayList.class
Map.class
Button.class
classes

```java
package letmecalculate;

public class Calculator {
  public int add(int a, int b){
        return a+b;
  }
  public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
  }
}
```

```java
import letmecalculate.Calculator;
public class Demo{
  public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
  }
}
```

Protection and Packages

- All classes (or interfaces) accessible to all others in the same package.
- Class declared public in one package is accessible within another. Non-public class is not
- Members of a class are accessible from a difference class, as long as they are not *private*
- *protected* members of a class in a package are accessible to subclasses in a different class

Visibility - Revisited

- *Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Visibility Modifiers

Accessible to:	public	protected	Package (default)	private
Same Class	Yes	Yes	Yes	Yes
Class in package	Yes	Yes	Yes	No
Subclass in different package	Yes	Yes	No	No
Non-subclass different package	Yes	No	No	No

Interfaces

Java Interface

- A Java *interface* is a collection of constants and abstract methods
  - abstract method: a method header without a method body; we declare an abstract method using the modifier `abstract`
  - since all methods in an interface are abstract, the `abstract` modifier is usually left off

- Methods in an interface have public visibility by default

Interface: Syntax

interface is a reserved word

```
public interface Doable
{
   public static final String NAME;

   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately
follows each method header

No method in an
interface has a definition (body)

Implementing an Interface

- A class formally implements an interface by
  - stating so in the class header in the `implements` clause
  - a class can implement multiple interfaces: the interfaces are listed in the implements clause, separated by commas

- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors

Implementing Interfaces

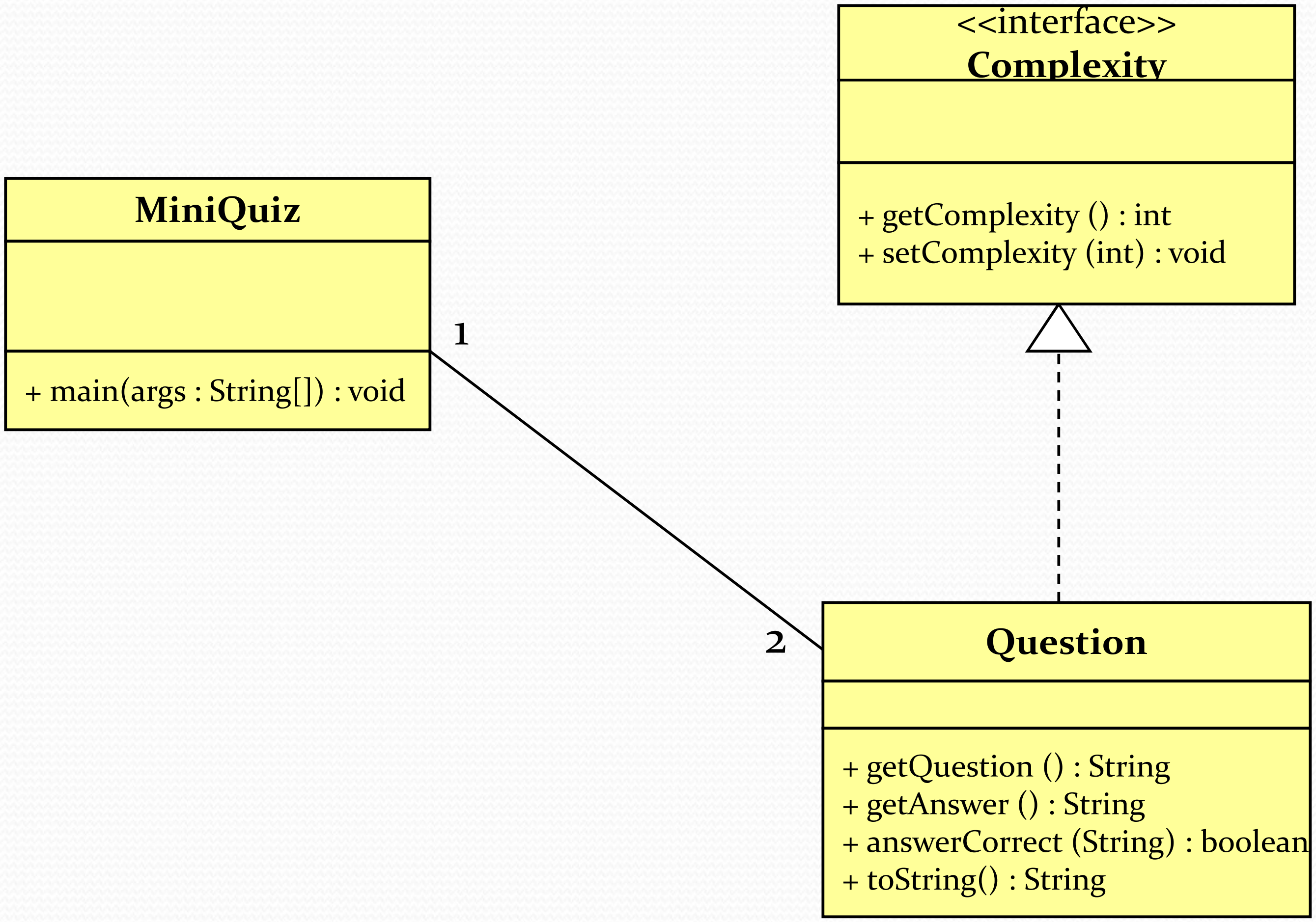
```
public class Something implements Doable
{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

```
public class ManyThings implements Doable, AnotherDoable
```

UML Diagram


<<interface>>
Complexity
+ getComplexity () : int
+ setComplexity (int) : void
MiniQuiz
+ main(args : String[]) : void
1
2
Question
+ getQuestion () : String
+ getAnswer () : String
+ answerCorrect (String) : boolean
+ toString() : String

Interfaces: Examples from Java Standard Class Library

- The Java Standard Class library defines many interfaces:
  - the `Iterator` interface contains methods that allow the user to move through a collection of objects easily
    - `hasNext(), next(), remove()`
  - the `Comparable` interface contains an abstract method called `compareTo`, which is used to compare two objects

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

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
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Exception Handling

Introduction

- Errors can be dealt with at place error occurs
  - Easy to see if proper error checking implemented
  - Harder to read application itself and see how code works
- Exception handling
  - Makes clear, robust, fault-tolerant programs
  - Java removes error handling code from "main line" of program
- Common failures
  - Memory exhaustion
  - Out of bounds array subscript
  - Division by zero
  - Invalid method parameters

Introduction

- Exception handling
  - Catch errors before they occur
  - Deals with synchronous errors (i.e., divide by zero)
  - Does not deal with asynchronous errors
    - Disk I/O completions, mouse clicks - use interrupt processing
  - Used when system can recover from error
    - Exception handler - recovery procedure
    - Error dealt with in different place than where it occurred
  - Useful when program cannot recover but must shut down cleanly

Introduction

- Exception handling
  - Should not be used for program control
    - Not optimized, can harm program performance
  - Improves fault-tolerance
    - Easier to write error-processing code
    - Specify what type of exceptions are to be caught
  - Another way to return control from a function or block of code

When Exception Handling Should Be Used

- Error handling used for
  - Processing exceptional situations
  - Processing exceptions for components that cannot handle them directly
  - Processing exceptions for widely used components (libraries, classes, methods) that should not process their own exceptions
  - Large projects that require uniform error processing

The Basics of Java Exception Handling

- Exception handling
  - Method detects error which it cannot deal with
    - *Throws* an exception
  - Exception handler
    - Code to *catch* exception and handle it
  - Exception only caught if handler exists
    - If exception not caught, block terminates

The Basics of Java Exception Handling

- Format
  - Enclose code that may have an error in **`try`** block
  - Follow with one or more **`catch`** blocks
    - Each **`catch`** block has an exception handler
  - If exception occurs and matches parameter in **`catch`** block
    - Code in catch block executed
  - If no exception thrown
    - Exception handling code skipped
    - Control resumes after **`catch`** blocks

```
try{
  code that may throw
exceptions
}
catch (ExceptionType ref) {
   exception handling code
}
```

The Basics of Java Exception Handling

- Termination model of exception handling
  - **`throw`** point
    - Place where exception occurred
    - Control cannot return to **`throw`** point
  - Block which threw exception expires
  - Possible to give information to exception handler

An Exception Handling Example: Divide by Zero

- Example program
  - User enters two integers to be divided
  - We want to catch division by zero errors
  - Exceptions
    - Objects derived from class **`Exception`**
  - Look in **`Exception`** classes in **`java.lang`**
    - Nothing appropriate for divide by zero
    - Closest is **`ArithmeticException`**
    - Extend and create our own exception class

Type of Exceptions

- There are two types of exceptions in Java
  - Unchecked exceptions
  - Checked exceptions

Unchecked Exceptions

- The compiler does not check to see if a method handles or throws these exceptions
  - Hence the name unchecked
- They need not be included in any method's throws list

Checked Exceptions

- The compiler checks whether these exceptions were handled in the method
- That must be included in a method’s throws list
- Compiler error occurs if these exceptions were not handled by the methods

Unchecked Exceptions Examples

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.

Checked Exceptions Examples

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

The try Statement

- To handle an exception in a program, the line that throws the exception is executed within a *try block*
- A try block is followed by one or more *catch* clauses
- Each catch clause has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type

The finally Clause

- A try statement can have an optional clause following the catch clauses, designated by the reserved word `finally`
- The statements in the finally clause always are executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

```
public class myclass {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Array Index Out of Bound");
    }
  }
}
```

Nested try Statements

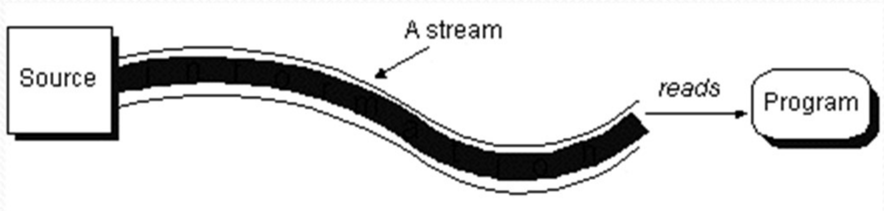
- ✓ A **try** statement can be inside the block of another try
- ✓ Each time a **try** statement is entered, the context of that exception is pushed on the stack
- ✓ If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- ✓ If a method call within a **try** block has **try** block within it, then then it is still nested **try**

```
class nestedtry {
        public static void main(String args[])
        {
                try {
                        int a[] = { 1, 2, 3, 4, 5 };
                        System.out.println(a[5]);
                        try {
                                int x = a[2] / 0;
                          }
                        catch (ArithmeticException e2) {
                                System.out.println("division by zero is not possible");
                        }
                }
                catch (ArrayIndexOutOfBoundsException e1) {
                        System.out.println("ArrayIndexOutOfBoundsException");
                        System.out.println("Element at such index does not exists");
                }
        }

}
```

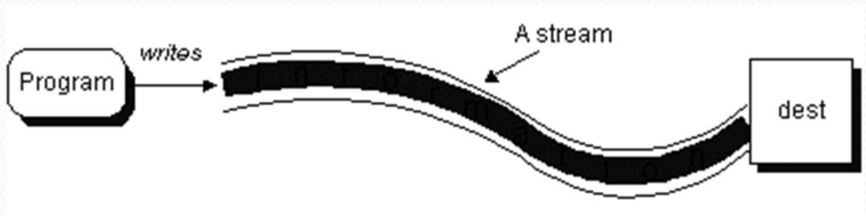
Java I/O Classes

Streams

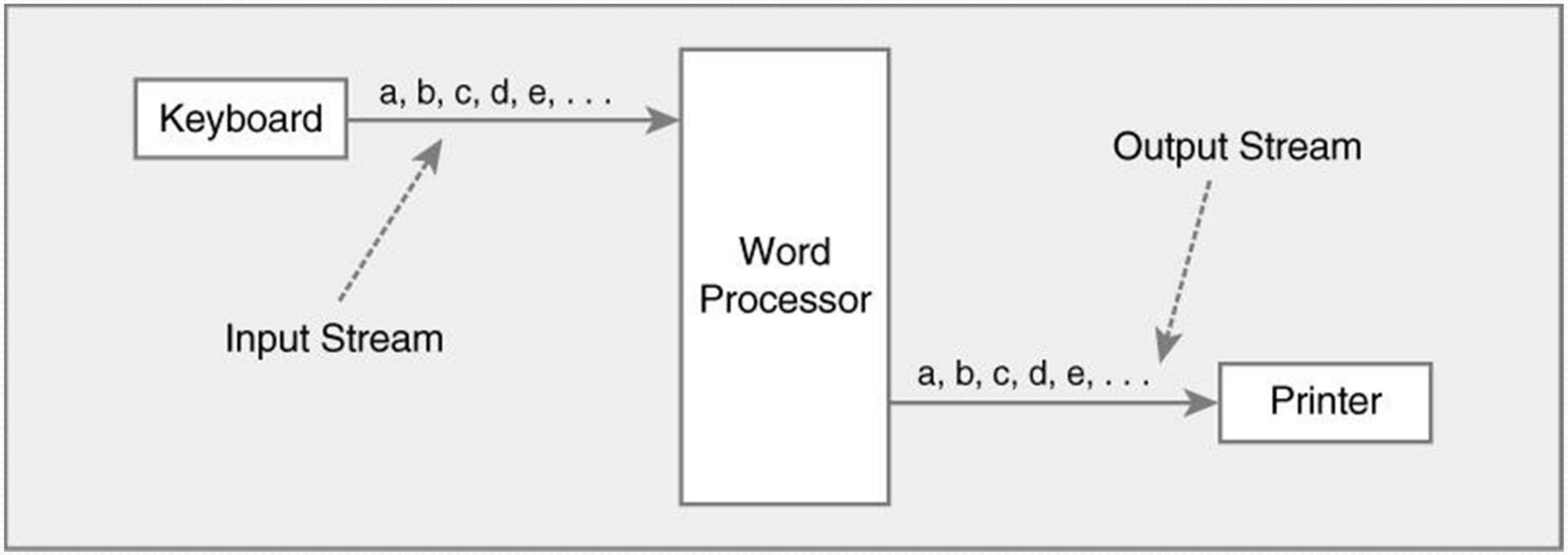
- To receive information, a program opens a stream to a “source” and reads the information:



- To send information, a program opens a stream to a destination (“sink”) and writes the information:



An Example


Keyboard
a, b, c, d, e, . . .
Input Stream
Word Processor
Output Stream
a, b, c, d, e, . . .
Printer

Streams

- Java provides many stream classes that let you work with data either
  - in the forms that you usually use (characters & numbers)
  - in low level byte form (8 bits at a time)
- Low level byte-oriented abstract classes
  - `InputStream` and `Outputstream`.
- Higher level character-based abstract classes
  - `Reader` and `Writer`

Using Streams

- No matter where the information is coming from or going to and no matter what type of data is being read or written, the algorithms for reading and writing data are pretty much always the same

Reading:

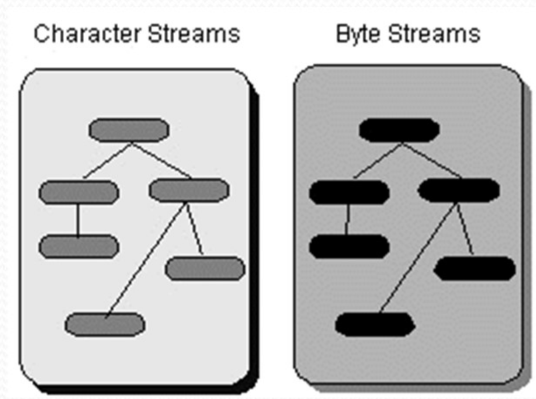
```
open a stream
while more information
    read information
close the stream
```

Writing:

```
open a stream
while more information
    write information
close the stream
```

java.io.*

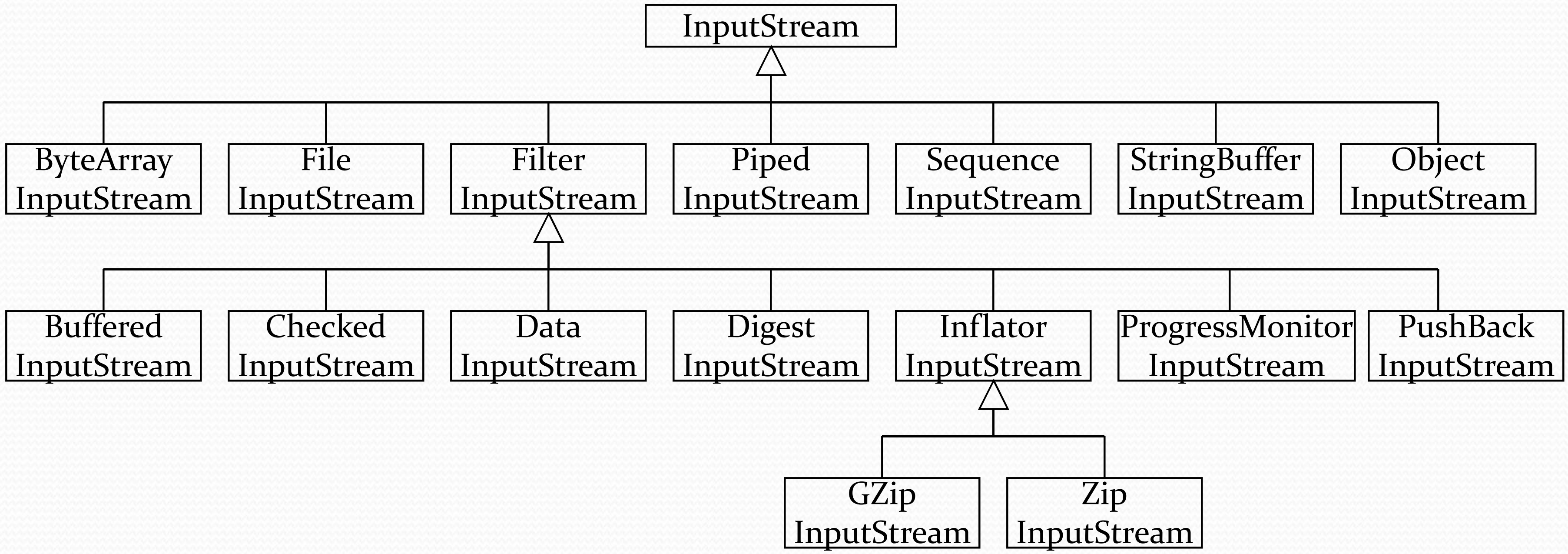
- The java.io package contains a collection of stream classes that support reading/writing from/to streams
- Streams are divided into two class hierarchies based on the type of data on which they operate.



Most Commonly Used Stream Classes

	Characters	Bytes
Files	`FileReader`	`FileInputStream`
	`FileWriter`	`FileOutputStream`
Buffering	`BufferedReader`	`BufferedInputStream`
	`BufferedWriter`	`BufferedOutputStream`
Printing	`PrintWriter`	`PrintStream`

Many choices!
Example: `InputStream` classes (byte-oriented input)



Input and Output Streams

`ByteArrayInputStream` `ByteArrayOutputStream`	Read or write a byte array.
`FileInputStream` `FileOutputStream`	Read or write data as bytes in a file.
`BufferedInputStream` `BufferedOutputStream`	Buffers the bytes in the underlying input or output stream.
`DataInputStream` `DataOutputStream`	A filter that allows the binary representation of Java primitive values (e.g., 'int' is 4 bytes) to be read or written by the specified underlying input or output stream.
`PushbackInputStream`	"Peek-a-boo" reader allows bytes to be "unread" from an underlying input stream.
`ObjectInputStream` `ObjectOutputStream`	Read or write binary representations of entire Java objects, using the underlying input or output stream.
`PipedInputStream` `PipedOutputStream`	Used in pairs by Java threads to communicate with each other.
`SequenceInputStream`	Concatenates several input streams.

Reader and Writer Streams

`CharArrayReader` `CharArrayWriter`	Read or write a character array.
`FileReader` `FileWriter`	Read or write characters in a file.
`BufferedReader` `BufferedWriter`	Buffers the bytes in the underlying Reader or Writer stream.
`StringReader` `StringWriter`	Read characters from a String, or write characters to a StringBuffer.
`PushbackReader`	"Peek-a-boo" reader allows characters to be "unread" from an underlying Reader. (Useful for writing parsers.)
`InputStreamReader` `OutputStreamReader`	Read or write characters in an underlying input or output stream. (e.g., like making a Reader out of an InputStream)
`PipedReader` `PipedWriter`	Used in pairs by Java threads for text-based communication with each other.
`LineNumberReader`	A BufferedReader that also keeps track of the number of lines read from the underlying Reader.

Multithreaded Programming

What are Threads?

- A piece of code that run in concurrent with other threads.
- Each thread is a ordered sequence of instructions.
- Threads are being extensively used express concurrency on both single and multiprocessors machines.
- Programming a task having multiple threads of control – Multithreading or Multithreaded Programming.

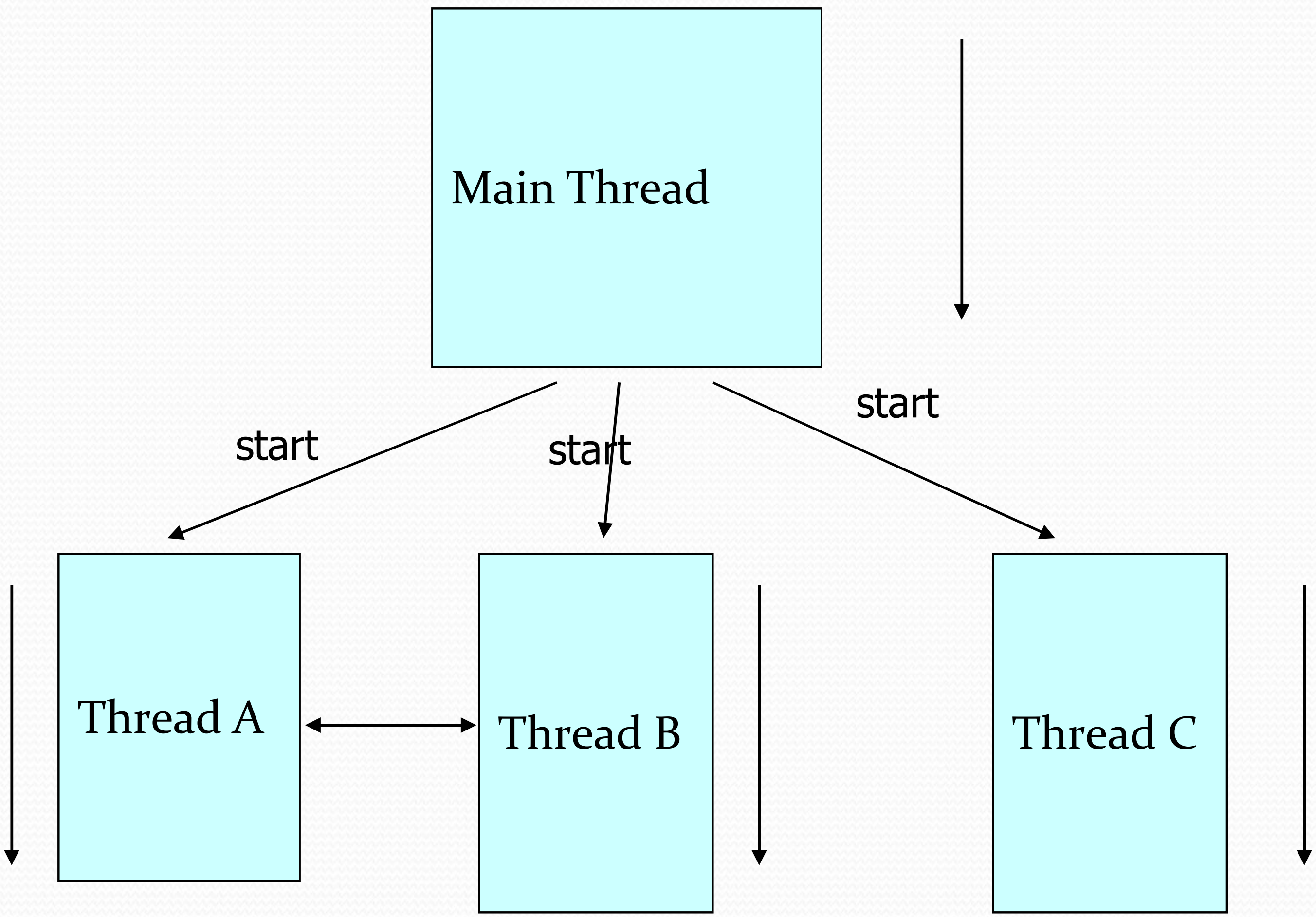
Java Threads

- 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.

Threads

- Threads are lightweight processes as the overhead of switching between threads is less
- They can be easily spawned
- The Java Virtual Machine spawns a thread when your program calls the Main Thread

A Multithreaded Program



Threads may switch or exchange data/results

A single threaded program

```
class ABC
{
....
     public void main(..)
     {
     ...
     ..
     }
}
```

begin

body

end

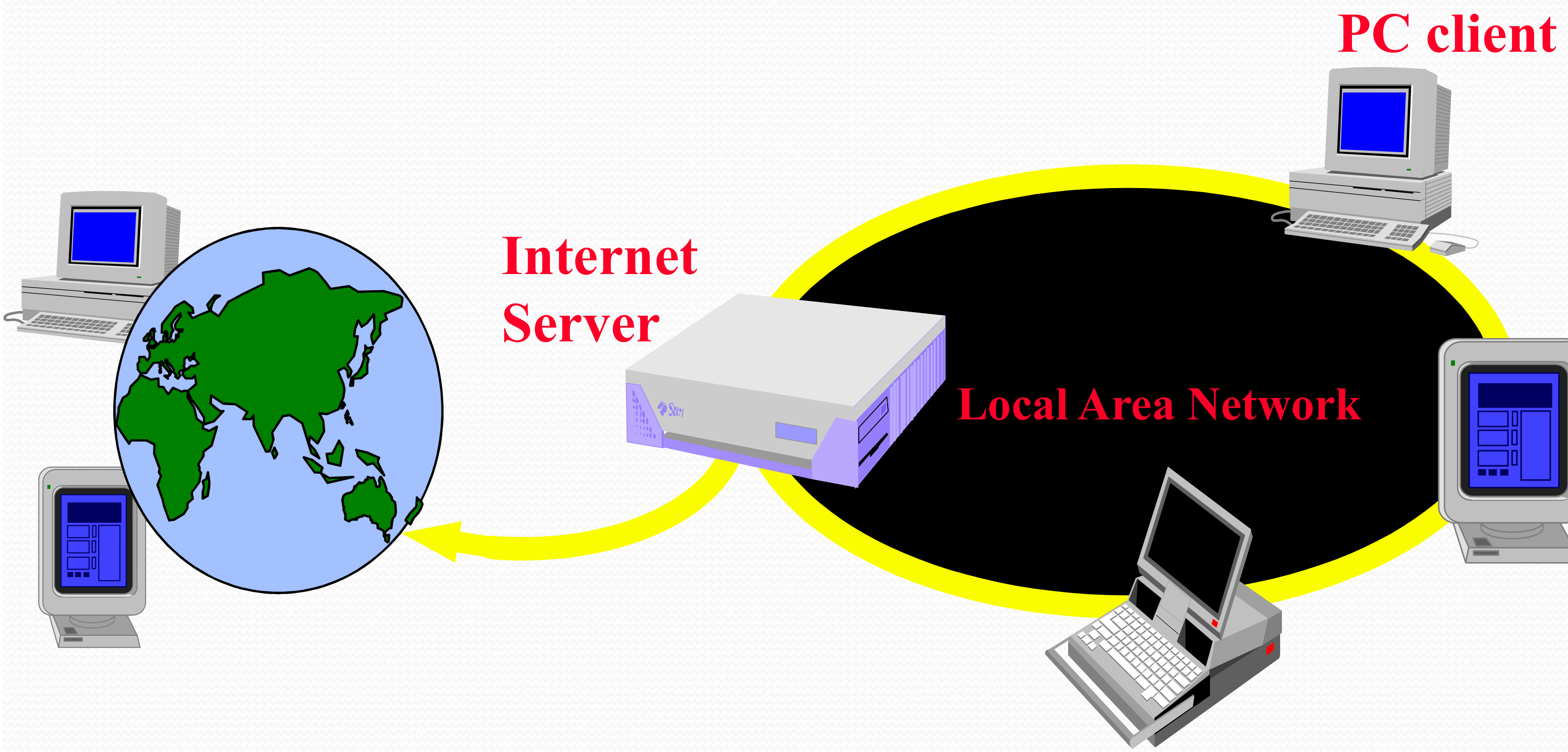
Scenario

- Consider a simple web server
- The web server listens for request and serves it
- If the web server was not multithreaded, the requests processing would be in a queue, thus increasing the response time and also might hang the server if there was a bad request.
- By implementing in a multithreaded environment, the web server can serve multiple request simultaneously thus improving response time

Why do we need threads?

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

Web/Internet Applications: Serving Many Users Simultaneously



Creating threads

- In java threads can be created by extending the Thread class or implementing the Runnable Interface
- It is more preferred to implement the Runnable Interface so that we can extend properties from other classes
- Implement the run() method which is the starting point for thread execution

Running threads

- Example

```
class mythread implements Runnable{
        public void run(){
                System.out.println("Thread Started");
        }
}

class mainclass {
        public static void main(String args[]){
                Thread  t = new Thread(new mythread()); // This is the way to instantiate a
                        thread implementing runnable interface
                t.start(); // starts the thread by running the run method
                }
}
```

- Calling t.run() does not start a thread, it is just a simple method call.
- Creating an object does not create a thread, calling start() method creates the thread.

Synchronization

- Synchronization prevent data corruption
- Synchronization allows only one thread to perform an operation on a object at a time.
- If multiple threads require an access to an object, synchronization helps in maintaining consistency.

Shared Resources

- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronising access to the data.
- Use "Synchronized" method:
  - public synchronized void update()
  - {
    - ...
  - }

Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
  - Java allows users to change priority:
    - ThreadName.setPriority(intNumber)
      - MIN_PRIORITY = 1
      - NORM_PRIORITY=5
      - MAX_PRIORITY=10

Accessing Shared Resources

- Applications Access to Shared Resources need to be coordinated.
  - Printer (two person jobs cannot be printed at the same time)
  - Simultaneous operations on your bank account
  - Can the following operations be done at the same time on the same account?
    - Deposit()
    - Withdraw()
    - Enquire()

Three threads example

```
class A extends Thread
{
        public void run()
          {
               for(int i=1;i<=5;i++)
                 {
                       System.out.println("\t From Thread A: i= "+i);
                 }
                 System.out.println("Exit from A");
          }
}

class B extends Thread
{
        public void run()
          {
               for(int j=1;j<=5;j++)
                 {
                       System.out.println("\t From Thread B: j= "+j);
                 }
                 System.out.println("Exit from B");
          }
}
```

```
class C extends Thread
{
        public void run()
          {
                for(int k=1;k<=5;k++)
                  {
                        System.out.println("\t From Thread C: k= "+k);
                  }

                   System.out.println("Exit from C");
          }
}

class ThreadTest
{
           public static void main(String args[])
            {
                      new A().start();
                      new B().start();
                      new C().start();
            }
}
```

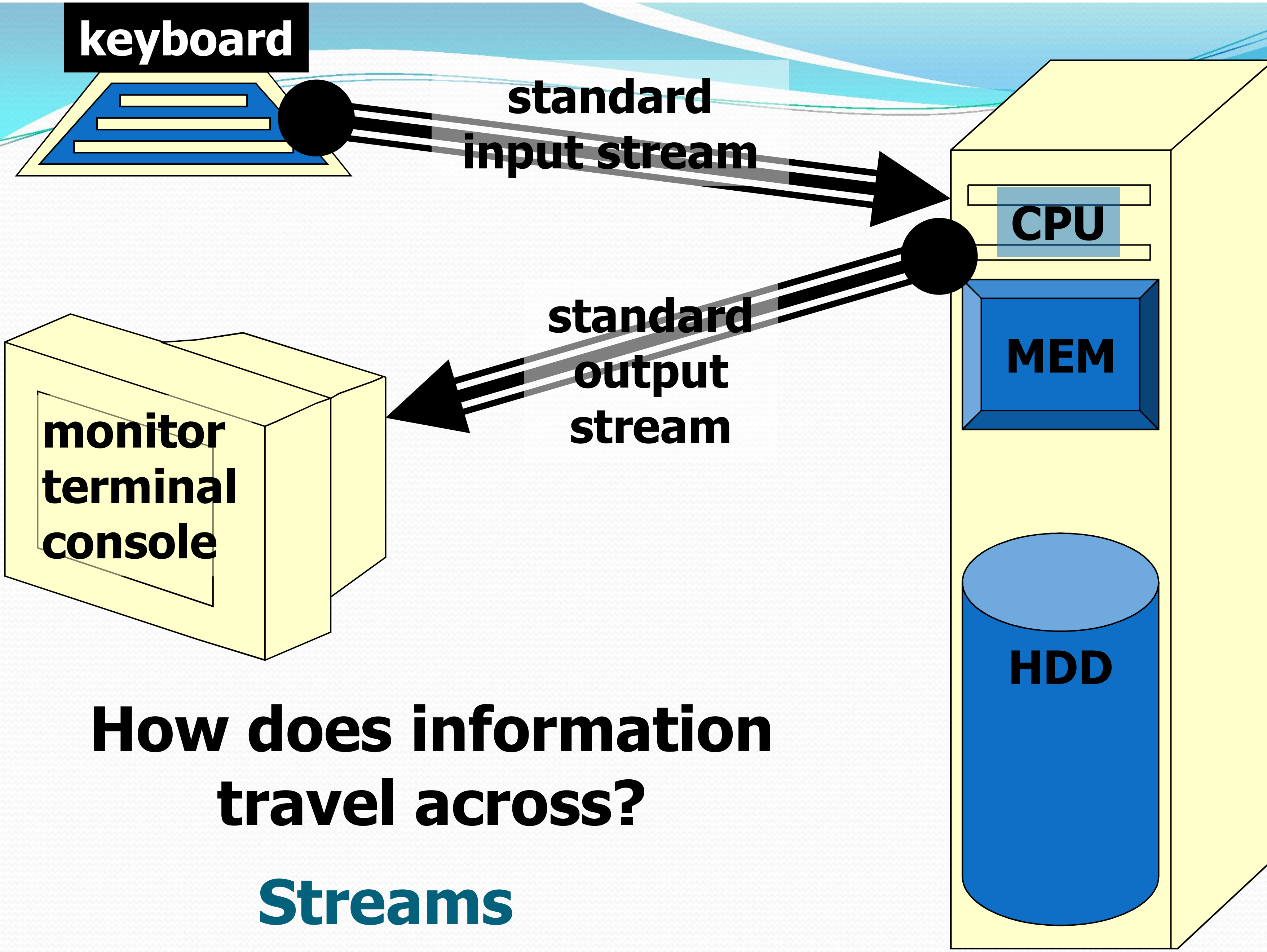
Run 1

From ThreadA: i= 1
From ThreadA: i= 2
From ThreadA: i= 3
From ThreadA: i= 4
From ThreadA: i= 5
Exit from A
From ThreadC: k= 1
From ThreadC: k= 2
From ThreadC: k= 3
From ThreadC: k= 4
From ThreadC: k= 5
Exit from C
From ThreadB: j= 1
From ThreadB: j= 2
From ThreadB: j= 3
From ThreadB: j= 4
From ThreadB: j= 5
Exit from B

Run2

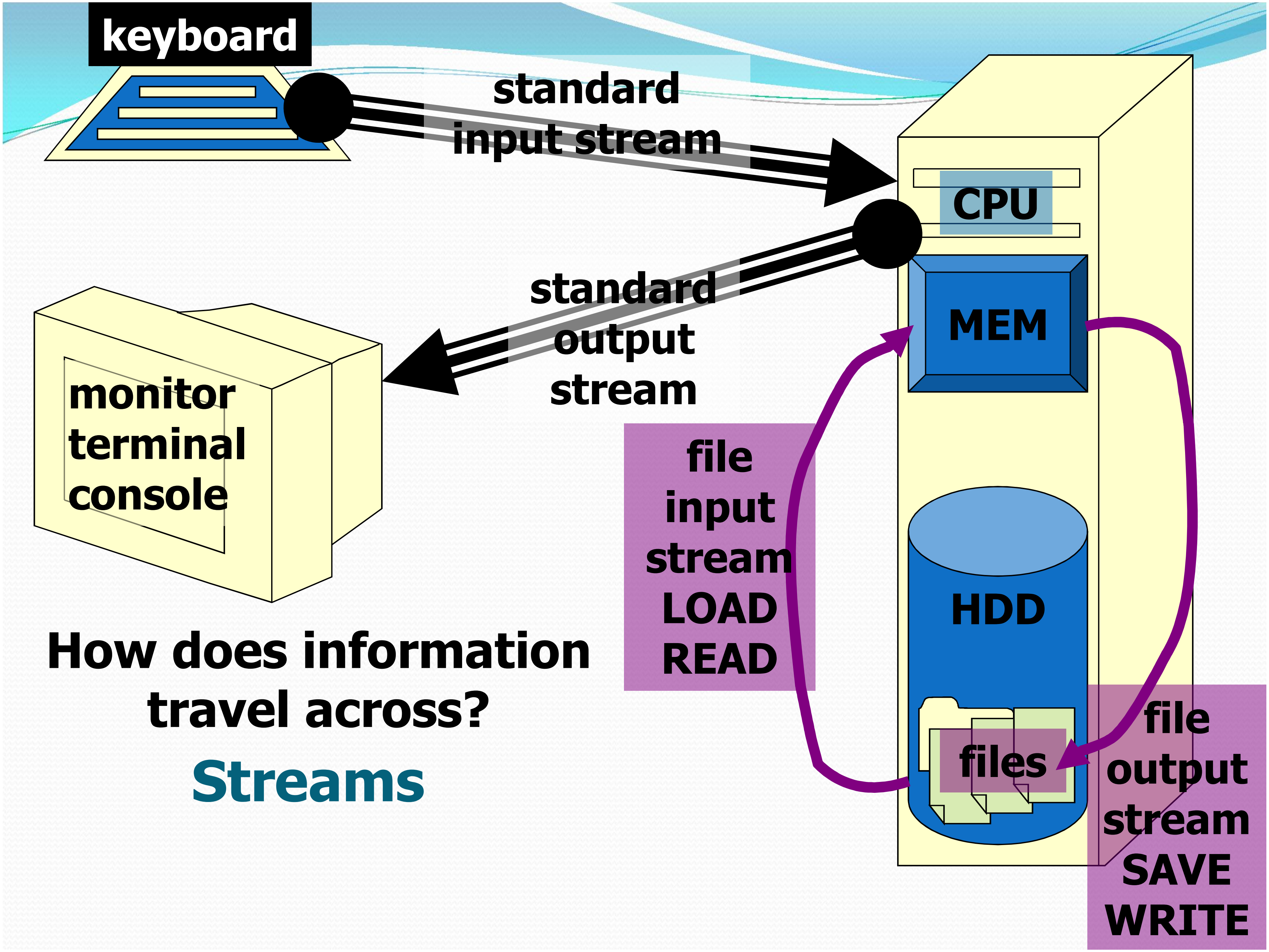
```
    From ThreadA: i= 1
    From ThreadA: i= 2
    From ThreadA: i= 3
    From ThreadA: i= 4
    From ThreadA: i= 5
    From ThreadC: k= 1
    From ThreadC: k= 2
    From ThreadC: k= 3
    From ThreadC: k= 4
    From ThreadC: k= 5
Exit from C
    From ThreadB: j= 1
    From ThreadB: j= 2
    From ThreadB: j= 3
    From ThreadB: j= 4
    From ThreadB: j= 5
Exit from B
Exit from A
```

Files and Streams



How does information travel across?

Streams


keyboard
standard input stream
CPU
standard output stream
MEM
monitor terminal console
file input stream LOAD READ
HDD
files
file output stream SAVE WRITE
How does information travel across?
Streams

Reading and Writing Text Files

- Text files – files containing simple text
  - Created with editors such as notepad, html, etc.

- Simplest way to learn is to extend use of **`Scanner`**
  - Associate with files instead of **`System.in`**

- All input classes, except Scanner, are in java.io
  - **`import java.io.*;`**

Numerical Input

- 2 ways (we've learned one, seen the other)
  - Use **`int`** as example, similar for **`double`**

- First way:
  - Use **`nextInt()`**

```
int number = scanner.nextInt();
```

- Second way:
  - Use **`nextLine(), Integer.parseInt()`**

```
String input = scanner.nextLine();
int number = Integer.parseInt(input);
```

Review: Scanner

- The constructor takes an object of type **`java.io.InputStream`** – stores information about the connection between an input device and the computer or program
  - Example: **`System.in`**
- Recall – only associate ***one*** instance of **`Scanner`** with **`System.in`** in your program
  - Otherwise, get bugs

Reading Files

- The same applies for both console input and file input

- We can use a different version of a Scanner that takes a ***File*** instead of **`System.in`**

- Everything works the same!

Reading Files

- To read from a disk file, construct a **`FileReader`**
- Then, use the **`FileReader`** to construct a **`Scanner`** object

```
FileReader rdr = newFileReader("input.txt");
Scanner fin = new Scanner(rdr);
```

File Class

- **`java.io.File`**
  - associated with an actual file on hard drive
  - used to check file's status
- Constructors
  - `File(<full path>)`
  - `File(<path>, <filename>)`
- Methods
  - `exists()`
  - `canRead()`, `canWrite()`
  - `isFile()`, `isDirectory()`

File Class

- **`java.io.FileReader`**
  - Associated with `File` object
  - Translates data bytes from File object into a stream of characters (much like InputStream vs. InputStreamReader)
- Constructors
  - `FileReader( <File object> );`
- Methods
  - `read(), readLine()`
  - `close()`

Writing to a File

- We will use a **`PrintWriter`** object to write to a file
  - What if file already exists? → Empty file
  - Doesn't exist? → Create empty file with that name
- How do we use a **`PrintWriter`** object?
  - Have we already seen one?

Writing to a File

- The out field of the System class is a **`PrintWriter`** object associated with the console
  - We will associate our **`PrintWriter`** with a file now

```
PrintWriter fout = new PrintWriter("output.txt");
fout.println(29.95);
fout.println(new Rectangle(5, 10, 15, 25));
fout.println("Hello, World!");
```

- This will print the exact same information as with **`System.out`** (except to a file "output.txt")!

Closing a File

- Only main difference is that we have to close the file stream when we are done writing
- If we do not, not all output will written
- At the end of output, call **`close()`**

```
fout.close();
```

File Locations

- When determining a file name, the default is to place in the same directory as your .class files
- If we want to define other place, use an absolute path (e.g. c:\My Documents)

```
in  = new
  FileReader("c:\\homework\\input.dat");
```

```
import java.io.*;
public class dupl {

   public static void main(String args[]) throws IOException {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

Files

File Processing

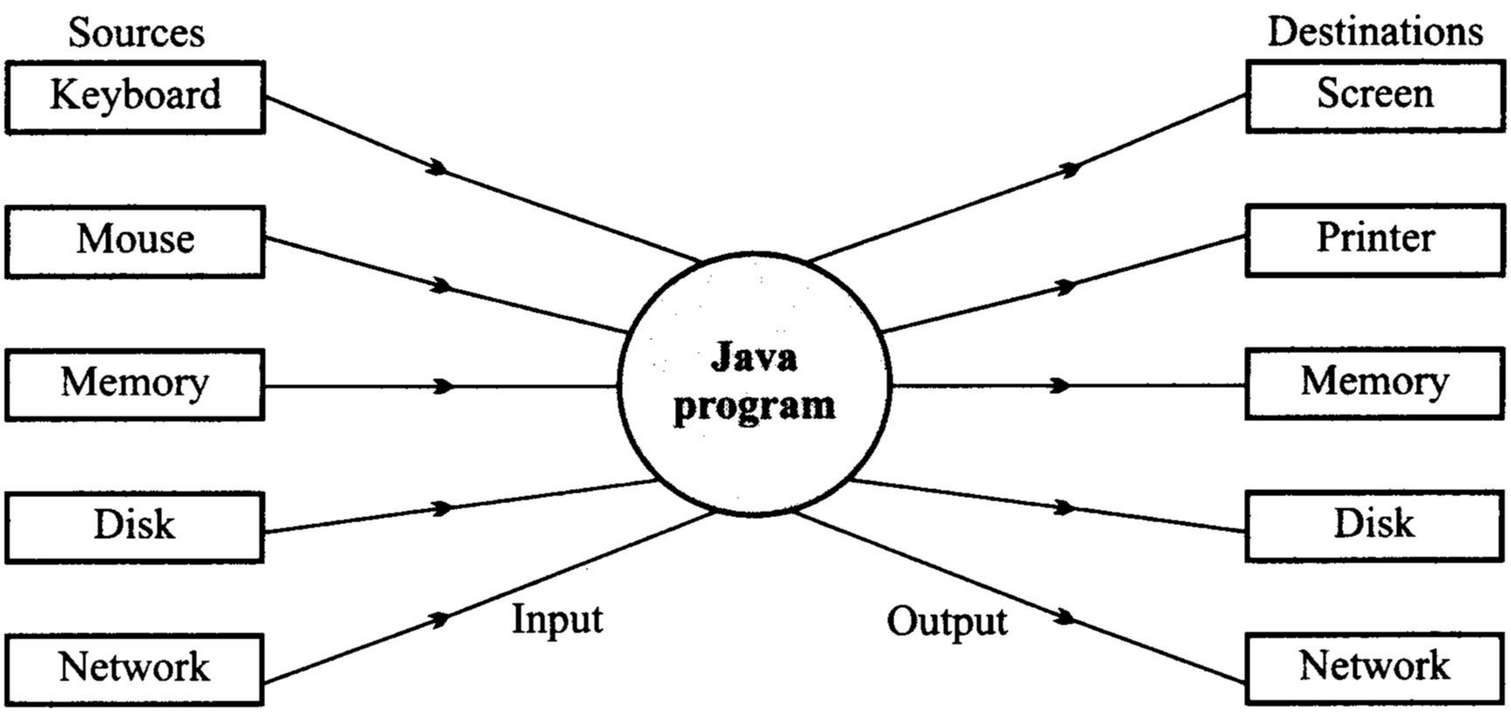
- Storing and manipulating data using files is known as file processing.
- Reading/Writing of data in a file can be performed at the level of bytes, characters, or fields depending on application requirements.
- Java also provides capabilities to read and write class objects directly. The process of reading and writing objects is called object serialisation.

C Input/Output Revision

```
FILE* fp;

fp = fopen("In.file", "rw");
fscanf(fp, ......);
frpintf(fp, .....);
fread(........., fp);
fwrite(.........., fp);
```

I/O and Data Movement

- The flow of data into a program (input) may come from different devices such as keyboard, mouse, memory, disk, network, or another program.
- The flow of data out of a program (output) may go to the screen, printer, memory, disk, network, another program.
- Both input and output share a certain common property such as unidirectional movement of data – a sequence of bytes and characters and support to the sequential access to the data.



Relationship of Java program with I/O devices

Serialization

What is Serialization?

- Ability to read or write an object to a stream
  - Process of "flattening" an object
- Used to save object to some permanent storage
  - Its state should be written in a serialized form to a file such that the object can be reconstructed at a later time from that file
- Used to pass on to another object via the *OutputStream* class
  - Can be sent over the network

Streams Used for Serialization

- ObjectOutputStream
  - For serializing (flattening an object)
- ObjectInputStream
  - For deserializing (reconstructing an object)

Requirement for Serialization

- To allow an object to be serializable:
  - Its class should implement the *Serializable* interface
    - *Serializable* interface is marker interface
  - Its class should also provide a default constructor (a constructor with no arguments)
- Serializability is inherited
  - Don't have to implement *Serializable* on every class
  - Can just implement *Serializable* once along the class hierarchy

Non-Serializable Objects

- Most Java classes are serializable
- Objects of some system-level classes are not serializable
  - Because the data they represent constantly changes
    - Reconstructed object will contain different value anyway
    - For example, thread running in my JVM would be using my system's memory. Persisting it and trying to run it in your JVM would make no sense at all.
- A *NotSerializableException* is thrown if you try to serialize non-serializable objects

What is preserved when an object is serialized?

- Enough information that is needed to reconstruct the object instance at a later time
  - Only the object's data are preserved
  - Methods and constructors are not part of the serialized stream
  - Class information is included

When to use *transient* keyword?

- How do you serialize an object of a class that contains a non-serializable class as a field?
  - Like a Thread object
- What about a field that you don't want to to serialize?
  - Some fields that you want to recreate anyway
  - Performance reason
- Mark them with the *transient* keyword
  - The *transient* keyword prevents the data from being serialized
  - Serialization does not care about access modifiers such as *private* -- all nontransient fields are considered part of an object's persistent state and are eligible for persistence

Example: transient keyword

```
class MyClass implements Serializable {

    // Skip serialization of the transient field
    transient Thread thread;
    transient String fieldIdontwantSerialization;

    // Serialize the rest of the fields
    int data;
    String x;

    // More code
}
```

Serialization: Writing an Object Stream

- Use its *writeObject* method of the *ObjectOutputStream* class

```
public final void writeObject(Object obj)
                                          throws IOException
```

where,

- *obj* is the object to be written to the stream

Deserialization: Reading an Object Stream

- Use its *readObject* method of the *ObjectInputStream* class

```
public final Object readObject()
       throws IOException, ClassNotFoundException
```

where,

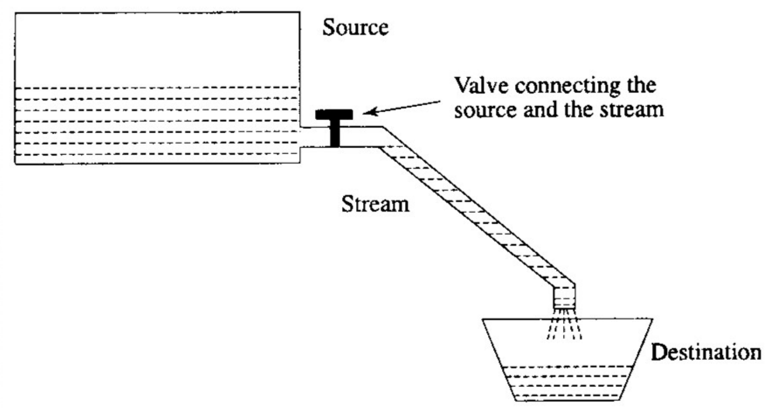
- *obj* is the object to be read from the stream

- The *Object* type returned should be typecasted to the appropriate class name before methods on that class can be executed

The Stream Classes

Streams

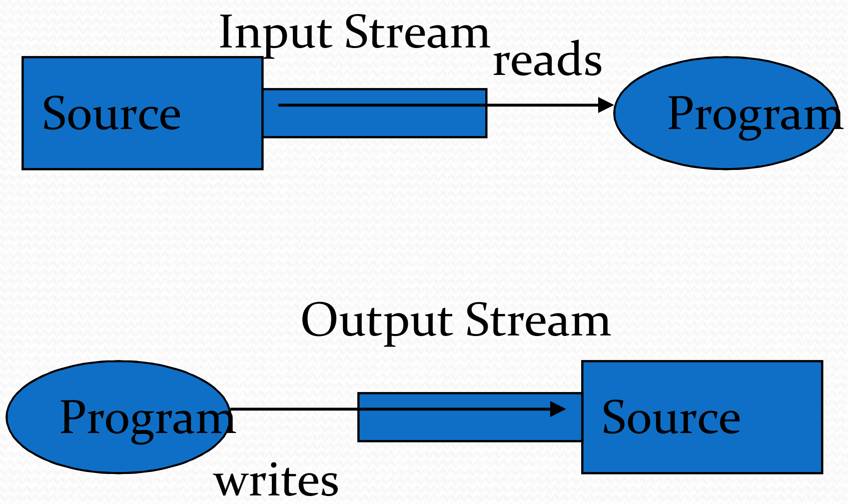
- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



Conceptual view of a stream

Stream Types

- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



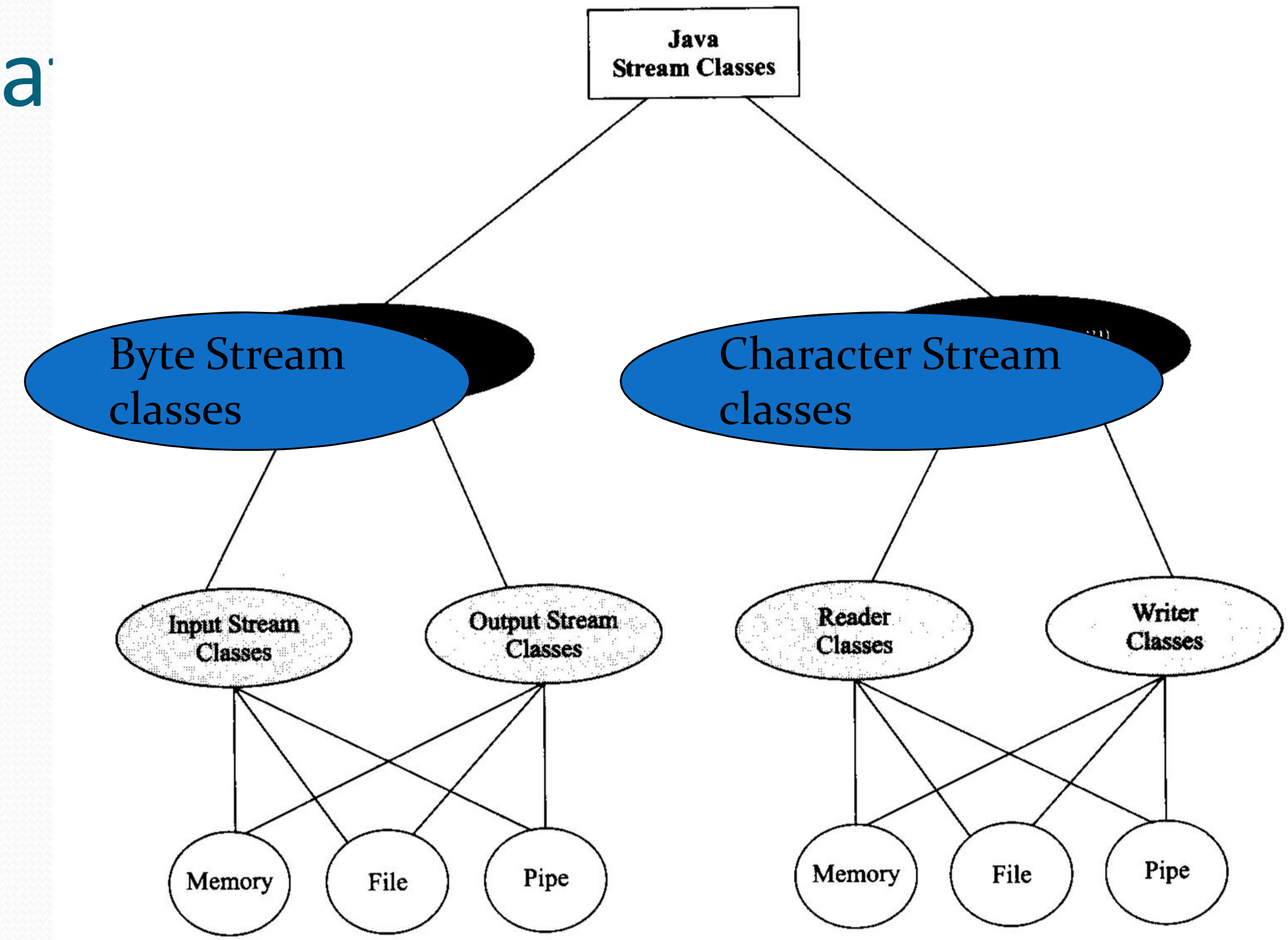
Java Stream Classes

- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

Streams

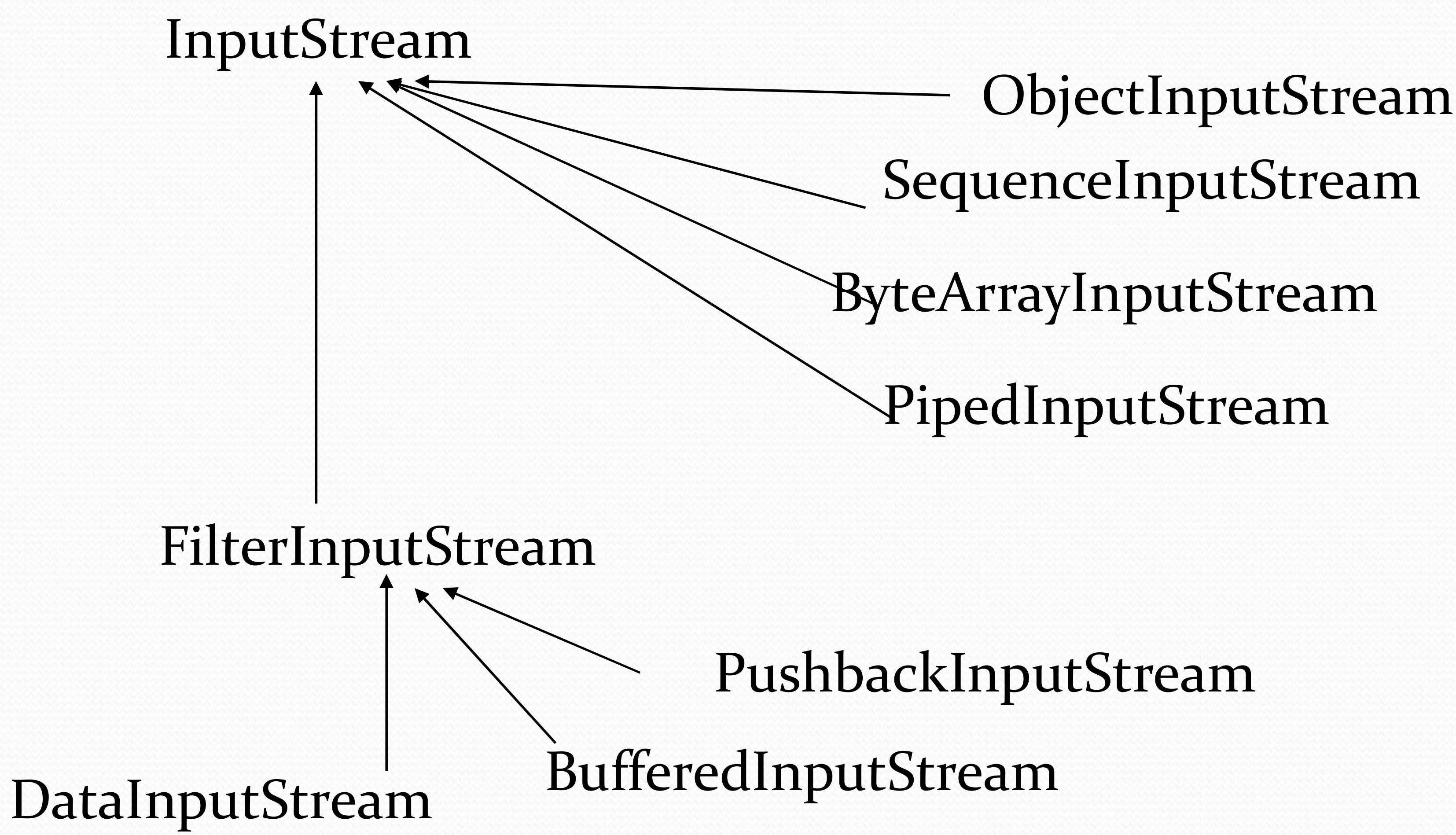
Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

Classifica



Classification of Java stream classes

Byte Input Streams



Byte Input Streams - operations

public abstract int read()	Reads a byte and returns as a integer 0-255
public int read(byte[] buf, int offset, int count)	Reads and stores the bytes in buffer starting at offset. Count is the maximum read.
public int read(byte[] buf)	Same as previous offset=0 and length=buf.length()
public long skip(long count)	Skips count bytes.
public int available()	Returns the number of bytes that can be read.
public void close()	Closes stream

Byte Input Stream - example

- Count total number of bytes in the file

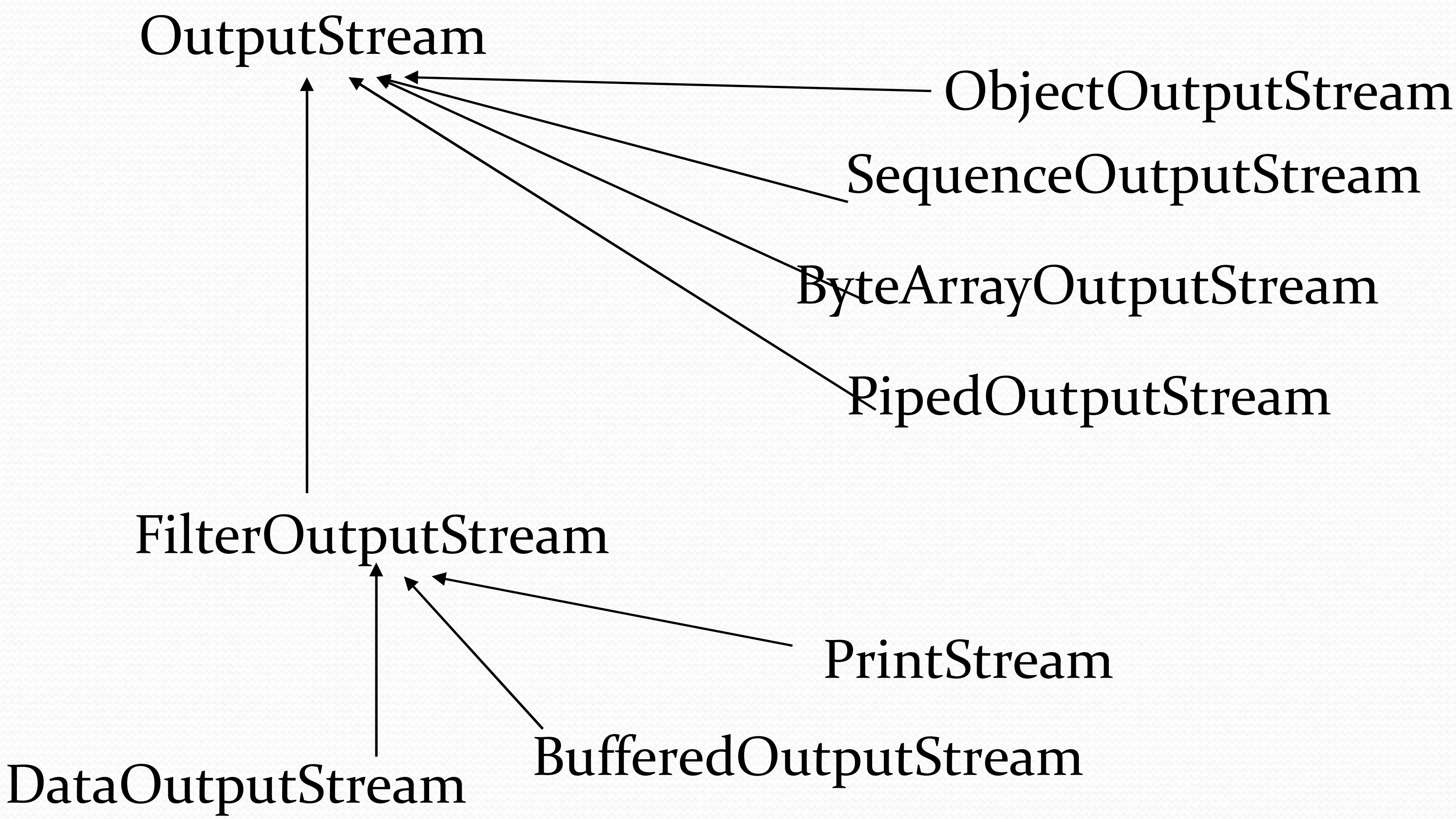
```
import java.io.*;

class CountBytes {
		public static void main(String[] args)
		    throws FileNotFoundException, IOException
		{
				FileInputStream in;
				in  =  new FileInputStream(“InFile.txt”);

				int total = 0;
				while (in.read() != -1)
						total++;
				System.out.println(total + “ bytes”);
		}
}
```

What happens if the file did not exist

- JVM throws exception and terminates the program since there is no exception handler defined.

```
Exception in thread "main" java.io.FileNotFoundException: FileIn.txt (No such file or directory)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:64)
    at CountBytes.main(CountBytes.java:12)
```

Byte Output Streams



Byte Output Streams - operations

public abstract void write(int b)	Write *b* as bytes.
public void write(byte[] buf, int offset, int count)	Write *count* bytes starting from *offset* in *buf.*
public void write(byte[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void flush()	Flushes the stream.
public void close()	Closes stream

Byte Output Stream - example

- Read from standard in and write to standard out

```
import java.io.*;

class ReadWrite {
		public static void main(string[] args)
				throws IOException
		{
				int b;
				while (( b = System.in.read()) != -1)
				{
						System.out.write(b);
				}

}
```

I/O Streams

- A stream is a sequence of bytes that flows from a source to a destination
- In a program, we read information from an input stream and write information to an output stream
- A program can manage multiple streams at a time
- The `java.io` package contains many classes that allow us to define various streams with specific characteristics

I/O Stream Categories

- The classes in the I/O package divide input and output streams into other categories
- An I/O stream is either a
  - *character stream*, which deals with text data
  - *byte stream*, which deals with byte data
- An I/O stream is also either a
  - *data stream*, which acts as either a source or destination
  - *processing stream*, which alters or manages information in the stream

I/O class hierarchy

- **class java.lang.Object**
  - **class java.io.InputStream**
    - **class java.io.ByteArrayInputStream**
    - **class java.io.FileInputStream**
    - **class java.io.FilterInputStream**
  - **class java.io.OutputStream**
    - **class java.io.ByteArrayOutputStream**
    - **class java.io.FileOutputStream**
    - **class java.io.FilterOutputStream**
  - **class java.io.Reader**
    - **class java.io.BufferedReader**
    - **…**
    - **class java.io.InputStreamReader**
  - **class java.io.Writer**
    - **class java.io.BufferedWriter**
    - **…**
    - **class java.io.OutputStreamWriter**

Sources of data streams

- 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`
- We use `System.out` when we execute `println` statements
- `System.in` is declared to be a generic `InputStream` reference, and therefore usually must be mapped to a more useful stream with specific characteristics
- `FileInputStream` and `FileReader` are classes whose constructors open a file for reading

Processing streams

- Processing classes have constructors that take InputSteams as input and produce InputStreams with added functionality
- BufferedReader, and BufferedWriter allow you to write bigger chunks of text to a stream.
  - Buffering is a way of combining multiple reads or writes into a single action. It is a good idea when working with text.
  - Examples: readLine() in BufferedReader and newLine() in BufferedWriter

IOExceptions

- The following exception classes are defined in the java.io package:

  CharConversionException
  EOFException
  FileNotFoundException
  InterruptedIOException
  InvalidClassException
  InvalidObjectException
  NotActiveException
  NotSerializableException
  ObjectStreamException
  OptionalDataException
  StreamCorruptedException
  SyncFailedException
  UnsupportedEncodingException
  UTFDataFormatException
  WriteAbortedException

Stream Benefits

- The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task.

- The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.

- Java programs written to adhere to the abstract, high-level InputStream, OutputStream, Reader, and Writer classes will function properly in the future even when new and improved concrete stream classes are invented.

- This model works very well when we switch from a file system-based set of streams to the network and socket streams.

- Finally, serialization of objects is expected to play an increasingly important role in Java programming in the future.

- Java's serialization I/O classes provide a portable solution to this sometimes tricky task programming.

Fundamentals of Applets – Graphics

Applets

- There are two types of Java programs:
  - Applications and Applets
- We will focus on applets.
  - an applet is a Java program that can be viewed on a Web browser that supports the Java language.
- The easiest way to explain what an applet is and how it works is by example.

Applet Example

- The applet Example:

```
import java.awt.*;
import java.applet.*;
 public class appl extends Applet
{
 public void paint(Graphics g)
{
 g.drawOval(40,40,120,150);
 g.drawOval(57,75,30,20);
 g.drawOval(110,75,30,20);
 g.fillOval(68,81,10,10);
 g.fillOval(121,81,10,10);
 g.drawOval(85,100,30,30);
 g.fillArc(60,125,80,40,180,180);
 g.drawOval(25,92,15,30);
 g.drawOval(160,92,15,30);
 }
}
```

appl Applet

- After compiling the code, the class file is called by an HTML document in a web browser or applet runner (appletviewer) and the output will be displayed on the screen.
- The HTML code (stored in file appl.html) to call an applet is:

```
<applet code = “filename.class”
        width = “width of applet in pixels”
        height = “height of applet in pixels”>
</applet>
```

- applet runner:
  - appletviewer appl.html

Applet Example

- Example (appl.html):

/*<applet code="appl.class"Width=250
height=200></applet>*/

Life Cycle of an Applet

- An Applet executes within an environment provided by a Web browser or a tool such as the applet viewer.
- It does not have a main() method
- There are four methods that are called during the life cycle of an applet:

    init(),
    start(),
    stop(),
    destroy().

Life Cycle of an Applet

- init() method is called only when the applet begins execution. It is common to place code here that needs to be executed only once, such as reading parameters that are defined in the HTML file.
- start() method is executed after the init() method completes execution. In addition, this method is called by the applet viewer or Web browser to resume execution of the applet.
- stop() method is called by the applet viewer or Web browser to suspend execution of an applet.
  - the start() and stop() methods may be called multiple times during the life cycle of the applet.

import Statements

- The first two lines of the program are:

  import java.applet.*;

  import.java.awt.*;

- These two lines “import” or let the Java compiler know that we want to use classes that are in the packages java. applet and java. awt.
  - The java.applet package:
    contains definitions for the applet class
  - The java.awt package:
    contains classes for displaying graphics

import Statements

- The “*” acts as a wildcard that will import all of the classes in the package
- Difference between this “*” and the one used at a command prompt.
  - You can not use it to indicate partial names such as L* to import all the classes that start with L.
- The “*” will import all the public classes in a package but **does not** import the subpackages.

import Statements

- To import all classes in a package hierarchy, you must import each level (or subpackage) explicitly.

  import java. awt.*; does not import the “peer” subpackage.
  To import the “peer” subpackage you must do it explicitly.
  Example:

  ```
  import java.awt.event.*;
  import.java.awt.image.*;
  ```

import Statement Syntax

- The form of an import statement is as follows:
  - import *packageName* .*;
    or
    import *packageName. className* ;
    Examples: import java.applet.Applet;
    import java.awt.Graphics;
- import statements must appear before any of the names defined in the import are used.
- It is a strong recommendation that all imports appear at the beginning of your program.

drawString() method

- The drawString() method belongs to the Graphics class
- g is a Graphics object and we want it to execute it's own drawString() method.
- We also pass it what we want to draw on the screen and where we want the graph to be drawn.
- The drawString() method is defined in the Graphics as follows:

```
Public void drawString( String s, int x, int y)
   {
     Code to draw s on the screen at location x, y
   }
```

Graphics

Graphics

- The java.awt package contains all the necessary classes you need to create graphical user interfaces (GUIs).
- Most of the graphics operations in Java are methods defined in the Graphics class.
- You don’t have to create an instance of the Graphics class because in the applet’s paint() method, a Graphics object is provided for you. By drawing in that object, you draw onto your applet which appears on the screen.
- The Graphics class is part of the java. awt package, so make sure you import it into your Java code.
  - import java. awt. Graphics;

Lines

- To draw a line onto the screen, use the drawLine() method:
  - void drawLine(int x1, int y1, int x2, int y2);
  - This draws a line from the point with coordinates (x1, y1) to the point with coordinates (x2, y2).
  - Example:

```
import java. awt. Graphics;
public class MyLine extends java. applet. Applet {
    public void paint( Graphics g) {
            g. drawLine( 25,25, 75,75);
    }
}
```

  - There is no way to change the line thickness in Java.
    So how do we make thicker lines?

Rectangles

- To draw a rectangle on the screen, use the drawRect() method:
  - void drawRect(int x, int y, int width, int height)
  - This draws an outline of a rectangle with the top left corner of the rectangle having the point (x, y). The size of the rectangle is governed by the width and height arguments.
- To fill in the rectangle we would use the method fillRect(). This works in the same way as drawRect() but fills in the rectangle with the current drawing color.
- To change the current drawing color we use the method:
  - void setColor(Color c)
  - The drawing color stays fixed until it is changed by another call to the setColor() method.

The Color Class

- This class contains 13 constant values that can be used:
  - black, blue, cyan, darkGray, Gray, green, lightGray, magenta, orange, pink, red, white, yellow
- To address them we have to reference them through the Color class
  - eg. **Color. black**
  - Too set the current color to blue:

    g. setColor(Color. blue)
- Colors in Java are described by the RGB (Red, Green, Blue) model.

  - This model specifies the amount of red, green, and blue in a color.
  - The intensity of each component is measured as an integer between

    0 and 255, with 0 representing no light.

    (0,0,0) is black

    (128,128,128) is medium gray

The Color Class

- To declare a new color in Java, use the "new" operator
  - Color myColor = new Color(255, 0, 128);
  - We now have a new color and since we know it is an object of the Color class we can use it directly

    g. setColor(myColor);
  - You can also define the color "on the fly" or in line with the **setColor()** method

    g. setColor(new Color(255,0,128));

The Font Class

- There are five basic fonts in Java
  - SanSerif (Helvetica), Serif (Times Roman), Monospaced (Courier), Dialog, DialogInput
- There are some constant values associated with the Font class as well.
  - Font.BOLD, Font.PLAIN, Font.ITALIC
- Create a Font object by using the “new” operator
  - Font myFont = new Font(“Helvetica”, Font.BOLD, 12);
  - After creating a font, you have to set it before it can be used:
    g.setFont(myFont);
  - You can also do this in line with the setFont() method
    g.setFont(new Font(“Helvetica”, Font.BOLD, 12));
- You can also combine styles by adding them together, for example

  Font myFont = new Font(“Helvetica”, Font.BOLD+ Font.ITALIC, 12)

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

It works at client side so less response time.

Secured

It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Lifecycle of Java Applet

Applet is initialized.
Applet is started.
Applet is painted.
Applet is stopped.
Applet is destroyed.

Applet Lifecycle

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.
public void init(): is used to initialized the Applet. It is invoked only once.
public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.
public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
public void destroy(): is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.
public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

It works at client side so less response time.

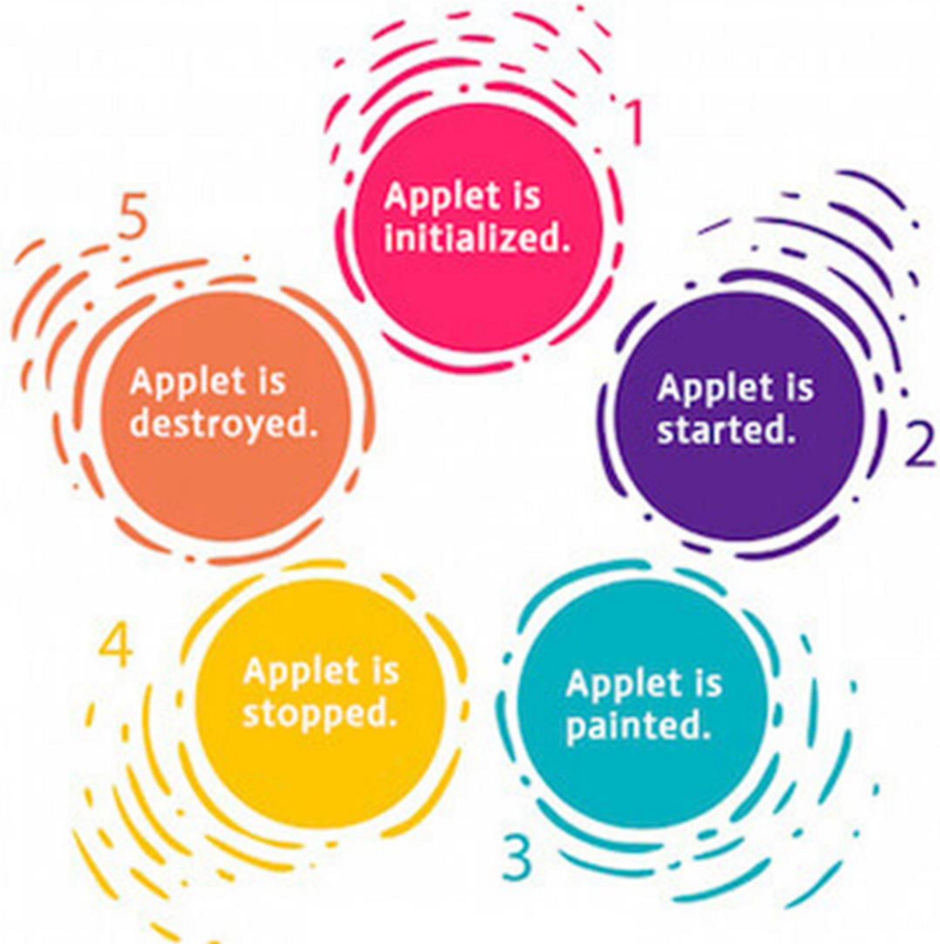
Secured

It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Lifecycle of Java Applet

Applet is initialized.
Applet is started.
Applet is painted.
Applet is stopped.
Applet is destroyed.

Applet Lifecycle



java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

public void init(): is used to initialized the Applet. It is invoked only once.

public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

public void destroy(): is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Uses

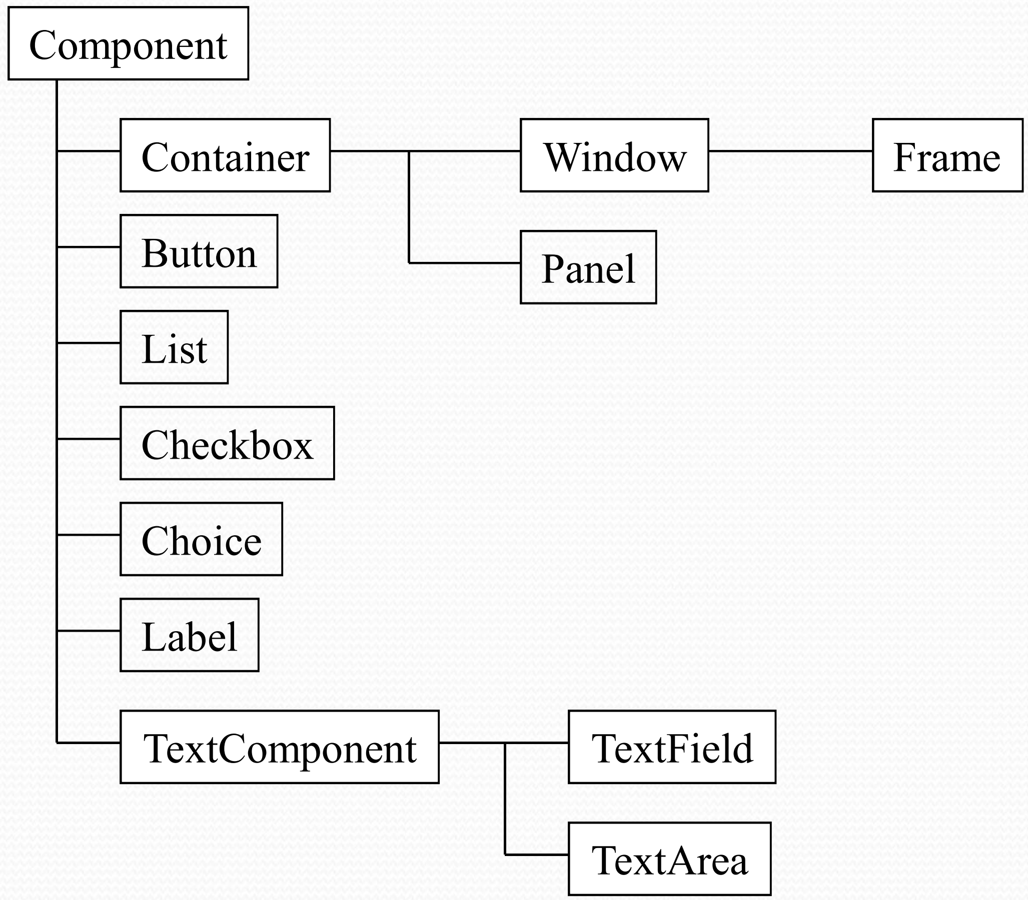
Java applets are **used** to provide interactive features to web **applications** and can be executed by browsers for many platforms. They are small, portable **Java** programs embedded in HTML pages and can run automatically when the pages are viewed.

Abstract Windowing Toolkit - AWT

AWT (Abstract Windowing Toolkit)

- The AWT is roughly broken into three categories
  - Components
  - Layout Managers
  - Graphics

AWT Class Hierarchy



Component

- Component is the superclass of most of the displayable classes defined within the AWT. Note: it is abstract.
- MenuComponent is another class which is similar to Component except it is the superclass for all GUI items which can be displayed within a drop-down menu.
- The Component class defines data and methods which are relevant to all Components

setBounds
setSize
setLocation
setFont
setEnabled
setVisible
setForeground -- colour
setBackground -- colour

Container

- Container is a subclass of Component. (ie. All containers are themselves, Components)
- Containers contain components
- For a component to be placed on the screen, it must be placed within a Container
- The Container class defined all the data and methods necessary for managing groups of Components
  - add
  - getComponent
  - getMaximumSize
  - getMinimumSize
  - getPreferredSize
  - remove
  - removeAll

Windows and Frames

- The Window class defines a top-level Window with no Borders or Menu bar.
  - Usually used for application splash screens
- Frame defines a top-level Window with Borders and a Menu Bar
  - Frames are more commonly used than Windows
- Once defined, a Frame is a Container which can contain Components

```
Frame aFrame = new Frame(Hello World);
aFrame.setSize(100,100);
aFrame.setLocation(10,10);
aFrame.setVisible(true);
```

Panels

- When writing a GUI application, the GUI portion can become quite complex.
- To manage the complexity, GUIs are broken down into groups of components. Each group generally provides a unit of functionality.
- A Panel is a rectangular Container whose sole purpose is to hold and manage components within a GUI.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Cancel"));

Frame aFrame = new Frame("Button Test");
aFrame.setSize(100,100);
aFrame.setLocation(10,10);

aFrame.add(aPanel);
```

Buttons

- This class represents a push-button which displays some specified text.
- When a button is pressed, it notifies its Listeners. (More about Listeners in the next chapter).
- To be a Listener for a button, an object must implement the ActionListener Interface.

```
Panel aPanel = new Panel();
Button okButton = new Button("Ok");
Button cancelButton = new Button("Cancel");

aPanel.add(okButton));
aPanel.add(cancelButton));

okButton.addActionListener(controller2);
cancelButton.addActionListener(controller1);
```

Labels

- This class is a Component which displays a single line of text.
- Labels are read-only. That is, the user cannot click on a label to edit the text it displays.
- Text can be aligned within the label

```
Label aLabel = new Label("Enter password:");
aLabel.setAlignment(Label.RIGHT);

aPanel.add(aLabel);
```

List

- This class is a Component which displays a list of Strings.
- The list is scrollable, if necessary.
- Sometimes called Listbox in other languages.
- Lists can be set up to allow single or multiple selections.
- The list will return an array indicating which Strings are selected

```
List aList = new List();
       aList.add("Calgary");
       aList.add("Edmonton");
       aList.add("Regina");
       aList.add("Vancouver");

aList.setMultipleMode(true);
```

Checkbox

- This class represents a GUI checkbox with a textual label.
- The Checkbox maintains a boolean state indicating whether it is checked or not.
- If a Checkbox is added to a CheckBoxGroup, it will behave like a radio button.

```
Checkbox creamCheckbox = new CheckBox("Cream");
Checkbox sugarCheckbox = new CheckBox("Sugar");
[]
if (creamCheckbox.getState())
{
        coffee.addCream();
}
```

Choice

- This class represents a dropdown list of Strings.
- Similar to a list in terms of functionality, but displayed differently.
- Only one item from the list can be selected at one time and the currently selected element is displayed.

```
Choice aChoice = new Choice();
aChoice.add("Calgary");
aChoice.add("Edmonton");
aChoice.add("Alert Bay");
[]

String selectedDestination= aChoice.getSelectedItem();
```

TextField

- This class displays a single line of optionally editable text.
- This class inherits several methods from TextComponent.
- This is one of the most commonly used Components in the AWT

```
TextField emailTextField = new TextField();
TextField passwordTextField = new TextField();
passwordTextField.setEchoChar("*");
[…]

String userEmail = emailTextField.getText();
String userpassword = passwordTextField.getText();
```

TextArea

- This class displays multiple lines of optionally editable text.
- This class inherits several methods from TextComponent.
- TextArea also provides the methods: appendText(), insertText() and replaceText()

```
// 5 rows, 80 columns
TextArea fullAddressTextArea = new TextArea(5, 80);
[]

String userFullAddress= fullAddressTextArea.getText();
```

Layout Managers

- Since the Component class defines the setSize() and setLocation() methods, all Components can be sized and positioned with those methods.
- Problem: the parameters provided to those methods are defined in terms of pixels. Pixel sizes may be different (depending on the platform) so the use of those methods tends to produce GUIs which will not display properly on all platforms.
- Solution: Layout Managers. Layout managers are assigned to Containers. When a Component is added to a Container, its Layout Manager is consulted in order to determine the size and placement of the Component.
- NOTE: If you use a Layout Manager, you can no longer change the size and location of a Component through the setSize and setLocation methods.

Layout Managers (cont)

- There are several different LayoutManagers, each of which sizes and positions its Components based on an algorithm:
  - FlowLayout
  - BorderLayout
  - GridLayout

- For Windows and Frames, the default LayoutManager is BorderLayout. For Panels, the default LayoutManager is FlowLayout.

Flow Layout

- The algorithm used by the FlowLayout is to lay out Components like words on a page: Left to right, top to bottom.
- It fits as many Components into a given row before moving to the next row.

```
Panel aPanel = new Panel();
aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

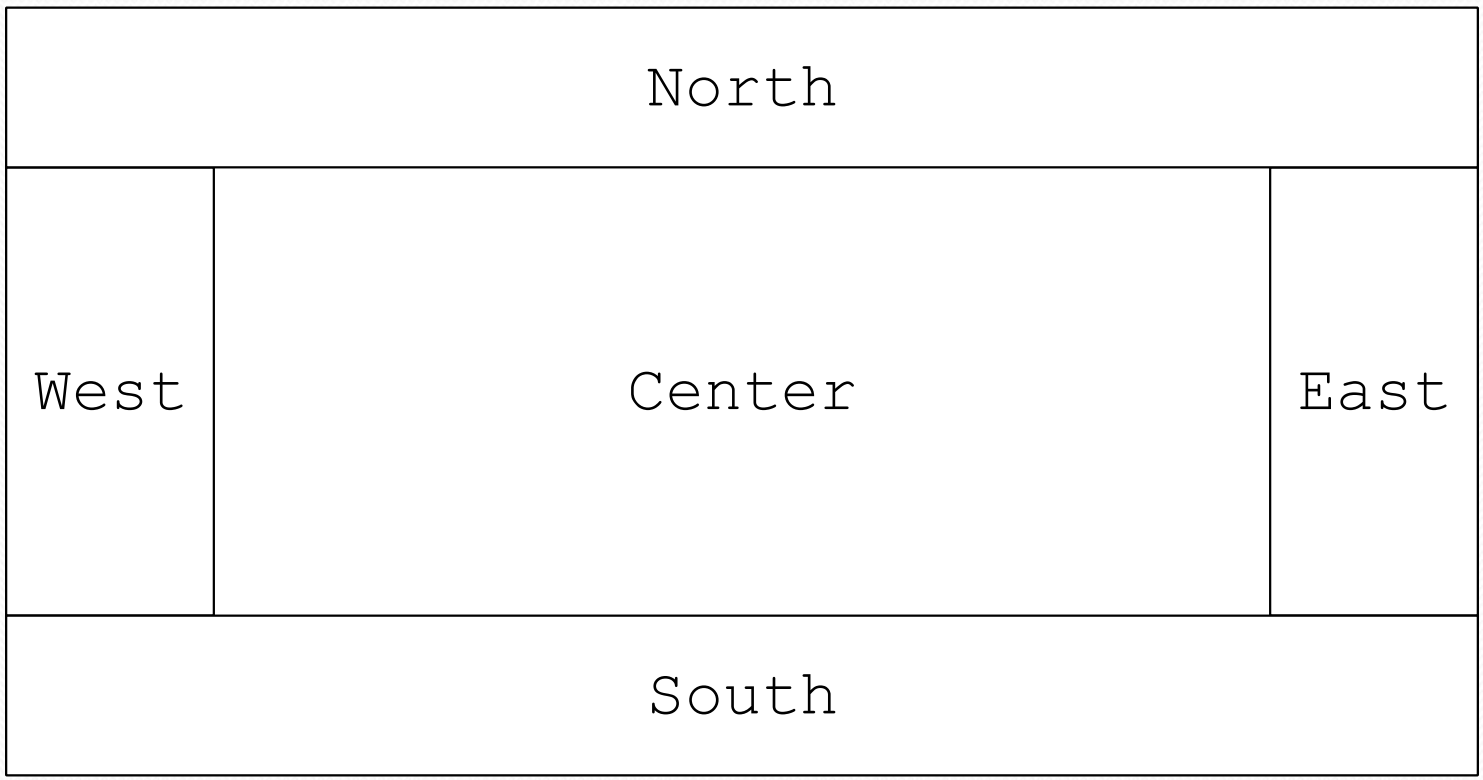
Border Layout

- The BorderLayout Manager breaks the Container up into 5 regions (North, South, East, West, and Center).
- When Components are added, their region is also specified:

```
Frame aFrame = new Frame();
aFrame.add("North", new Button("Ok"));
aFrame.add("South", new Button("Add"));
aFrame.add("East", new Button("Delete"));
aFrame.add("West", new Button("Cancel"));
aFrame.add("Center", new Button("Recalculate"));
```

Border Layout (cont)

- The regions of the BorderLayout are defined as follows:



Grid Layout

- The GridLayout class divides the region into a grid of equally sized rows and columns.
- Components are added left-to-right, top-to-bottom.
- The number of rows and columns is specified in the constructor for the LayoutManager.

```
Panel aPanel = new Panel();
GridLayout theLayout = new GridLayout(2,2);
aPanel.setLayout(theLayout);

aPanel.add(new Button("Ok"));
aPanel.add(new Button("Add"));
aPanel.add(new Button("Delete"));
aPanel.add(new Button("Cancel"));
```

Graphics

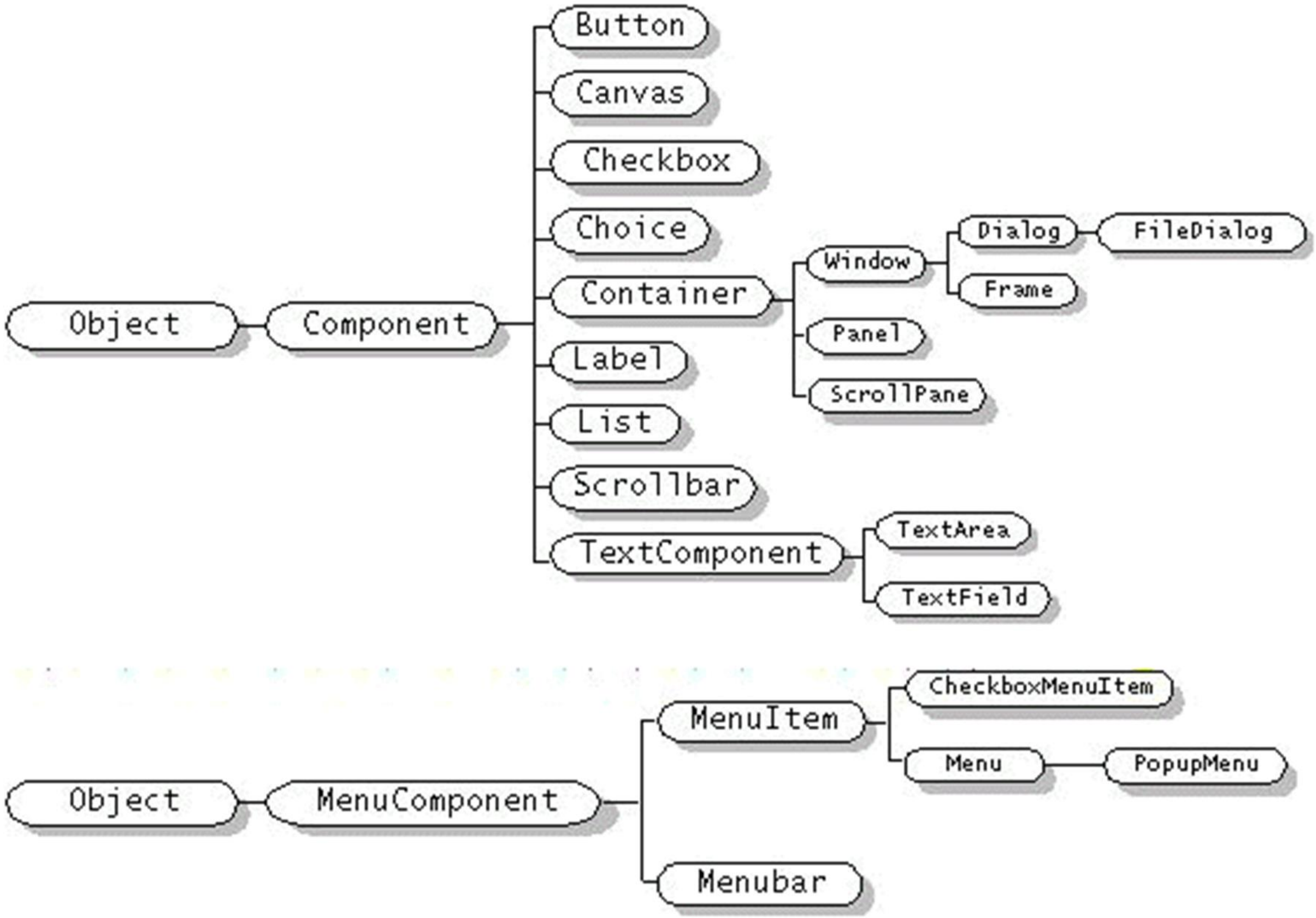
- It is possible to draw lines and various shapes within a Panel under the AWT.
- Each Component contains a Graphics object which defines a Graphics Context which can be obtained by a call to getGraphics().
- Common methods used in Graphics include:

drawLine
drawOval
drawPolygon
drawPolyLine
drawRect
drawRoundRect
drawString
draw3DRect
fill3DRect
fillArc
fillOval
fillPolygon
fillRect
fillRoundRect
setColor
setFont
setPaintMode
drawImage

AWT Components

Using AWT Components

- Component
  - Canvas
  - Scrollbar
  - Button
  - Checkbox
  - Label
  - List
  - Choice
  - TextComponent
    - TextArea
    - TextField
- Component
  - Container
    - Panel
    - Window
      - Dialog
        - FileDialog
      - Frame
- MenuComponent
  - MenuItem
    - Menu



Object
Component
Button
Canvas
Checkbox
Choice
Container
Window
Dialog
FileDialog
Frame
Panel
ScrollPane
Label
List
Scrollbar
TextComponent
TextArea
TextField
Object
MenuComponent
MenuItem
CheckboxMenuItem
Menu
PopupMenu
Menubar

Frame

```
import java.awt.*;

public class TestFrame extends Frame {
    public TestFrame(String title){
            super(title);
    }
    public static void main(String[] args){
            Frame f = new TestFrame("TestFrame");
            f.setSize(400,400);
            f.setLocation(100,100);
            f.show();
    }
}
```

How to Use Buttons?

```
import java.awt.*;
public class button {
public static void main(String[] args) {
    Frame f=new Frame("Button Example");
    Button b=new Button("Click Here");
    b.setBounds(50,100,80,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

How to Use Labels?

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class text extends JFrame {
        static JFrame f;
        static JLabel l;
        text()
        {
        }
        public static void main(String[] args)
        {
                f = new JFrame("label");
                l = new JLabel();
                l.setText("label text");
                JPanel p = new JPanel();
                p.add(l);
                f.add(p);
                f.setSize(300, 300);
                f.show();
        }
}
```

How to Use Checkboxes?

```
import javax.swing.*;
public class checkbox
{
    checkbox(){
      JFrame f= new JFrame("CheckBox Example");
      JCheckBox checkBox1 = new JCheckBox("C++");
      checkBox1.setBounds(100,100, 50,50);
      JCheckBox checkBox2 = new JCheckBox("Java", true);
      checkBox2.setBounds(100,150, 50,50);
      f.add(checkBox1);
      f.add(checkBox2);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
   {
   new checkbox();
   }
}
```

How to Use Choices?

```
import java.awt.*;
import javax.swing.*;
class choice {
        static Choice c;
        static JFrame f;
        choice()
        {
        }
        public static void main(String args[])
        {
                f = new JFrame("choice");
                JPanel p = new JPanel();
                c = new Choice();
                c.add("Andrew");
                c.add("Arnab");
                c.add("Ankit");
                p.add(c);
                f.add(p);
                f.show();
                f.setSize(300, 300);
        }
}
```

How to Use TextArea

```
import java.awt.*;
public class TextAreaExample
{
    TextAreaExample(){
      Frame f= new Frame();
         TextArea area=new TextArea("Welcome");
      area.setBounds(10,30, 300,300);
      f.add(area);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
  new TextAreaExample();
}
}
```

How to Use TextField

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

How to Use Lists?

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class ListEx1
{
String [] seasons;
Frame jf;
List list;
Label label1;
ListEx1()
{
jf= new Frame("List");
list= new List(7);
label1 = new Label("Select your favorite sports from the list :");
list.add("Badminton");
list.add("Hockey");
list.add("Tennis");
list.add("Football");
list.add("Cricket");
```

```
list.add("Formula One");
list.add("Rugby");
jf.add(label1);
jf.add(list);
jf.setLayout(new FlowLayout());
jf.setSize(260,220);
jf.setVisible(true);
}
public static void main(String... ar)
{
new ListEx1();
}
}
```

How to Use Menus?

```
import javax.swing.*;
class MenuExample
{
        JMenu menu, submenu;
        JMenuItem i1, i2, i3, i4, i5;
        MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1); menu.add(i2); menu.add(i3);
        submenu.add(i4); submenu.add(i5);
```

```
menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

Event Handling

Event Handling

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

Three Steps of Event Handling

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

1. Prepare to accept events

- Import package java.awt.event
- Applet manifests its desire to accept events by promising to "implement" certain methods
- 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:

  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

- The appropriate abstract methods are implemented.
- Example:

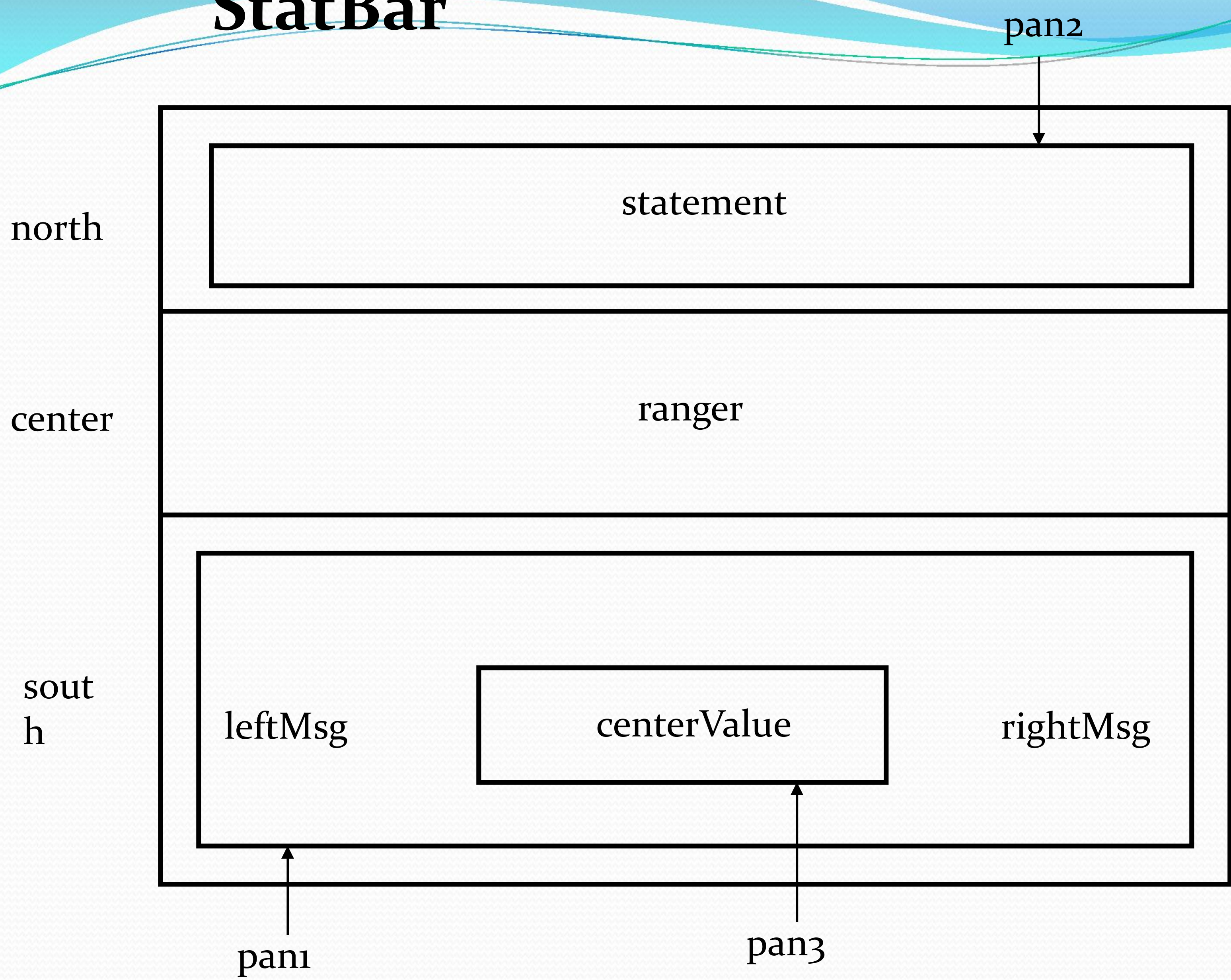
  actionPerformed() is automatically called whenever the user clicks the button.

  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.

StatBar


pan2
north
statement
center
ranger
sout
h
leftMsg
centerValue
rightMsg
pan1
pan3

Event handling

Event handling

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

Events

- An *event* is an object that describes a state change in a source.
- It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.
- Events can be defined as needed and appropriate by application.

Event sources

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

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

  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example,
  1. The method that registers a keyboard event listener is called **addKeyListener()**.
  2. The method that registers a mouse motion listener is called **addMouseMotionListener()**.

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

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

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

- A source must also provide a method that allows a listener to unregister an interest in a specific type of event.
- The general form is:

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

  Here, *Type* is the name of the event and *el* is a reference to the event listener.
- For example, to remove a keyboard listener, you would call **removeKeyListener( )**.
- The methods that add or remove listeners are provided by the source that generates events.
- For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event classes

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

  EventObject(Object *src*)

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

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

Event Classes in java.awt.event

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

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

Event Listeners

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

Delegation event model

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

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

Note

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

Handling mouse events

- mouse events can be handled by implementing the **MouseListener** and the **MouseMotionListener** interfaces.
- **MouseListener Interface** defines five methods. The general forms of these methods are:
    1. void mouseClicked(MouseEvent me)
    2. void mouseEntered(MouseEvent me)
    3. void mouseExited(MouseEvent me)
    4. void mousePressed(MouseEvent me)
    5. void mouseReleased(MouseEvent me)
- **MouseMotionListener Interface.** This interface defines two methods. Their general forms are :
    1. void mouseDragged(MouseEvent me)
    2. void mouseMoved(MouseEvent me)

Handling keyboard events

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

Networking

InetAddress Class

- The Inetaddress class provides you with a limited interface to DNS for doing both forward and reverse internet address lookups
  - An InetAddress class method corresponds to a DNS request

InetAddress Class

- No public constructor
- Three static methods:
  - InetAddress getByName(String)
    - Static method used to retrieve the address for the host name passed as the parameter.
  - InetAddress [ ] getAllByName(String)
    - Static method used to retrieve all the addresses for the host name passed as a parameter.
  - InetAddress getLocalHost( )
    - Static method used to retrieve the address for the current, or local, host.

InetAddress Class

- Three additional “getter” methods
  - String getHostName( )
    - Returns the host name.
  - byte[ ] getAddress( )
    - Returns the IP address.
  - String getHostAddress( )
    - Returns the IP address as a string.

InetAddress Examples

```
try
{    InetAddress fullname = netAddress.getByName(“bigyellowcat.cs.binghamton.edu");
     InetAddress alias = InetAddress.getByName(“bigyellowcat");
     InetAddress octets = InetAddress.getByName(“128.226.121.44");
     if (fullname.equals(alias) && fullname.equals(octets))
          // All is right with the world! }
catch (UnknownHostException e)
       { // Exception handling here. }
```

TCP Sockets

- Once a TCP socket connection is made, a virtual stream is in place. Java’s IO model is that of a stream, therefore the models are consistent; all you need to do connect a TCP socket to a stream and read and write the streams as normal

Socket Class - TCP Client sockets

- Socket(String ip, int port)
  - Creates a streaming socket and binds it to the host and port specified as parameters.
- Socket(String ip, int port, boolean TCPorUDP)
  - Creates a socket and binds it to the host and port specified as parameters. The last parameter is used to indicate whether the socket should be a stream or datagram socket.
- Socket(InetAddress ia, int port)
  - Creates a streaming socket connected to the specified host and port.
- Socket(InetAddress ia, int port, boolean TCPorUDP)
  - Creates a socket connected to the specified host and port. The last parameter specifies whether the socket should be a stream or datagram socket.

Client Sockets

- InetAddress getInetAddress( )
  - Returns an InetAddress object representing the host for this socket.
- Int getPort( )
  - . Returns the port number on the remote host for this socket
- Int getLocalPort( )
  - Returns the port number on the local host for this socket.
- InputStream getInputStream( )
  - Returns an input stream for the socket.
- OutputStream getOutputStream( )
  - Returns an output stream for the socket.
- Close( )
  - Closes the socket.
- SetSocketImplFactory (SocketImplFactory)
  - Sets the socket factory that will be used to create all sockets.

Reading and Writing

```
try
 {
   Socket socket = new Socket("somehost.somewhere.com", -1);
   // Always a good idea to buffer the stream to mitigate blocking.
    PrintStream out =        new PrintStream( new
           BufferedOutputStream(socket.getOutputStream()));
   out.println("Are you listening?");
   DataInputStream in = new DataInputStream( new
           BufferedInputStream(socket.getInputStream()));
   in.readLine();
    // ...
    // Don't forget to close the socket!
    socket.close()
}
catch (Exception e)
   // Exception handling logic.
```

UDP Sockets

- Since UDP is a connectionless protocol; there is no virtual stream between the hosts so streams are not used for IO.
- UDP applications are not thought of in terms of clients and servers, but rather in terms of senders and receivers.
  - For conversational applications both ends (sender and receiver) will be changing states from sender to receiver and back again
  - Many UDP based applications are simple send a request then receive the data (sender's perspective), like a DNS request. The receiver's perspective is to 'listen' for a request, send the response, listen for more requests.

DatagramPacket Class

- UDP sockets send and receive Datagrams
- Constructors: two for receiving, four for sending
  - DatagramPacket(byte[ ] buff , int len)
    - Constructs a DatagramPacket for receiving packets of length len.
  - DatagramPacket(byte[] buf, int off, int len)
    - Constructs a DatagramPacket for receiving packets of length len, specifying an offset of off bytes into the buffer.
  - DatagramPacket((byte[] buf, int len, InetAddress addr, int port)
    - Constructs a datagram packet for sending packets of length len to the specified port number on the specified host.
  - DatagramPacker(byte[] buf, int off, int len, InetAddress addr, int port)
    - Constructs a datagram packet for sending packets of length len with offset off to the specified port number on the specified host.
  - DatagramPacket(byte[] buf, int off, int len, SocketAddress addr)
    - Constructs a datagram packet for sending packets of length len with offset off to the specified port number on the specified host.

DatagramPacket Class

- DatagramPacket(byte[] buf, int len, SocketAddress addr)
  - Constructs a datagram packet for sending packets of length len to the specified port number on the specified host.

Getter methods

getAddress( )
getData( )
getLength( )
getOffset( )
getPort( )
getSocketAddress( )

Setter methods

setAddress(InetAddress iaddr)
setData(byte[ ] buf)
setData(byte[] buf, int offset, int length)
setLength(int len)
setPort(int port)
setSocketAddress(SocketAddress saddr)

DatagramSocket Class – UDP Sockets

- Constructors
  - DatagramSocket()
    - Constructs a datagram socket and binds it to any available port on the local host.
  - DatagramSocket(DatagramSocketImpl impl)
    - Creates an unbound datagram socket with the specified DatagramSocketImpl.
  - DatagramSocket(int port)
    - Constructs a datagram socket and binds it to the specified port on the local host.
  - DatagramSocket(int port, InetAddress iaddr)
    - Creates a datagram socket, bound to the specified local address.
  - DatagramSocket(SocketAddress bindaddr)
    - Creates a datagram socket, bound to the specified local socket address.

DatagramSocket Class – operational Methods

- Operational (void) Methods
  - bind(SocketAddress addr)
  - connect(InetAddress address, int port)
  - connect(SocketAddress addr)
  - disconnect( )
  - **receive**(DatagramPacket p)
  - **send**(DatagramPacket p)
  - close( )

DatagramSocket Class – getter methods

Getter Methods:

DatagramChannel getChannel( )

InetAddress getInetAddress( )

boolean getBroadcast( )

InetAddress getLocalAddress( )

int getLocalPort( )

SocketAddress getLocalSocketAddress( )

SocketAddress getRemoteSocketAddress( )

int getPort( )

int getReceiveBufferSize( )

int getSendBufferSize( )

boolean getReuseAddress( )

int getSoTimeout( )

int getTrafficClass( )

DatagramSocket Class – setter methods

Setter Methods:

void setBroadcast(boolean on)

static void setDatagramSocketImplFactory (DatagramSocketImplFactory fac)

void serReceiveBufferSize(int size)

void setReuseAddress(boolean on)

void setSevdBufferSize(int size)

void setSoTimeout(int timeout)

void setTrafficClass(int tc)

DatagramSocket Class – test methods

Test Methods:

boolean isBound( )

boolean isClosed( )

boolean isConnected( )

URL Class

- RFC 2396
- essentially a “pointer” to a resource on the World Wide Web
  - different services use slightly different formats
    - file://ftp.yoyodyne.com/pub/files/foobar.txt
    - http://www.yahoo.com/index.html
    - ftp://useracct@someftpserver.com
    - news:rec.gardening
    - gopher://gopher.banzai.edu:1234/

URL Class - Constructors

- **URL(String spec)**
  - Creates a URL object from the String representation.
- URL(String protocol, String host, int port, String file)
  - Creates a URL object from the specified protocol, host, port number, and file.
- URL(String protocol, String host, int port, String file, URLStreamHandler handler)
  - Creates a URL object from the specified protocol, host, port number, file, and handler.
- URL(String protocol, String host, String file)
  - Creates a URL from the specified protocol name, host name, and file name.
- URL(URL context, String spec)
  - Creates a URL by parsing the given spec within a specified context
- URL(URL context, String spec, URLStreamHandler handler)
  - Creates a URL by parsing the given spec with the specified handler within a specified context.

URL Class - Methods

<u>Getters</u>

String getAuthority( )

Object getContent( )

Object getContent(Classes[ ] classes)

int getDefaultPort( )

String getFile( )

String getHost( )

String getPath( )

int getPort( )

String getProtocol( )

String getQuery( )

String getRef( )

String getUserInfo( )

URL Class - Methods

Setters

set(String protocol, String host, int port, String file, String ref)

set(String protocol, String host, int port, String authority, String userInfo, String path, String query,

String ref)

setURLStreamHandlerFactory(URLStreamHandlerFactory fac)

URL Class – Utility methods

int hashCode( )

URLConnection openConnection( )

InputStream openStream( )

boolean sameFile(URL other)

String toExternalForm( )

String toString( )

boolean equals(Object obj)

ServerSocket

- Used as the main connection point for some service you wish to provide.
- Once created, it listens for connection requests then queues the request for disposition
- On Unix/Linux you must be root to use

ServerSocket – Life cycle

- A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
- The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
- Depending on the type of server, either the Socket's getInputStream( ) method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
- The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
- The server, the client, or both close the connection.
- The server returns to step 2 and waits for the next connection.

ServerSockets - threads

- Simultaneous requests are held in a queue, as each request is removed from the queue and processed new connections requests can be added to the queue. Connection requests received while the queue is full will be blocked.
  - Some clients use multiple retries in this case as queue space will usually open up pretty quickly.
- For simple protocols (DayTime) the queue can usually handle all of the requests without problem
- For more complex protocols (HTTP) use a thread to process each connection. Threads have less overhead than spawning an entire child process.

ServerSocket - Constructors

- public ServerSocket(int port) throws IOException, BindException
- public ServerSocket(int port, int queueLength) throws IOException, BindException
- public ServerSocket(int port, int queueLength, InetAddress bindAddress) throws IOException

ServerSocket – methods

- accept( ) – accepts a connection request and creates a socket to the remote user
- close( ) – close the server socket and wait for next connection request

ServerSocket - example

```
ServerSocket server = new ServerSocket(5776);
 while (true)
     { Socket connection = server.accept( );
       OutputStreamWriter out =
              new
OutputStreamWriter(connection.getOutputStream( ));
       out.write("You've connected to this server. Bye-bye
now.\r\n");
       connection.close( );
     }
```

Server Sockets - problems

- You are not root
- Port is already in use

Proxy Servers

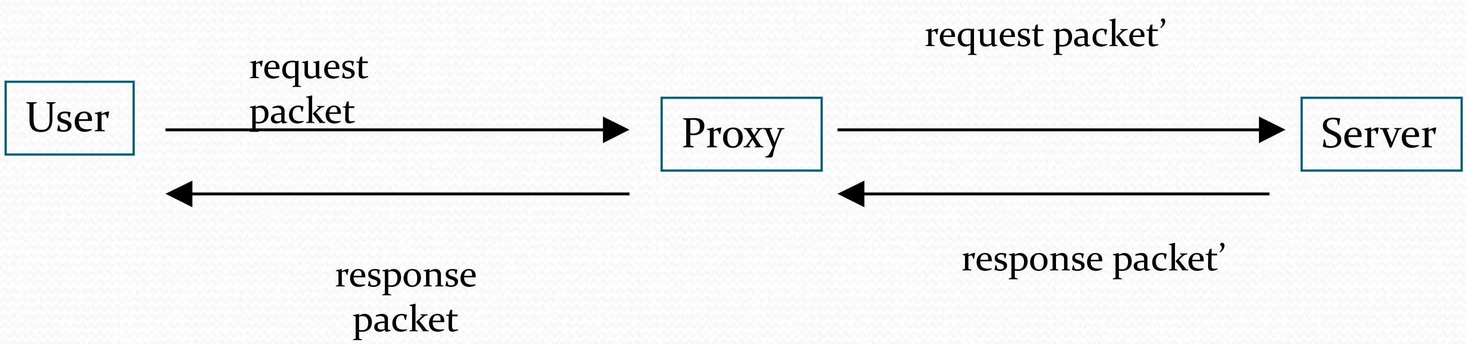
- Part of an overall Firewall strategy
- Sits between the local network and the external network
  - Originally used primarily as a caching strategy to minimize outgoing URL requests and increase perceived browser performance
  - Primary mission is now to insure anonymity of internal users
    - Still used for caching of frequently requested files
    - Also used for content filtering
- Acts as a go-between, submitting your requests to the external network
  - Requests are translated from your IP address to the Proxy's IP address
  - E-mail addresses of internal users are removed from request headers
  - Cause an actual break in the flow of communications

Security Advantages

- Terminates the TCP connection before relaying to target host (in and out)
- Hide internal clients from external network
- Blocking of dangerous URLs
- Filter dangerous content
- Check consistency of retrieved content
- Eliminate need for transport layer routing between networks
- Single point of access, control and logging

TCP Connection Termination

- Both the outgoing and incoming TCP connections are terminated
- prevents a hacker from hijacking a stale connection on a service that is being proxied
- ex . HTTP page request



Connection left open until the proxy closes it after receiving response packet and sending it back to user

Connection only left open until server closes the connection after sending the response packet

TCP Connection Termination

- Transport layer packets don't need to be routed because the entire request must be regenerated
  - Prevents transport layer exploits
    - source routing
    - fragmentation
    - several DoS attacks
- Since some protocols don't have proxies available many admins will enable routing , this alleviates any benefit gained
- Most good proxy servers will allow you to create generic proxies using SOCKS or the redir utility

Performance Aspects

- Caching
  - By keeping local copies of frequently accessed file the proxy can serve those files back to a requesting browser without going to the external site each time, this dramatically improves the performance seen by the end user
  - Only makes sense to implement this at the ISP rather than the small business level because of the number of pages available
  - Because of dynamic content many pages are invalidated in the cache right away
- Load balancing
  - A proxy can be used in a reverse direction to balance the load amongst a set of identical servers (servers inside the firewall and users outside)
  - Used especially with web dynamic content (.asp, .php,.cfm,.jsp)

Proxy Liabilities

- Single point of failure
  - if the proxy dies , no one can get to the external network
- Client software must usually be designed to use a proxy
- Proxies must exist for each service
- Doesn’t protect the OS
  - proxies run at the application level
- Usually optimized for performance rather than security
  - WINGATE was installed to be easy to configure; opened a winsock proxy to the external interface, which let hackers essentially hijack the machine
- Create a service bottleneck
  - solved via parallelism (more proxies, and load balance)

Transparent / Opaque

- Transparent – both parties (local/remote) are unaware that the connection is being proxied
  - Zorp - application layer proxy is transparent

- Opaque – the local party must configure client software to use the proxy
  - client software must be proxy-aware software
  - Netscape proxy server is opaque

- With all of the things modern firewalls can do in the area of redirection you could configure the firewall to redirect all http requests to a proxy
  - no user configuration required (transparent)

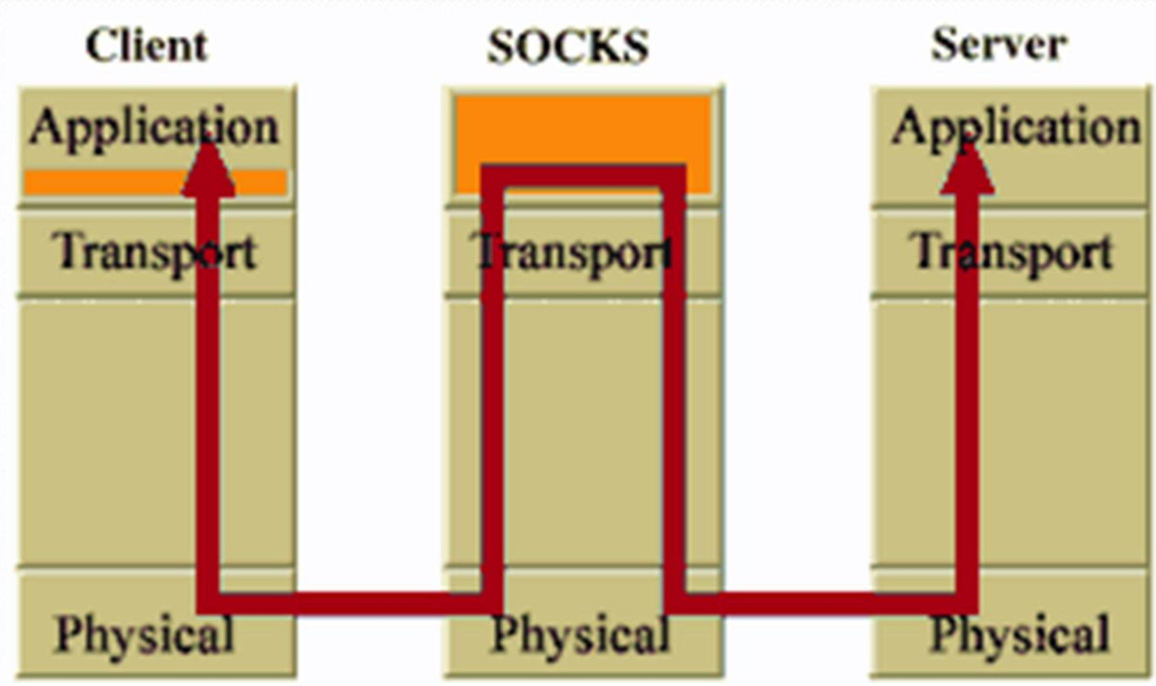
Circuit Level Proxies

- Since some protocols require a real connection between the client and server, a regular proxy can't be used
  - Windows Media Player, Internet Relay Chat (IRC), or Telnet
- Circuit-level proxy servers were devised to simplify matters.
  - Instead of operating at the Application layer, they work as a "shim" between the Application layer and the Transport layer, monitoring TCP handshaking between packets from trusted clients or servers to untrusted hosts, and vice versa. The proxy server is still an intermediary between the two parties, but this time it establishes a virtual circuit between them.
- By using SOCKS (RFC 1928) this can be done
  - SOCKS defines a cross-platform standard for accessing circuit-level proxies
  - SOCKS Version 5 also supports both username/password (RFC 1929) and API-based (RFC 1961) authentication. It also supports both public and private key encryption.
  - SOCKS 5 is capable of solving this problem by establishing TCP connections and then using these to relay UDP data.

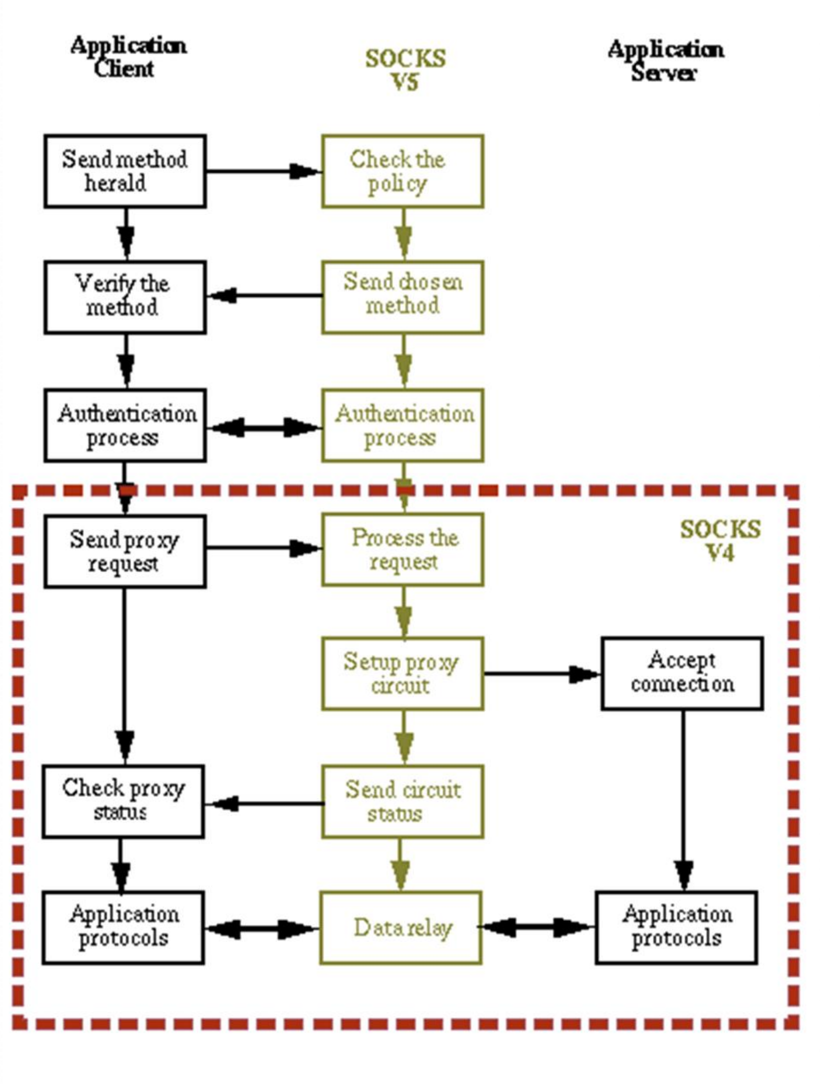
SOCKS based Proxying

- RFC 1928
- Not a true application layer proxy
- SOCKS protocol provides a framework for developing secure communications by easily integrating other security technologies
- SOCKS includes two components
  - SOCKS server
    - implemented at the application layer
  - SOCKS client
    - implemented between the application and transport layers
- The basic purpose of the protocol is to enable hosts on one side of a SOCKS server to gain access to hosts on the other side of a SOCKS Server, without requiring direct IP-reachability.
- Copies packet payloads through the proxy

Socks Architecture



Socks Functionality

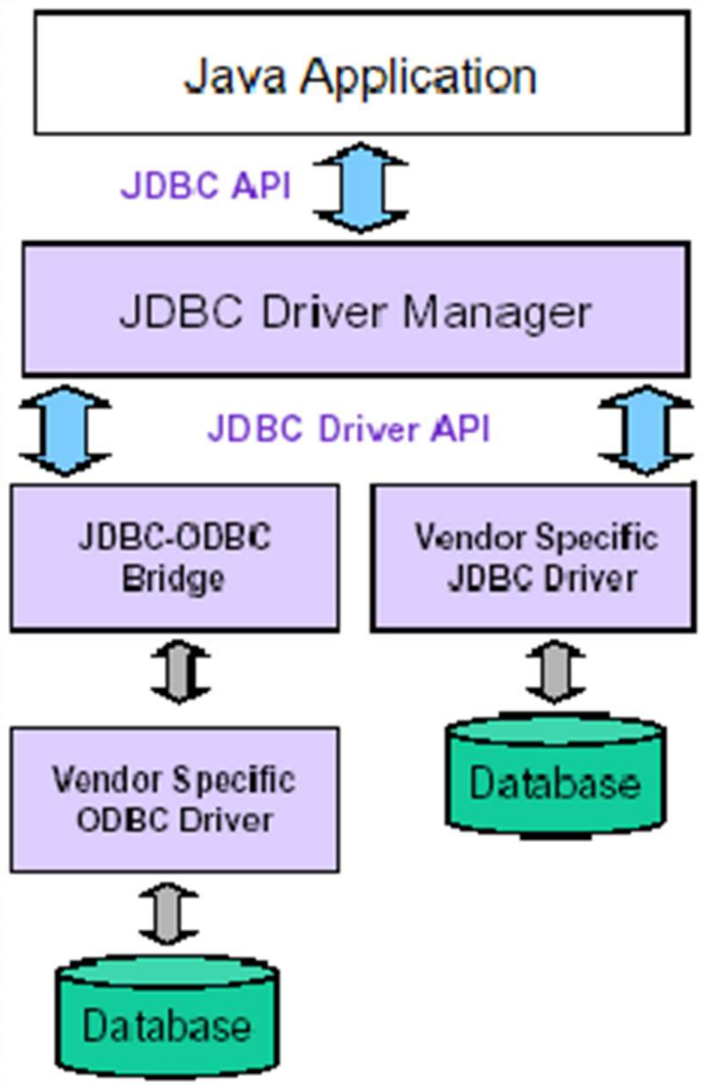


Java Database Connectivity

What is JDBC?

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

General Architecture



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

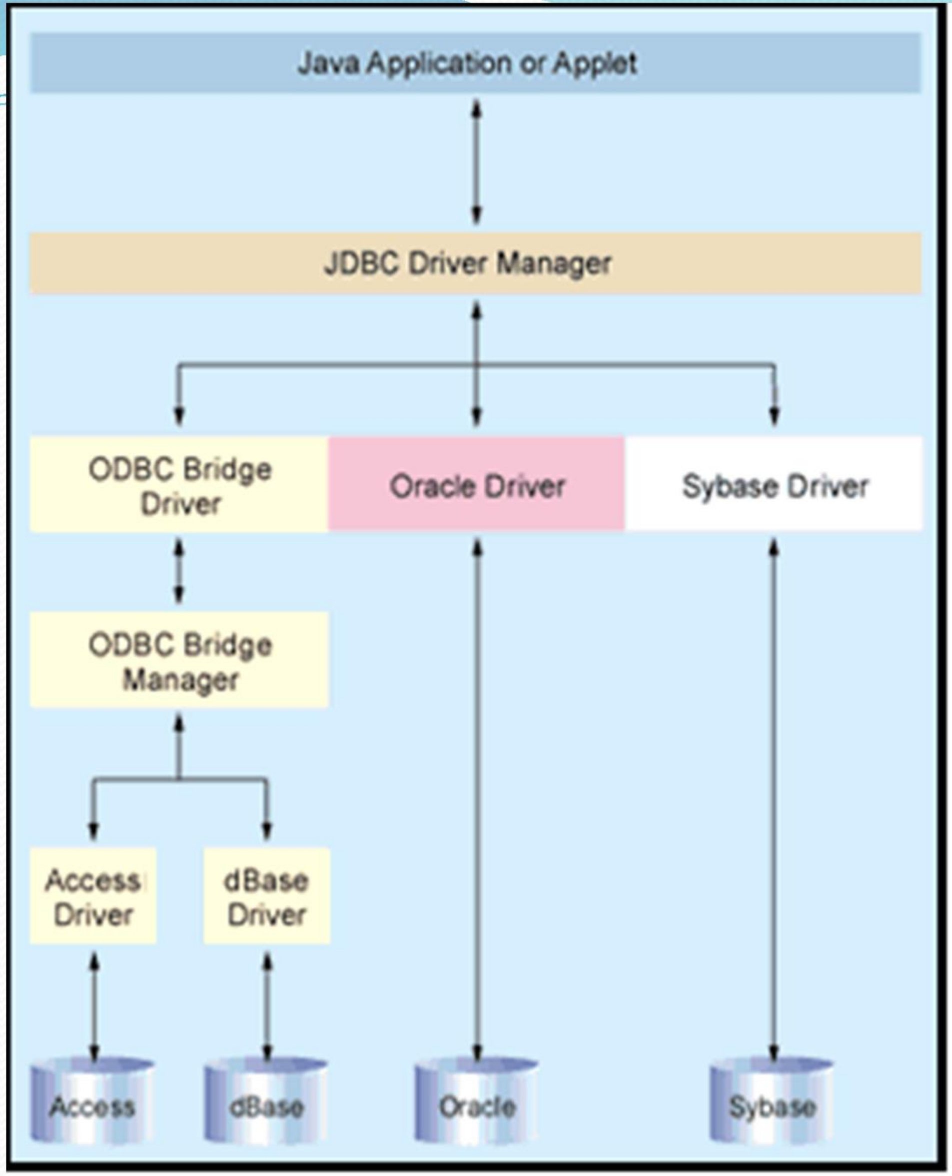


Figure 1. Anatomy of Data Access. The Driver Manager provides a consistent layer between your Java app and back-end database. JDBC works natively (such as with the Oracle driver in this example) or with any ODBC datasource.

Basic steps to use a database in Java

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

1. Establish a connection

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

2. Create JDBC statement(s)

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

Executing SQL Statements

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

- String insertLehigh = "Insert into Lehigh values" + "(123456789,abc,100)";
  stmt.**executeUpdate**(insertLehigh);

Get ResultSet

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

ResultSet rs = Stmt.executeQuery(queryLehigh);
//What does this statement do?

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

Close connection

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

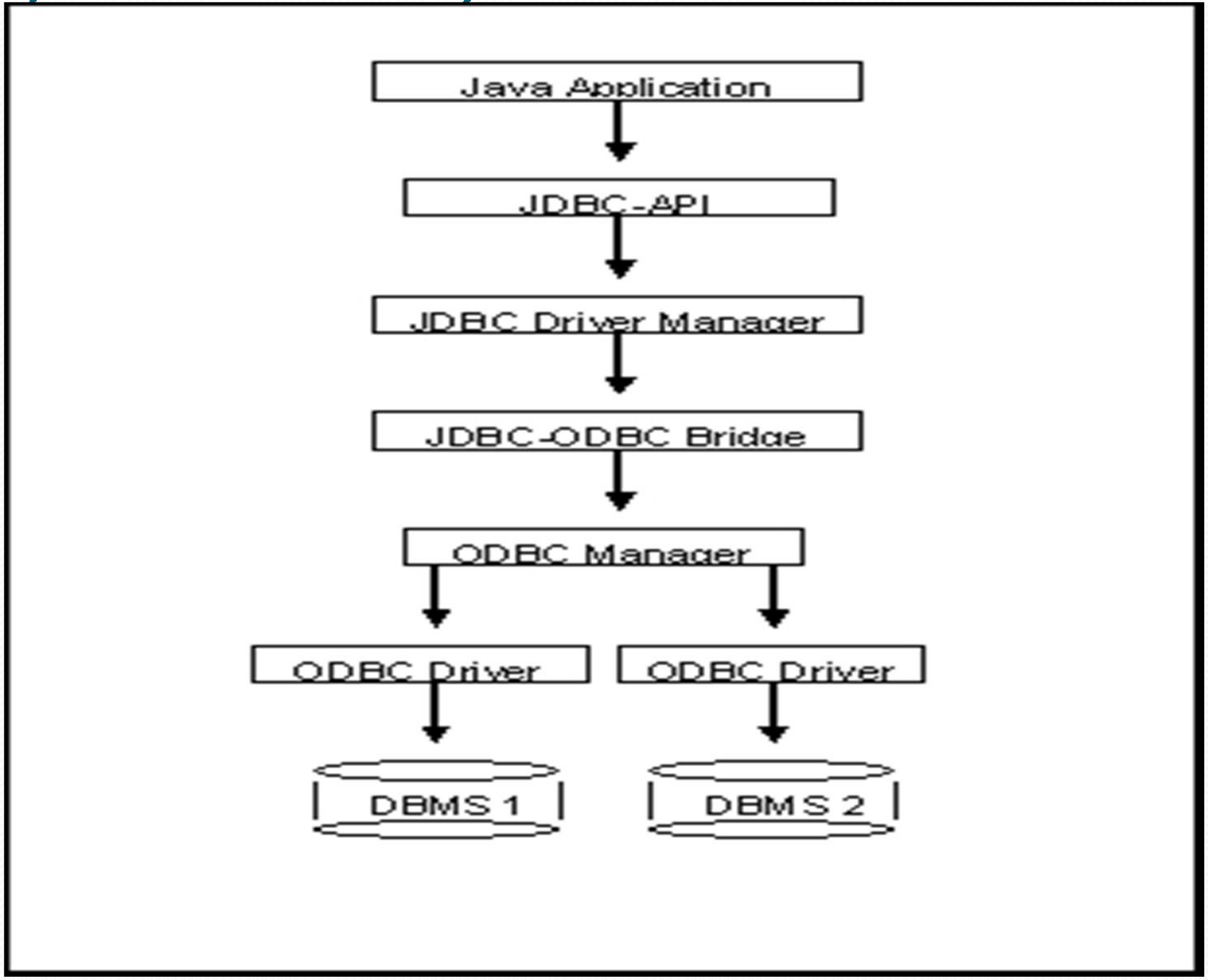
Transactions and JDBC

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

Handling Errors with Exceptions

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

Another way to access database (JDBC-ODBC)



What's a bit different about this architecture?

Why add yet another layer?

Sample program

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

Sample program(cont)

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

Mapping types JDBC - Java

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

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

*SQL3 data type supported in JDBC 2.0

JDBC 2 – Scrollable Result Set

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

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

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

JDBC 2 – Updateable ResultSet

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

Metadata from DB

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

This information is made available through
a DatabaseMetaData object.

Metadata from DB - example

```
…
Connection con = …. ;

DatabaseMetaData dbmd = con.getMetaData();

String catalog = null;
String schema = null;
String table = “sys%”;
String[ ] types = null;

ResultSet rs =
   dbmd.getTables(catalog , schema , table , types );
…
```

JDBC – Metadata from RS

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

JDBC and beyond

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

SQLJ

```
// SQLJ
int n;
#sql { INSERT INTO emp VALUES (:n)};

// vs. straight JDBC
int n;
Statement stmt = conn.prepareStatement
   ("INSERT INTO emp VALUES (?)");
stmt.setInt(1,n);
stmt.execute ();
stmt.close();
```

Remote Method Invocation(RMI)

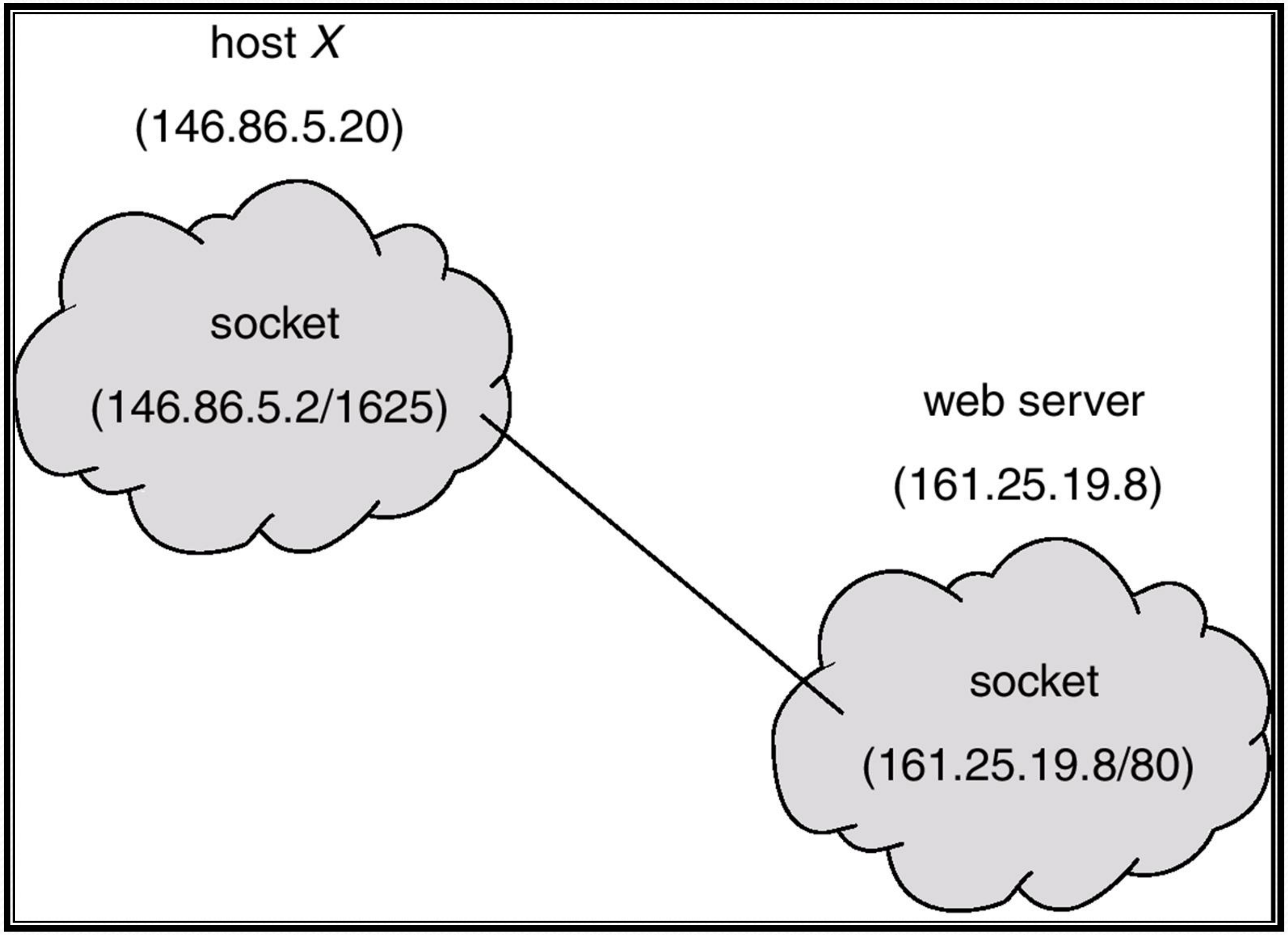
Client-Server Communication

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Sockets

- A socket is defined as an *endpoint for communication*.
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets.
- Considered a low-level form of communication between distributed processes.
    - Sockets allow only an unstructured stream of bytes to be exchanged. It is the responsibility of the client or server application to impose a structure on the data.

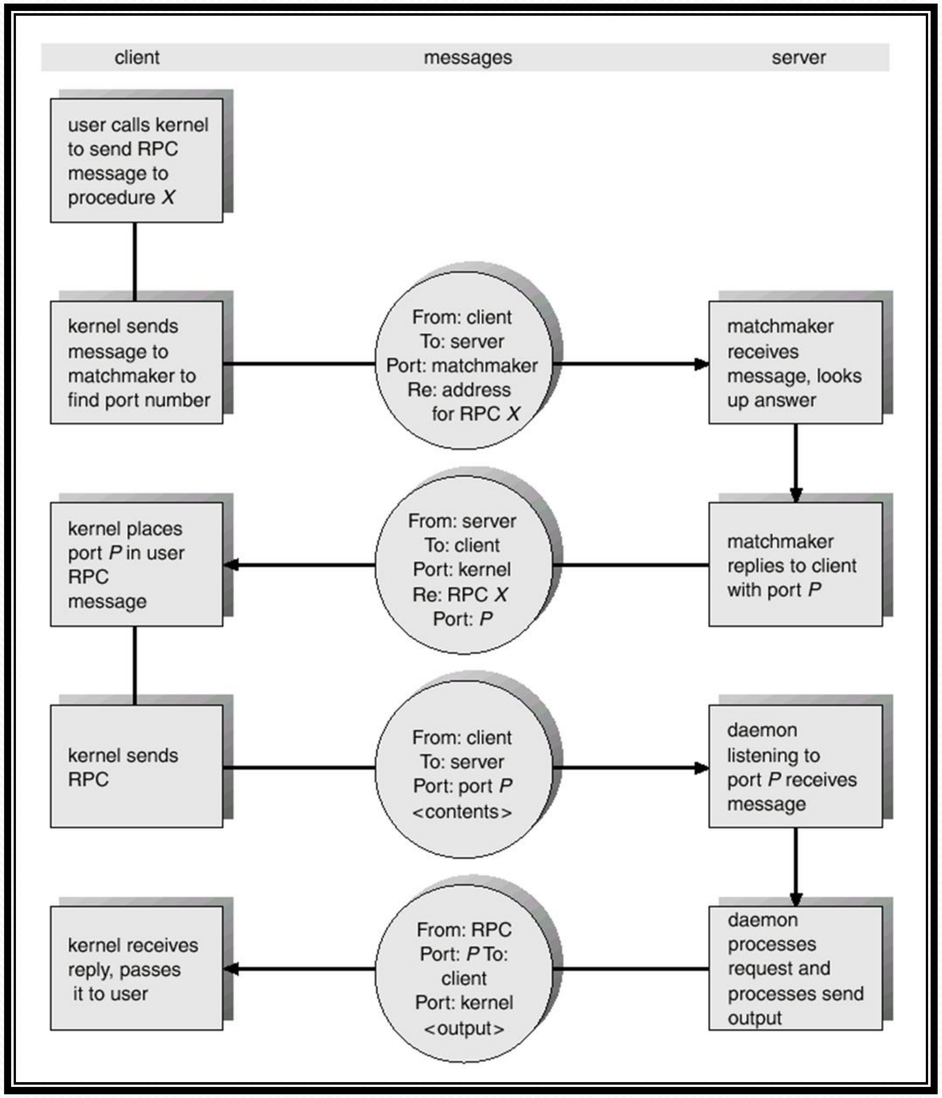
Socket Communication



Remote Procedure Calls

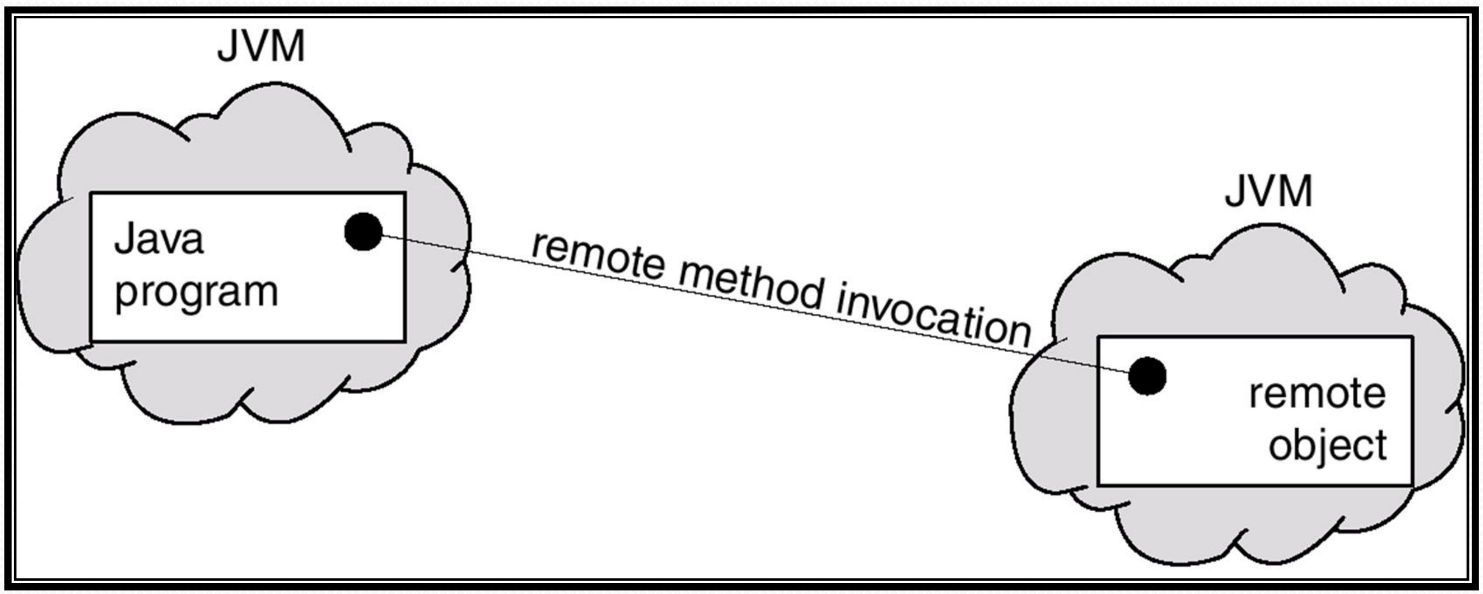
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stub** – client-side proxy for the actual procedure on the server. Server has a similar stub as well.
- The client-side stub locates the server and *marshals* the parameters.
- The server-side stub receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
- External data representation (XDR)→ I.e most-significant (big-endian), least-significant(little-endian)

Execution of RPC

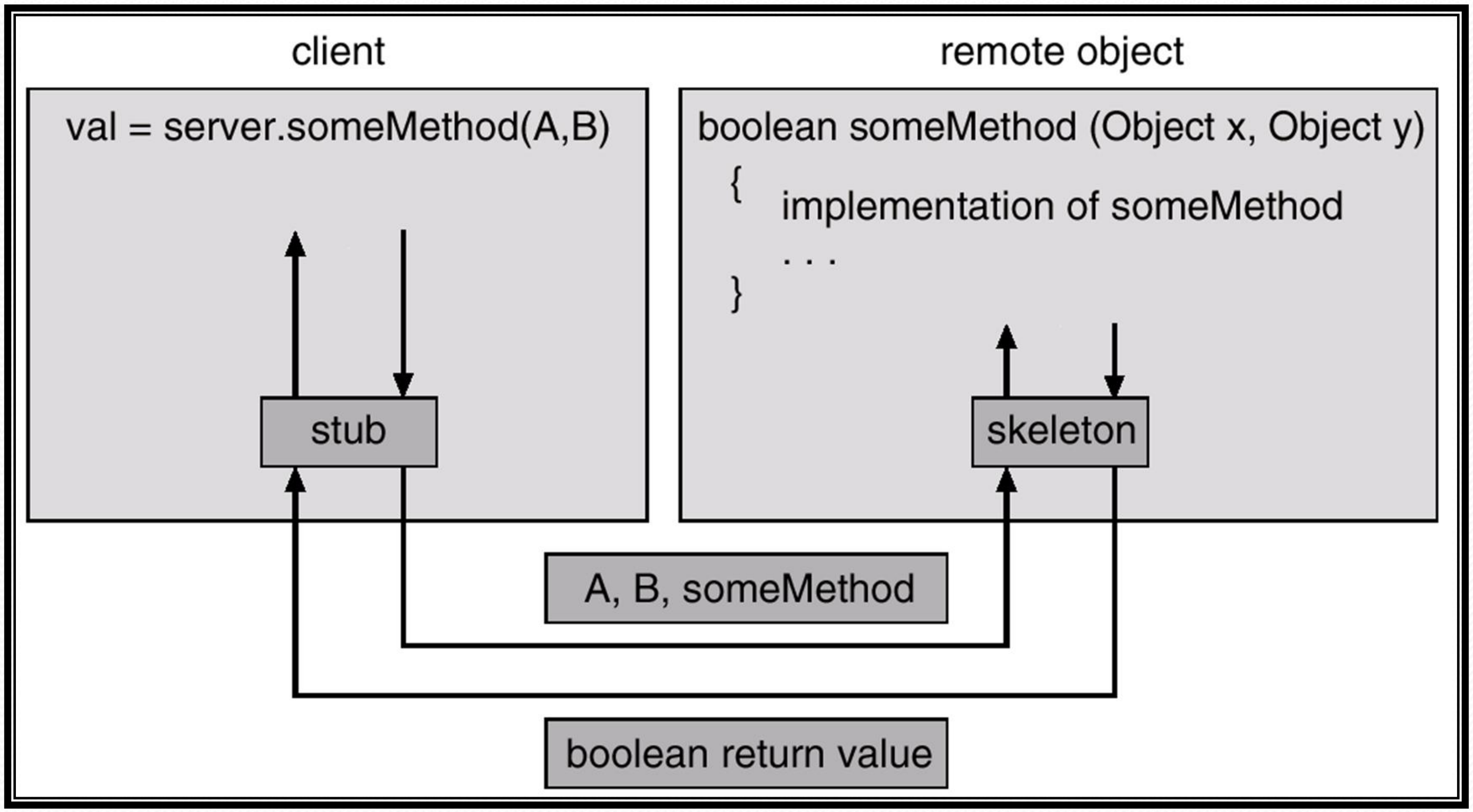


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



Marshalling Parameters



Remote Method Invocation

- RMI and RPC differs in two ways:

1. RPCs support procedural programming whereby only remote procedures or functions may be called. RMI is object based: It supports invocation of methods on remote objects.
2. The parameters to remote procedures are ordinary data structures in RPC; with RMI it is possible to pass objects as parameters to remote methods.

- If the marshaled parameters are local (non remote) objects, they are passed by copy using a technique known as object serialization.
  - Object serialization allowed the state of an object to be written toa byte stream.

Introduction to RMI

- Remote Method Invocation (RMI)
  - Allows remote method calls
    - Objects in different programs can communicate
    - Method calls appear same as those in same program
  - Based on Remote Procedure Calls (RPC)
    - Developed in 1980's
    - Allows procedural program (like C) to call function on another computer
    - Performs networking and marshalling of data (packaging arguments and return values)
    - Not compatible with objects
    - Interface Definition Language required - describe functions
  - RMI is Java's implementation of RPC

Introduction to RMI

- RMI
  - Register method as remotely accessible
    - Client can look up method and receive a reference
    - Use reference to call method
    - Syntax same as a normal method call
  - Marshalling of data
    - Can transfer objects as well
    - Class **`ObjectOutputStream`** converts **`Serializable`** object into stream of bytes
      - Transmit across network
    - Class **`ObjectInputStream`** reconstructs object
  - No Interface Definition Language needed
    - Use Java's own interface

Defining the Remote Interface

- First step
  - Define remote interface that describes remote methods
    - Client calls remote methods, server implements them
- To create a remote interface
  - Define interface that extends interface **`Remote`** (**`java.rmi`**)
    - Tagging interface - no methods to define
    - An object of a class that implements interface Remote directly or indirectly is a *remote object* and can be accesses from any JVM.
  - Each method in **`Remote`** interface must **`throw RemoteException`**
    - Potential network errors

Defining the Remote Interface

- Interface **`TemperatureServer`**
  - Extends **`Remote`**
  - Describes method **`getWeatherInfo`**

Implementing the Remote Interface

- Define **`TemperatureServerImpl`**
  - Implements Remote interface TemperatureServer
  - Client interacts with **`TemperatureServerImpl`** object
  - Uses array of **`WeatherInfo`** objects to store data
    - Copy sent to client when calls **`getWeatherInfo`**

Implementing the Remote Interface

```
public class TemperatureServerImpl extends UnicastRemoteObject
                                       implements TemperatureServer {
```

- **`UnicastRemoteObject`**
  - Provides functionality for remote objects
  - Constructor exports object so it can receive remote calls
    - Wait for client on anonymous port number

```
    public TemperatureServerImpl() throws RemoteException
```

  - Subclass constructors must throw **`RemoteException`**s

```
         URL url = new URL(
            "http://iwin.nws.noaa.gov/iwin/us/traveler.html" );
```

- **`URL`** object
  - Contains URL for Traveler's Forecast web page
  - Throws **`MalformedURLException`**

Implementing the Remote Interface

```
        BufferedReader in =
           new BufferedReader(
              new InputStreamReader( url.openStream() ) );
```

- Open connection to file specified by URL
- Method **`openStream`** (class **`URL`**)
  - Opens network connection using Http protocol
  - If successful, **`InputStream`** object returned (else **`IOException`**)
- **`InputStreamReader`**
  - Translates bytes to Unicode characters
- **`BufferedReader`**
  - Buffers characters
  - Method **`readLine`**
    - Returns one line as a **`String`**

Implementing the Remote Interface

```
            String separator = "</PRE><HR> <BR><PRE>";
            while ( !in.readLine().startsWith( separator ) )
               ;    // do nothing
```

- Sentinel **`String`** to find relevant part of HTML code
  - **`readLine`** until sentinel found

```
            String s1 =
               "CITY            WEA     HI/LO   WEA     HI/LO";
```

- A string used as column head
  - Second "**`WEA    HI/LO"`**" is for next day, we do not use
- Locate column head and get first city's info

```
            inputLine = in.readLine();  // get first city's info
```

Implementing the Remote Interface

```
            WeatherInfo w = new WeatherInfo(
               inputLine.substring( 0, 16 ),
               inputLine.substring( 16, 22 ),
               inputLine.substring( 23, 29 ) );
            cityVector.addElement( w ); // add to Vector
```

- **`WeatherInfo`** objects
  - City name, temperature, description of weather
    - Method **`substring`** to extract data from line
  - Store all **`WeatherInfo`** objects in a **`Vector`**

```
            weatherInformation[ i ] =
               ( WeatherInfo ) cityVector.elementAt( i );
```

- Store data in **`WeatherInfo`** array
  - **`elementAt`** returns **`Object`** (must be cast)

- Close connection

```
         in.close();  // close connection to NWS server
```

Implementing the Remote Interface

```
116            String serverObjectName = "//localhost/TempServer";
```

- Name of server object
  - Used by clients to connect
  - `//`*host*`:`*port*`/`*remoteObjectName*
    - host - computer running registry for remote objects
      - Where remote object executes
    - port - port number of registry on host (1099 default)
    - remoteObjectName - client uses to locate object
- Registry managed by **`rmiregistry`** (located at host and port)
  - Remote objects register with it, clients use it to locate service
  - **`localhost`** (same computer)
    - Same as IP **`127.0.0.1`**

Implementing the Remote Interface

```
TemperatureServerImpl temp =
   new TemperatureServerImpl();
String serverObjectName = "//localhost/TempServer";
Naming.rebind( serverObjectName, temp );
```

- **`static`** method **`rebind`** (class **`Naming`**)
  - Binds object to **`rmiregistry`**
  - Named **`//localhost/TempServer`**
    - Name used by client
  - **`rebind`** replaces any previous objects with same name
    - Method **`bind`** does not

Define the client

- Next step
  - Client code to get weather info from **`TemperatureServerImpl`**
  - Calls **`getWeatherInfo`** through RMI
  - Graphically display weather info
    - Class **`WeatherItem`** (extends **`JLabel`**) stores info about each city
    - Display name, High/low, and image (depending on conditions)

Define the client

```java
   private void getRemoteTemp( String ip )
         String serverObjectName = "//" + ip + "/TempServer";
```

- Can specify IP address at command line (more later)

```java
         TemperatureServer mytemp = ( TemperatureServer )
            Naming.lookup( serverObjectName );
```

- **`static`** method **`lookup`** (class **`Naming`**)
- Returns reference to **`Remote`** object
  - Cast to **`TemperatureServer`**
- Reference may be used as normal
  - Only difference that copy of array returned

```java
         WeatherInfo weatherInfo[] = mytemp.getWeatherInfo();
```

```
        JPanel p = new JPanel();
        for ( int i = 0; i < w.length; i++ ) {
           w[ i ] = new WeatherItem( weatherInfo[ i ]
           p.add( w[ i ] );
        }
```

- Add **WeatherItems**
  - Initialize with **WeatherInfo**

```
   public static void main( String args[] )
   {
      TemperatureClient gt = null;
      if ( args.length == 0 )
         gt = new TemperatureClient( "localhost" );
      else
         gt = new TemperatureClient( args[ 0 ] );
```

- **main**
  - Passes command line argument (ip) to constructor
  - **localhost** default

Define the client

- Class **`WeatherItem`**
  - **`extends JLabel`**
  - **`static`** initializer block
    - For complex initialization of **`static`** variables
    - **`backgroundImage`** - **`ImageIcon`**, has background
    - **`weatherImages`** - **`ImageIcon`** array, holds weather images

```
static {
   backgroundImage = new ImageIcon( "images/back.jpg" );
   weatherImages =
      new ImageIcon[ weatherImageNames.length ];

   for ( int i = 0; i < weatherImageNames.length; ++i )
      weatherImages[ i ] = new ImageIcon(
         "images/" + weatherImageNames[ i ] + ".jpg" );
}
```

Define the client

- Array of descriptions and matching array of images
  - **`weatherConditions`** and **`weatherImages`**

```
public WeatherItem( WeatherInfo w )
   weatherInfo = w;
   for ( int i = 0; i < weatherConditions.length;
      if ( weatherConditions[ i ].equals(
              weatherInfo.getDescription().trim() ) )
            weather = weatherImages[ i ];
```

- Tests **`WeatherInfo`** object, loads proper image

Compile and Execute the Server and the Client

- Build and execute application
  - All pieces in place
  - Compile classes with **`javac`**
  - Remote server class (**`TemperatureServerImpl`**) compiled with **`rmic`** compiler
    - Makes a stub class - allows client to access remote methods and server to provide its services
    - Gets remote method calls, passes to RMI system, which performs networking
    - **`rmic TemperatureServerImpl`**

Compile and Execute the Server and the Client

- Start **`rmiregistry`**
  - Type **`rmiregistry`** at command window
    - No text in response

Compile and Execute the Server and the Client

- Must bind remote server object
  - Run TemperatureServerImpl application

    `java TemperatureServerImpl`
  - Superclass **`UnicastRemoteObject`**
    - Constructor exports remote object
    - **`main`** binds object to **`rmiregistry`**
    - **`rmiregistry`** provides host and port number to clients

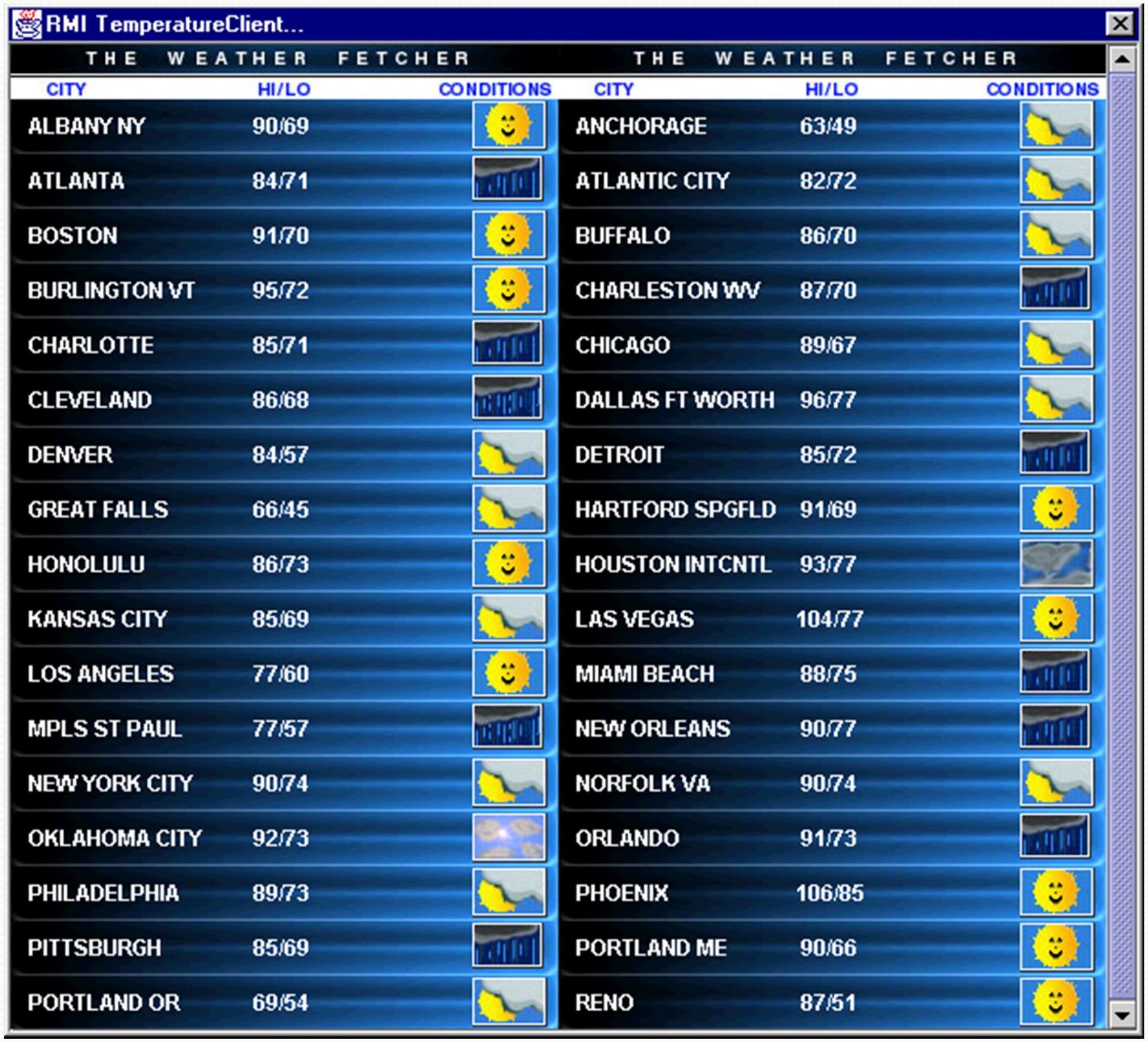
```
JAVA
e:\books\jhtp3\examples\ch20>java TemperatureServerImpl
Initializing server: please wait.
Updating weather information...
Finished Processing Data.
The Temperature Server is up and running.
```

Compile and Execute the Server and the Client

- Execute TemperatureClient
  - java TemperatureClient
  - If server on different machine, specify IP on command line

    `java TemperatureClient 192.168.150.4`
  - Result on next slide

Program Output



Servlets

Introduction

- Networking
  - Massive, complex topic
  - Java networking in several packages
- **`java.net`**
  - Socket based communications
    - View networking as streams of data
      - Reading/writing to socket like reading/writing to file
  - Packet based communications
    - Transmit packets of information.
- Remote Method Invocation (RMI)
  - Objects in different Java Virtual Machines can communicate

Introduction

- Client-server relationship
  - Client request action
  - Server performs action, responds to client
  - This view foundation of servlets
    - Highest-level view of networking
    - Servlet extends functionality of server
  - Useful for database-intensive applications
    - Thin clients - little client-side support needed
    - Server controls database access
      - Logic code written once, on server

Overview of Servlet Technology

- Servlets
  - Analog to applets
    - Execute on server's machine, supported by most web servers
  - Demonstrate communication via HTTP protocol
    - Client sends HTTP request
    - Server receives request, servlets process it
    - Results returned (HTML document, images, binary data)

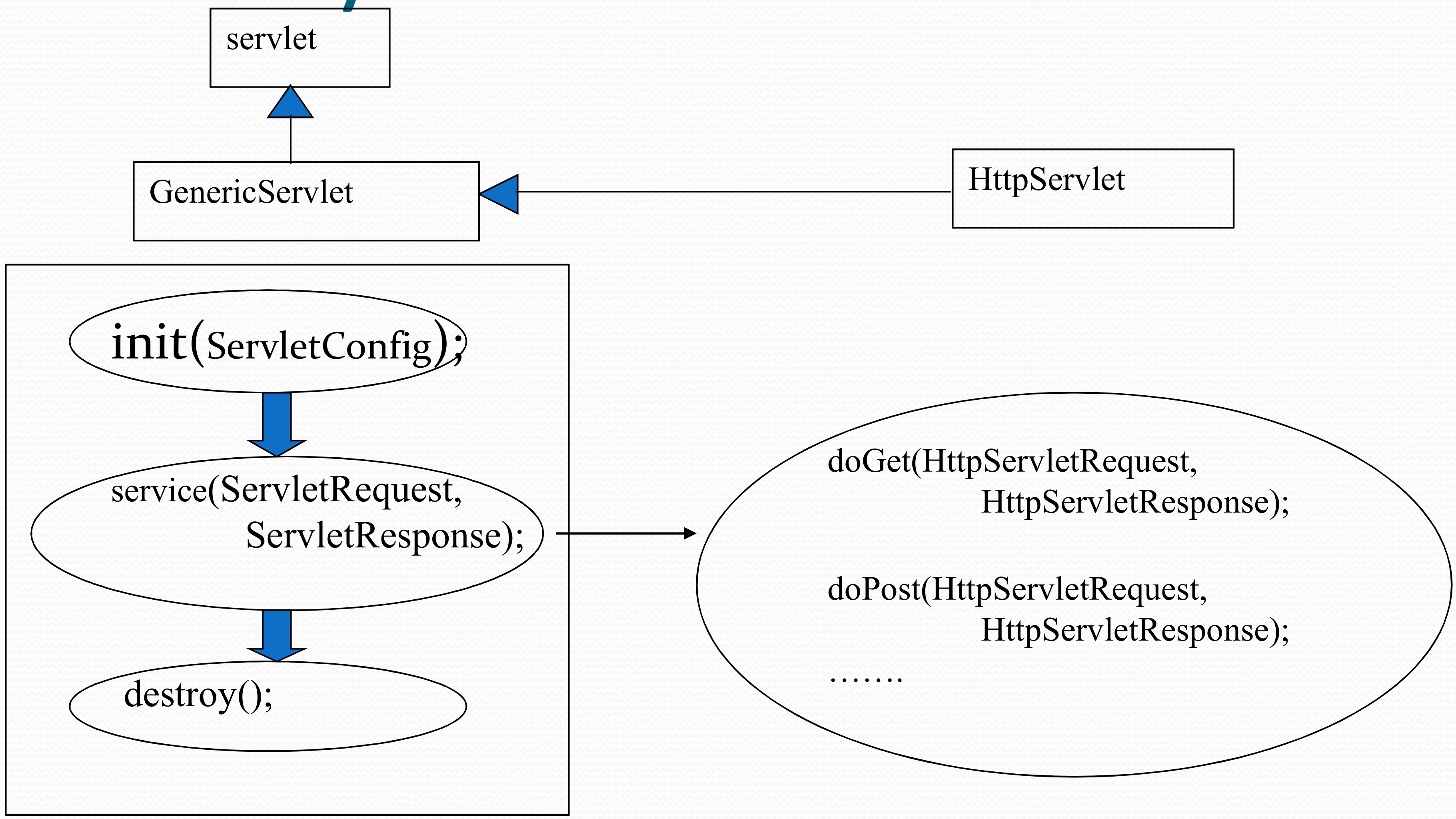
The Servlet API

- **`Servlet`** interface
  - Implemented by all servlets
  - Many methods invoked automatically by server
    - Similar to applets (**`paint`**, **`init`**, **`start`**, etc.)
  - **`abstract`** classes that implement Servlet
    - **`GenericServlet`** (**`javax.servlet`**)
    - **`HTTPServlet`** (**`javax.servlet.http`**)
  - Examples in chapter extend **`HTTPServlet`**
- Methods
  - **`void init( ServletConfig config )`**
    - Automatically called, argument provided

The Servlet API

- Methods
  - `ServletConfig getServletConfig()`
    - Returns reference to object, gives access to config info
  - `void service ( ServletRequest request, ServletResponse response )`
    - Key method in all servlets
    - Provide access to input and output streams
      - Read from and send to client
  - `void destroy()`
    - Cleanup method, called when servlet exiting

Life Cycle of Servlet



`HttpServlet` Class

- **`HttpServlet`**
  - Base class for web-based servlets
  - Overrides method **`service`**
    - Request methods:
      - **`GET`** - retrieve HTML documents or image
      - **`POST`** - send server data from HTML form
  - Methods **`doGet`** and **`doPost`** respond to **`GET`** and **`POST`**
    - Called by **`service`**
    - Receive **`HttpServletRequest`** and **`HttpServletResponse`** (return **`void`**) objects

HttpServletRequest Interface

- **HttpServletRequest** interface
  - Object passed to **doGet** and **doPost**
  - Extends **ServletRequest**
- Methods
  - **String getParameter(String name)**
    - Returns value of parameter **name** (part of **GET** or **POST**)
  - **Enumeration getParameterNames()**
    - Returns names of parameters (**POST**)
  - **String[] getParameterValues(String name)**
    - Returns array of strings containing values of a parameter
  - **Cookie[] getCookies()**
    - Returns array of **Cookie** objects, can be used to identify client

HttpServletResponse Interface

- **HttpServletResponse**
  - Object passed to **doGet** and **doPost**
  - Extends **ServletResponse**
- Methods
  - **void addCookie(Cookie cookie)**
    - Add **Cookie** to header of response to client
  - **ServletOutputStream getOutputStream()**
    - Gets byte-based output stream, send binary data to client
  - **PrintWriter getWriter()**
    - Gets character-based output stream, send text to client
  - **void setContentType(String type)**
    - Specify MIME type of the response (Multipurpose Internet Mail Extensions)
    - MIME type "text/html" indicates that response is HTML document.
    - Helps display data

Handling HTTP GET Requests

- HTTP **`GET`** requests
  - Usually gets content of specified URL
    - Usually HTML document (web page)
- Example servlet
  - Handles HTTP **`GET`** requests
  - User clicks **Get Page** button in HTML document
    - **`GET`** request sent to servlet **`HTTPGetServlet`**
  - Servlet dynamically creates HTML document displaying "Welcome to Servlets!"

Handling HTTP GET Requests

```
import javax.servlet.*;
import javax.servlet.http.*;
```

- Use data types from **`javax.servlet`** and **`javax.servlet.http`**

```
public class HTTPGetServlet extends HttpServlet {
```

- **`HttpServlet`** has useful methods, inherit from it

```
   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
      throws ServletException, IOException
```

- Method **`doGet`**
  - Responds to **`GET`** requests
  - Default action: **`BAD_REQUEST`** error (file not found)
  - Override for custom **`GET`** processing
  - Arguments represent client request and server response

Handling HTTP GET Requests

```
response.setContentType( "text/html" );  // content type
```

- **`setContentType`**
  - Specify content
  - **`text/html`** for HTML documents

```
PrintWriter output;
output = response.getWriter();           // get writer
```

- **`getWriter`**
  - Returns **`PrintWriter`** object, can send text to client
  - **`getOutputStream`** to send binary data (returns **`ServletOutputStream`** object)

Handling HTTP GET Requests

```
19        buf.append( "<HTML><HEAD><TITLE>\n" );
20        buf.append( "A Simple Servlet Example\n" );
21        buf.append( "</TITLE></HEAD><BODY>\n" );
22        buf.append( "<H1>Welcome to Servlets!</H1>\n" );
23        buf.append( "</BODY></HTML>" );
```

- Lines 19-23 create HTML document

```
24        output.println( buf.toString() );
25        output.close();    // close PrintWriter stream
```

  - **`println`** sends response to client
  - **`close`** terminates output stream
    - Flushes buffer, sends info to client

Handling HTTP GET Requests

- Running servlets
  - Must be running on a server
    - Check documentation for how to install servlets
    - Tomcat web server
    - Apache Tomcat

Handling HTTP GET Requests

- Port number
  - Where server waits for client (handshake point)
  - Client must specify proper port number
    - Integers 1 - 65535, 1024 and below usually reserved
  - Well-known port numbers
    - Web servers - port 80 default
    - JSDK/Apache Tomcat 4.0 Webserver- port 8080
      - Change in **`default.cfg`** (**`server.port=8080`**)

Handling HTTP POST Requests

- HTTP **`POST`**
  - Used to post data to server-side form handler (i.e. surveys)
  - Both **`GET`** and **`POST`** can supply parameters
- Example servlet
  - Survey
    - Store results in file on server
  - User selects radio button, presses **Submit**
    - Browser sends **`POST`** request to servlet
  - Servlet updates responses
    - Displays cumulative results

Session Tracking

- Web sites
  - Many have custom web pages/functionality
    - Custom home pages - http://my.yahoo.com/
    - Shopping carts
    - Marketing
  - HTTP protocol does not support persistent information
    - Cannot distinguish clients
- Distinguishing clients
  - Cookies
  - Session Tracking

Cookies

- Cookies
  - Small files that store information on client's computer
  - Servlet can check previous cookies for information
- Header
  - In every HTTP client-server interaction
  - Contains information about request (**`GET`** or **`POST`**) and cookies stored on client machine
  - Response header includes cookies servers wants to store
- Age
  - Cookies have a lifespan
  - Can set maximum age
    - Cookies can expire and are deleted

Cookies

- Example
  - Demonstrate cookies
  - Servlet handles both POST and GET requests
  - User selects programming language (radio buttons)
    - **`POST`** - Add cookie containing language, return HTML page
    - **`GET`** - Browser sends cookies to servlet
      - Servlet returns HTML document with recommended books
  - Two separate HTML files
    - One invokes **`POST`**, the other **`GET`**
    - Same **`ACTION`** - invoke same servlet

Cookies

```
   public void doPost( HttpServletRequest request,
                       HttpServletResponse response )
```

```
      String language = request.getParameter( "lang" );
```

- Method **`doPost`**
  - Get language selection

```
      Cookie c = new Cookie( language, getISBN( language ) );
      c.setMaxAge( 120 );  // seconds until cookie removed
```

- Cookie constructor
  - **`Cookie ( name, value )`**
  - **`getISBN`** is utility method
  - **`setMaxAge( seconds )`** - deleted when expire

Cookies

```
      response.addCookie( c );  // must precede getWriter
```

- Add cookie to client response
  - Part of HTTP header, must come first
  - Then HTML document sent to client

```
   public void doGet( HttpServletRequest request,
                      HttpServletResponse response )
```

```
      Cookie cookies[];
      cookies = request.getCookies(); // get client's cookies
```

- Method **`doGet`**
- **`getCookies`**
  - Returns array of **`Cookies`**

Cookies

```
      if ( cookies != null ) {
            output.println(
               cookies[ i ].getName() + " How to Program. " +
               "ISBN#: " + cookies[ i ].getValue() + "<BR>" );
```

- **`Cookie`** methods
  - **`getName, getValue`**
  - Used to determine recommended book
  - If cookie has expired, does not execute

Session Tracking with HttpSession

- **`HttpSession`** (**`javax.servlet.http`**)
  - Alternative to cookies
  - Data available until browsing ends
- Methods
  - Creation

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

  - **`getSession( createNew )`**
    - Class **`HttpServletRequest`**
    - Returns client's previous **`HttpSession`** object
    - **`createNew`** - if **`true`**, creates new **`HttpSession`** object if does not exist

Session Tracking with HttpSession

```
      session.putValue( language, getISBN( language ) );
```

- **`putvalue( name, value )`**
  - Adds a name/value pair to object

```
      valueNames = session.getValueNames();
      for ( int i = 0; i < valueNames.length; i++ ) {
         String value =
            (String) session.getValue( valueNames[ i ] );
```

- **`getValueNames()`**
  - Returns array of **`String`**s with names
- **`getValue( name )`**
  - Returns value of **`name`** as an **`Object`**
  - Cast to proper type

Session Tracking with HttpSession

- Redo previous example
  - Use **`HttpSession`** instead of cookies
  - Use same HTML files as before
    - Change **`ACTION`** URL to new servlet

Java Beans

Definition: What is a Bean?

- **Definition:** A Java Bean is a reusable software component that can be visually manipulated in builder tools.

Introduction

- JavaBeans (beans)
  - Reusable software component model
  - Assemble predefined components
    - Create powerful applications and applets
  - Graphical programming and design environments
    - Builder tools
    - Support beans, reuse and integrate components
  - Component assembler
    - Programmer who use defined components
      - Work on design of GUI and functionality
    - Do not need to know implementation
      - Just need to know services

Introduction

- Example of bean concept
  - Have animation bean
    - Want two buttons, start and stop
  - With beans, can "hook up" buttons to **`startAnimation`** and **`stopAnimation`** methods
    - When pressed, method called
    - Builder tool does work
  - Use previously defined, reusable components
  - Little or no code must be written
    - Component assembler can "connect the dots"
  - More info about beans at
    - **`http://java.sun.com/beans/`**

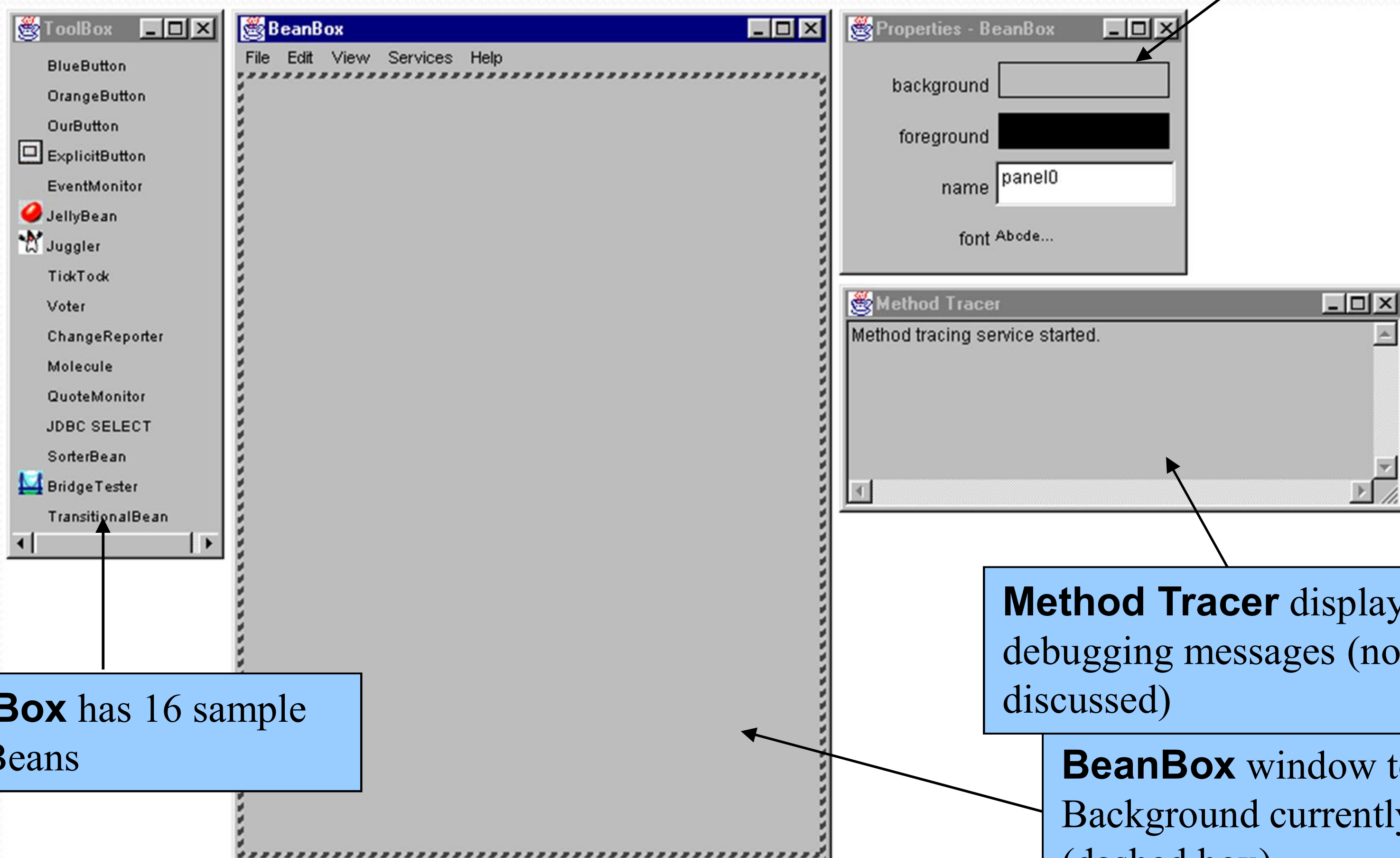
BeanBox Overview

- BeanBox installation
  - Free utility from JavaBeans Development Kit (BDK)

  `http://java.sun.com/beans/software/index.html`
    - Windows, Solaris, and platform independent versions
    - In Windows version, minor bug
      - Do not install in directory with spaces in name
    - To run, go to install directory, beanbox subdirectory, load **`run.bat`** (or **`run.sh`**)
- BeanBox test container for JavaBeans
  - Preview how bean will be displayed
  - Not meant to be robust development tool

BeanBox Overview

- Use screen captures from Windows
  - Start application, following appears:



Properties customizes selected bean.

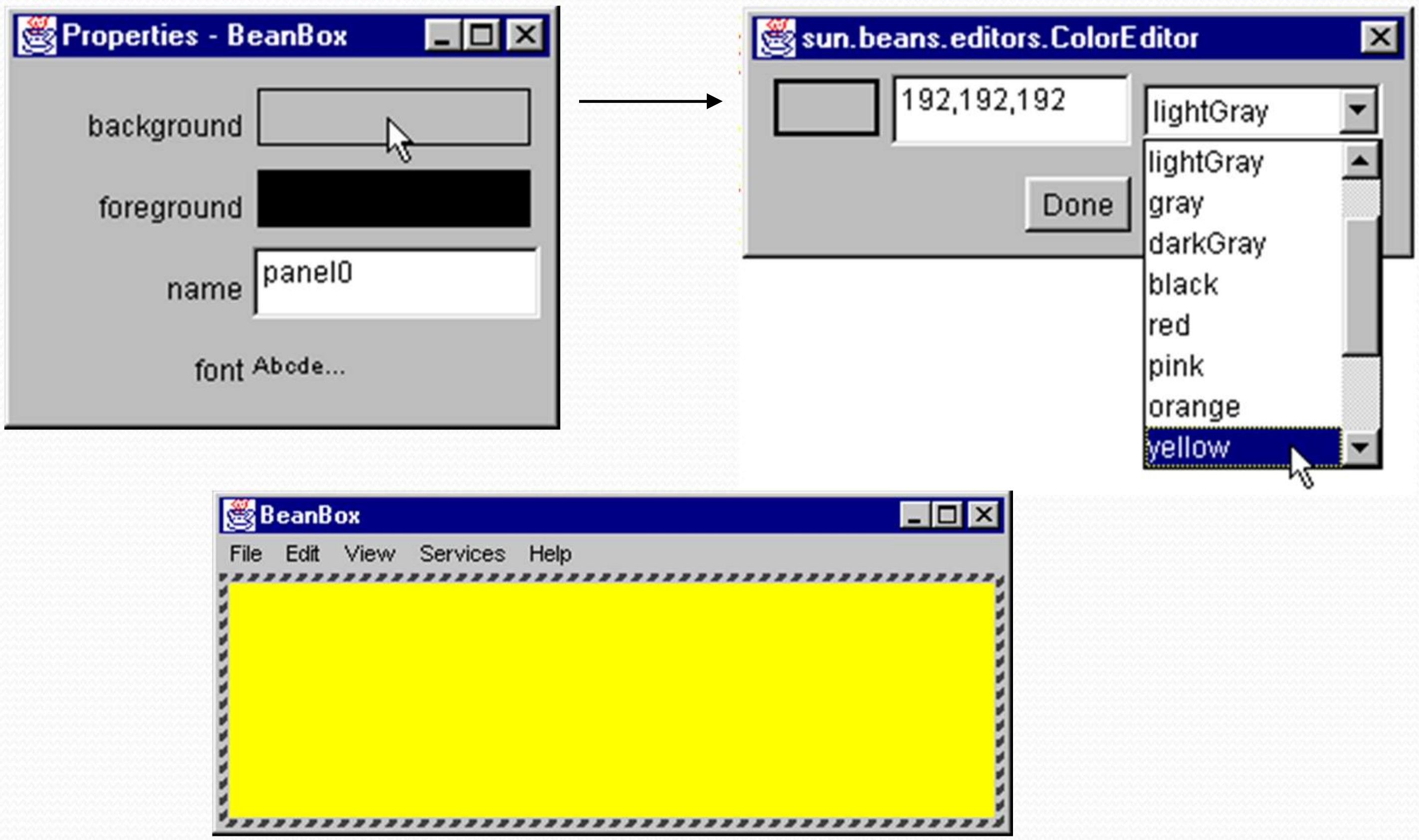
Method Tracer displays debugging messages (not discussed)

ToolBox has 16 sample JavaBeans

BeanBox window tests beans. Background currently selected (dashed box).

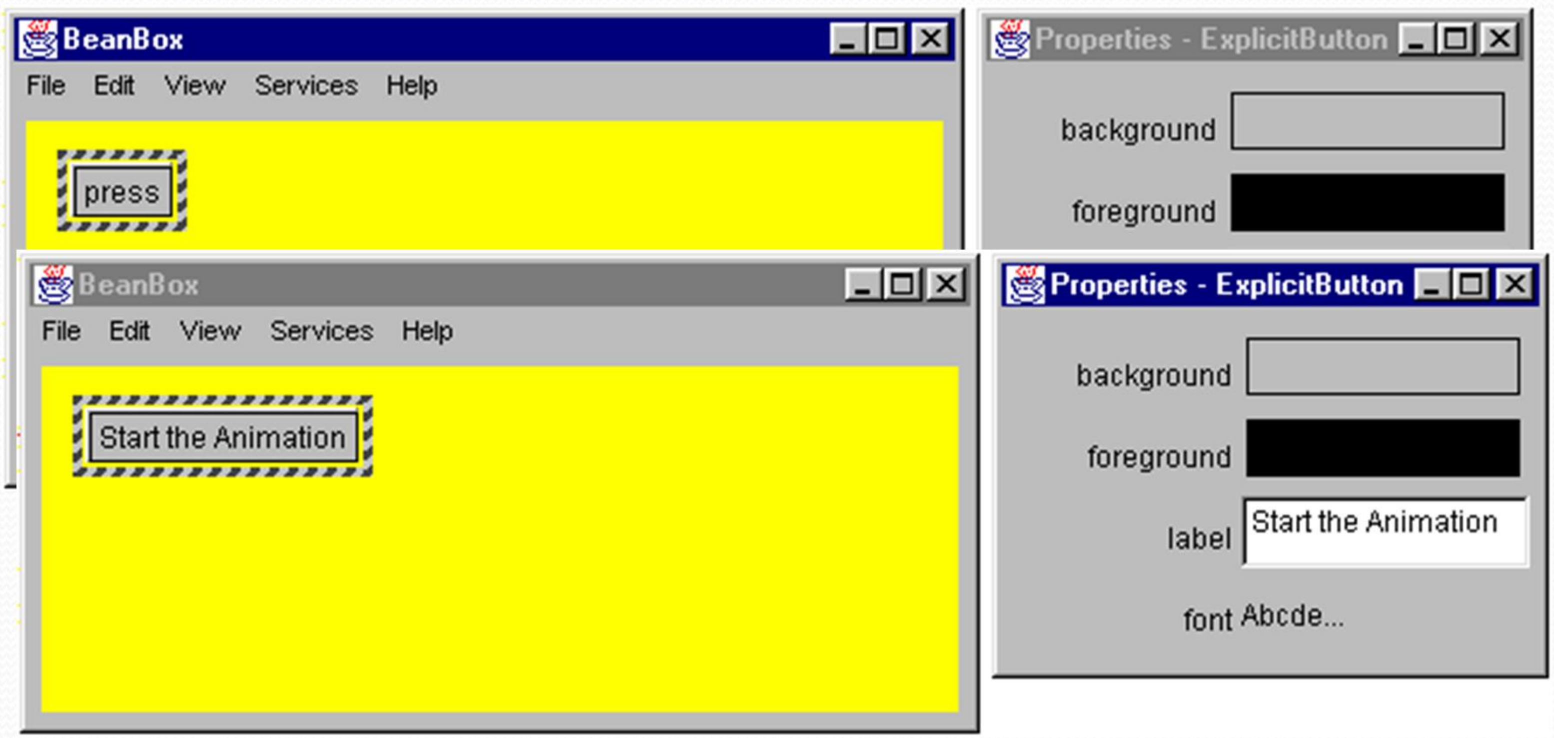
BeanBox Overview

- Initially, background selected
  - Customize in **Properties** box



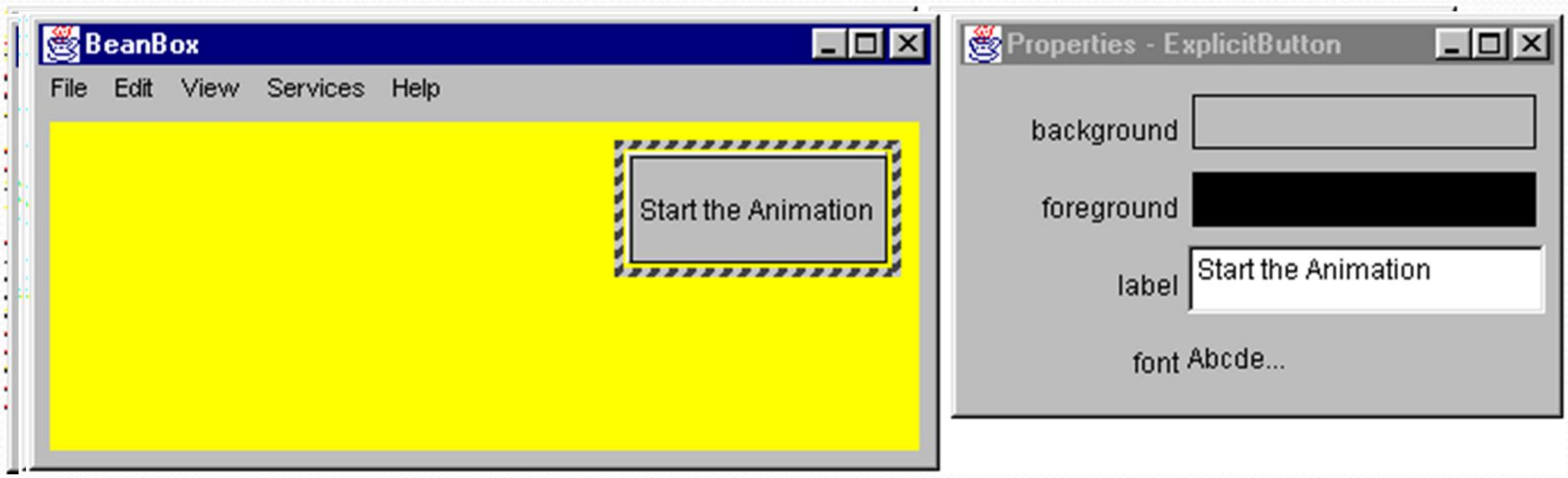
BeanBox Overview

- Now, add JavaBean in **BeanBox** window
  - Click **`ExplicitButton`** bean in **ToolBox** window
    - Functions as a **`JButton`**



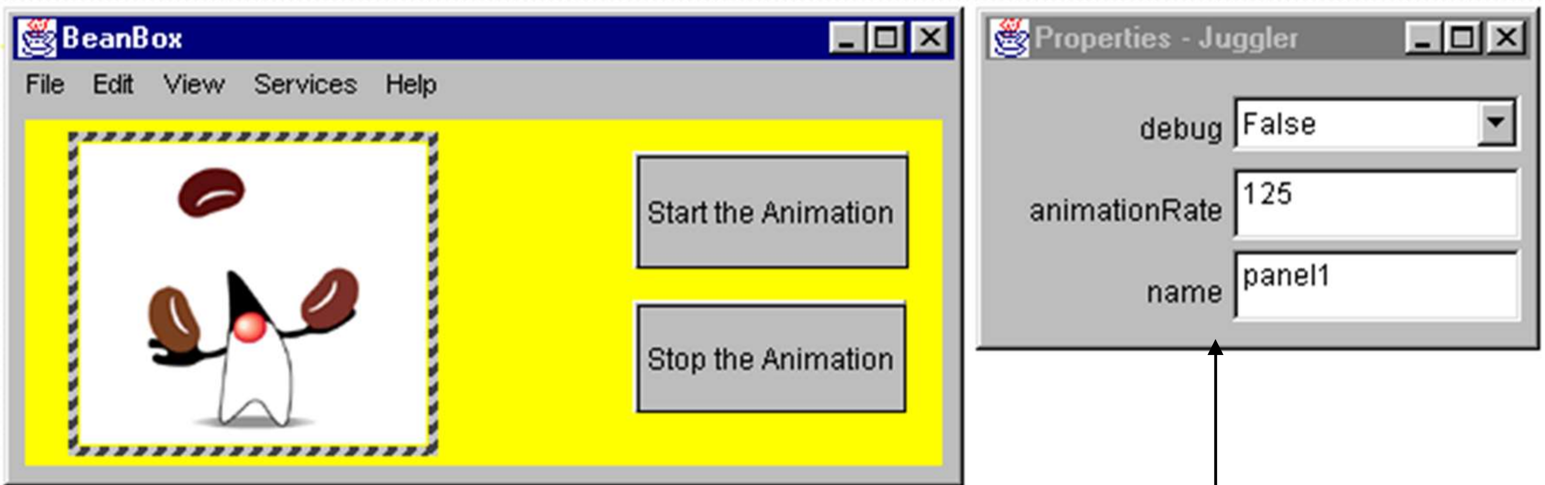
BeanBox Overview

- Select button (if not selected) and move to corner
  - Position mouse on edges, move cursor appears
  - Drag to new location
- Resize button



BeanBox Overview

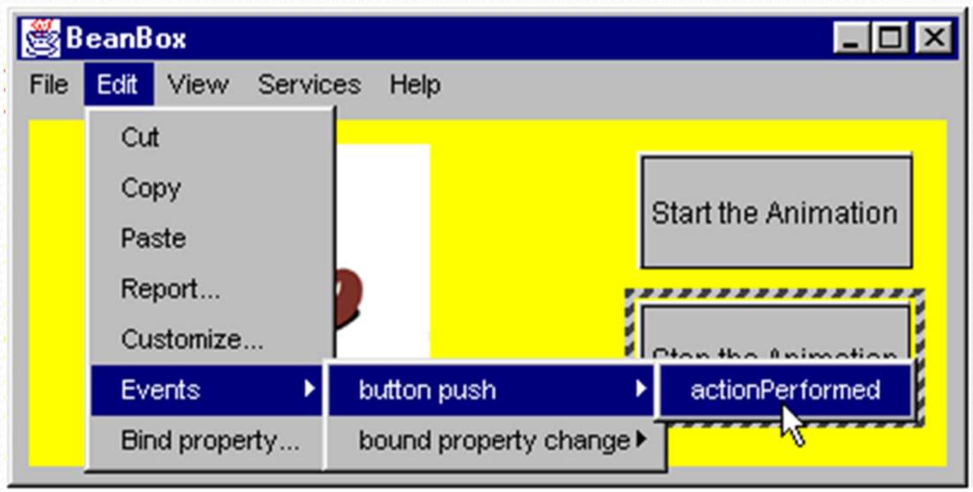
- Add another button (same steps)
  - "Stop the Animation"
- Add animation bean



Properties for juggler.

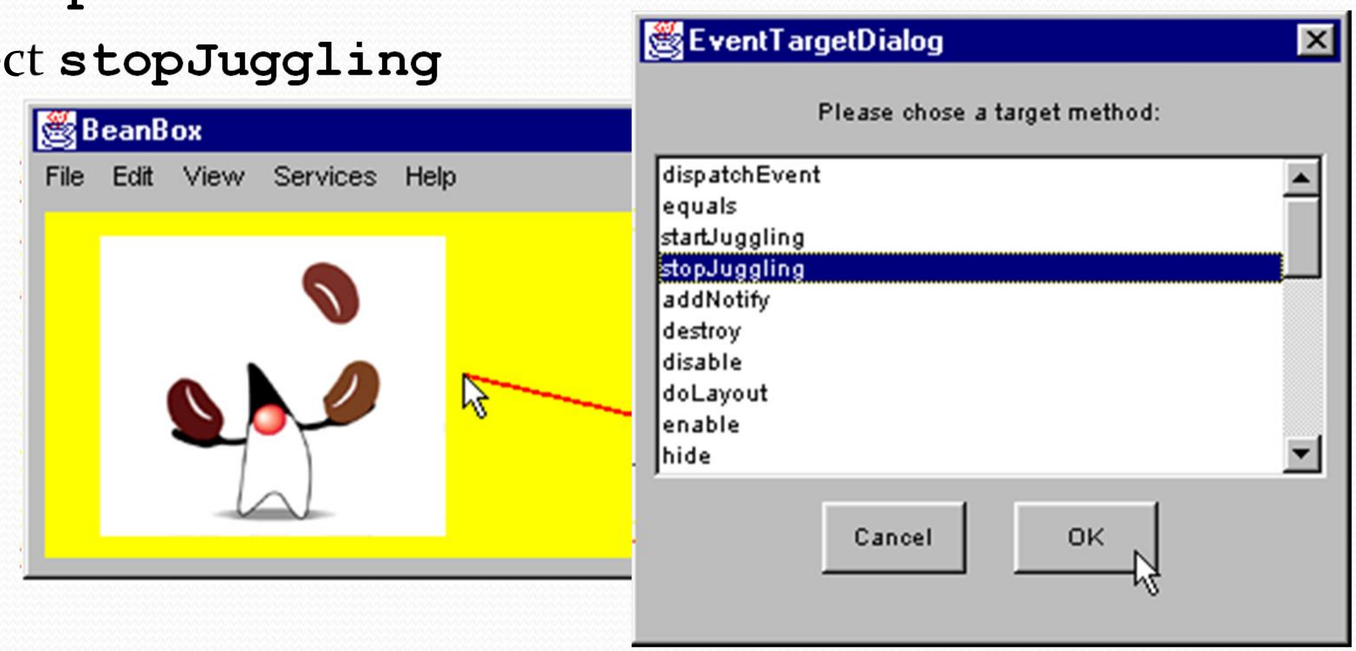
BeanBox Overview

- Now, "hook up" events from buttons
  - Start and stop animation
- **Edit** menu
  - Access to events from beans that are an event source (bean can notify listener)
  - Swing GUI components are beans
  - Select "Stop the Animation"
  - **Edit->Events->button push -> actionPerformed**



BeanBox Overview

- Line appears from button to mouse
  - Target selector - target of event
    - Object with method we intend to call
    - Connect the dots programming
  - Click on **`Juggler`**, brings up **EventTargetDialog**
    - Shows **`public`** methods
    - Select **`stopJuggling`**

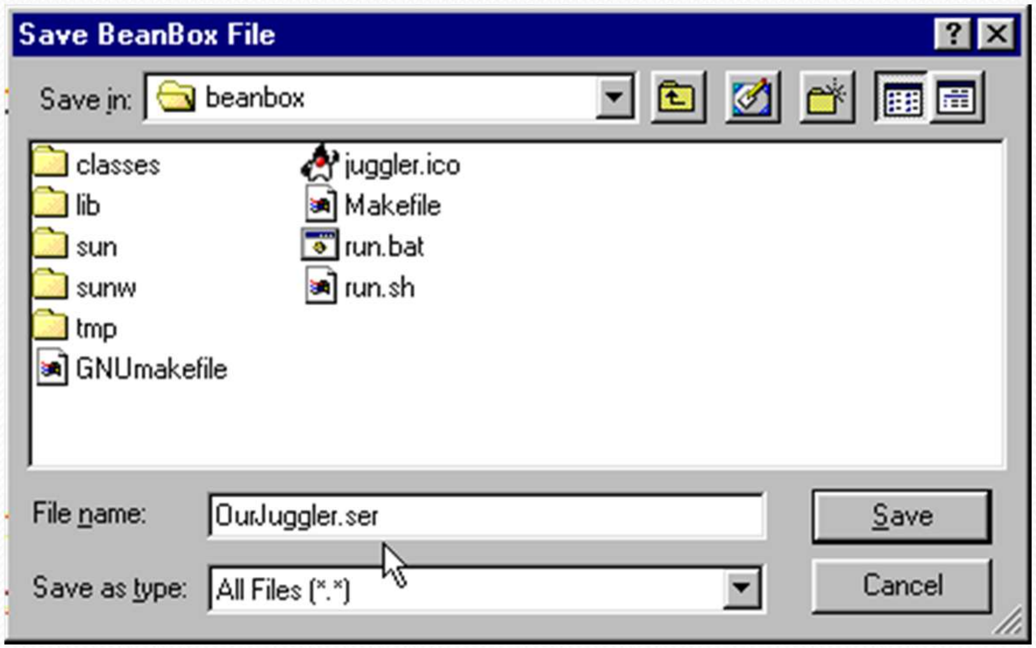


BeanBox Overview

- Event hookup complete
  - Writes new hookup/event adapter class
  - Object of class registered as **`actionListener`** fro button
  - Can click button to stop animation
- Repeat for "Start the Animation" button
  - Method **`startAnimation`**

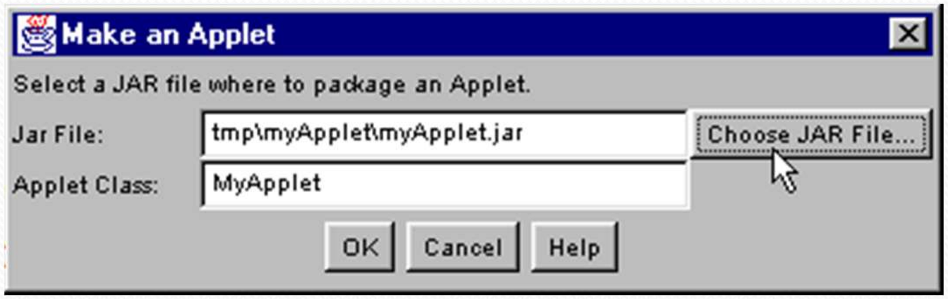
BeanBox Overview

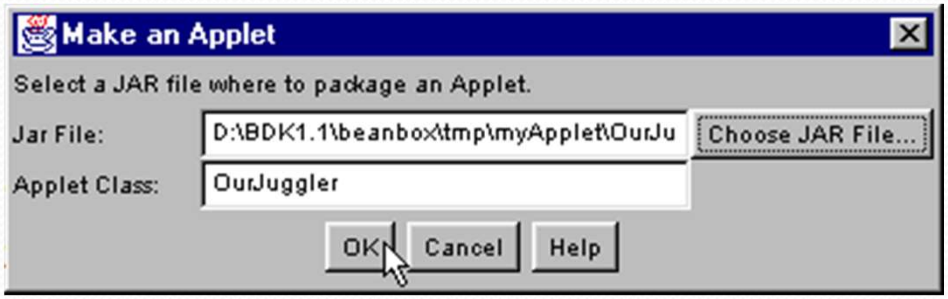
- Save as design
  - Can reloaded into **BeanBox** later
  - Can have any file extension
- Opening
  - Applet beans (like **`Juggler`**) begin executing immediately



BeanBox Overview

- Save as Java Applet
  - **File->Make Applet**
    - Stores `.class` file in `.jar` (Java Archive File)
    - Can rename and change directory





BeanBox Overview

- To run applet
  - Go to command line, go to directory where applet saved
  - Should be **`.html`** file, load into **`appletviewer`**
    - Background not yellow
    - **BeanBox** container not saved as part of applet
    - Applet is a container, can hold beans
  - Archive property of **`<applet>`** tag
    - Comma separated list of **`.jar`** files used
    - **`.jar`** files for beans listed
    - Source code in *AppletName*__**`files`** directory

Creating a JavaBean: Java Archive Files and the `jar` Utility

- Place class in Java Archive file (JAR)
  - Create text file **`manifest.tmp`**
    - Manifest file describes contents of JAR file
    - **`jar`** utility uses **`manifest.tmp`**
      - Creates **`MANIFEST.MF`** in **`META-INF`** directory
      - Used by development environments
  - Can execute application from JAR file with java interpreter
    - Specify class with **`main`**

Creating a JavaBean: Java Archive Files and the `jar` Utility

```
Main-Class: jhtp3beans.LogoAnimator

Name: jhtp3beans/LogoAnimator.class
Java-Bean: True
```

- Manifest file for **`LogoAnimator`**
  - Specify class with **`main`**, runs bean as application
- **`java -jar LogoAnimator.jar`**
  - Run application from bean
  - Interpreter looks at manifest file
    - Executes **`main`** of **`Main-Class`**

`java. -cp LogoAnimator.jar jhtp3beans.LogoAnimator`

- **`cp`** - class path, JAR file to look for classes
- Followed by application class (explicit name, with package)

Creating a JavaBean: Java Archive Files and the `jar` Utility

```
Main-Class: jhtp3beans.LogoAnimator

Name: jhtp3beans/LogoAnimator.class
Java-Bean: True
```

- **`Name:`** name of file with bean class (full package and class name)
  - Dots `.` used in package named replaced with **`/`**
- **`Java-Bean: true`** - file is a JavaBean
  - Possible to have non-JavaBean in JAR file
  - Used to support JavaBeans
- Each class separated by blank line
  - **`Java-Bean:`** immediately follows **`Name:`**

Creating a JavaBean: Java Archive Files and the `jar` Utility

- Create JAR file
  - **`jar`** utility at command line

```
jar cfm LogoAnimator.jar manifest.tmp jhtp3beans\*.*
```

  - Options
    - **`c`** - creating JAR file
    - **`f`** - indicates next argument is name of file
    - **`m`** - next argument **`manifest.tmp`** file
      - Used to create **`MANIFEST.MF`**
  - Next, list files to be included in JAR file
    - Directory structure of JAR file should match **`manifest.tmp`**

Creating a JavaBean: Java Archive Files and the `jar` Utility

- To confirm files were archived
  - **`jar tvf LogoAnimator.jar`**
  - Options
    - **`t`** - list table of contents
    - **`v`** - verbose mode
    - **`f`** - next argument is JAR file to use
  - Execute **`LogoAnimator`** application

    `java -jar LogoAnimator.jar`

Adding Beans to the BeanBox

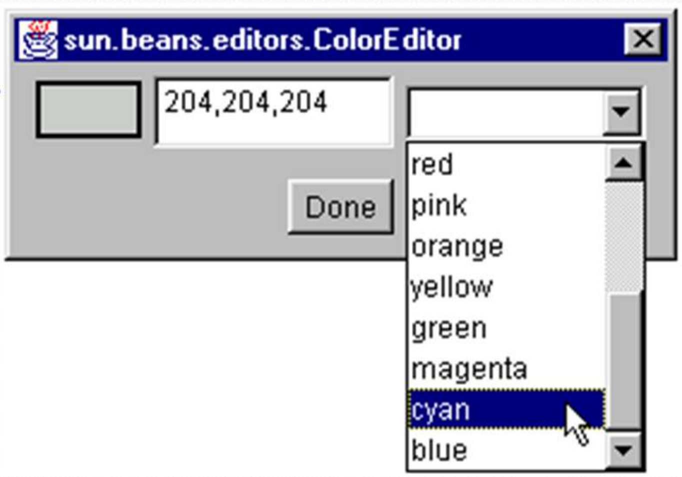
- Using Beans
  - `LogoAnimator` is wrapped in a JAR file as a JavaBean
    - Can use in **BeanBox**
  - Two ways to load bean
    - Put JAR file in `BDK1.1\jars` directory
      - Loaded into toolbox
    - Use **File -> LoadJar**

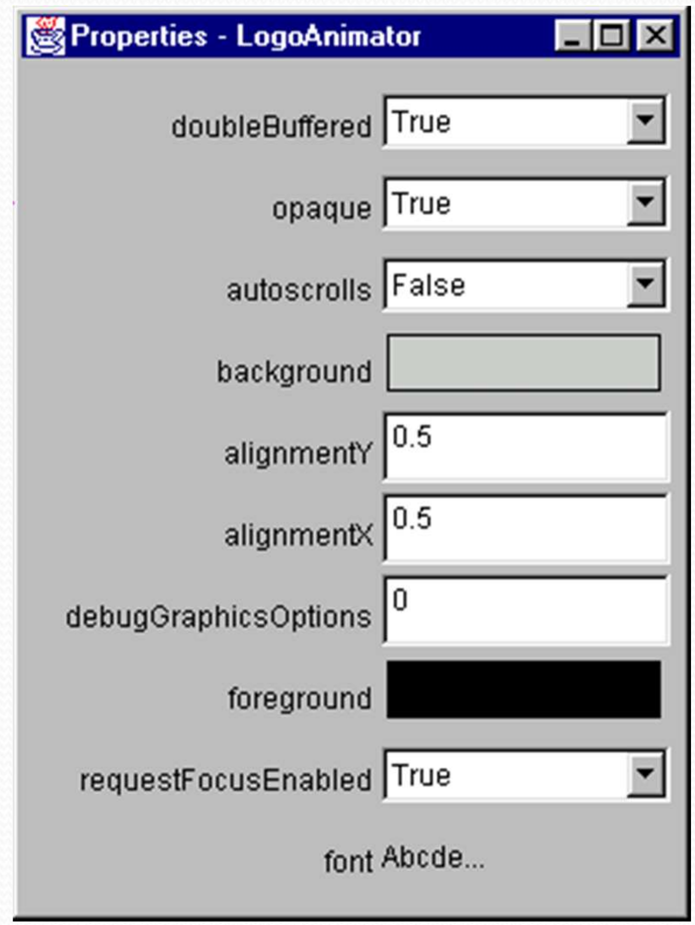
Adding Beans to the BeanBox

- To add to design area
  - Click bean in **ToolBox**
  - Click crosshair where bean should appear

Adding Beans to the BeanBox

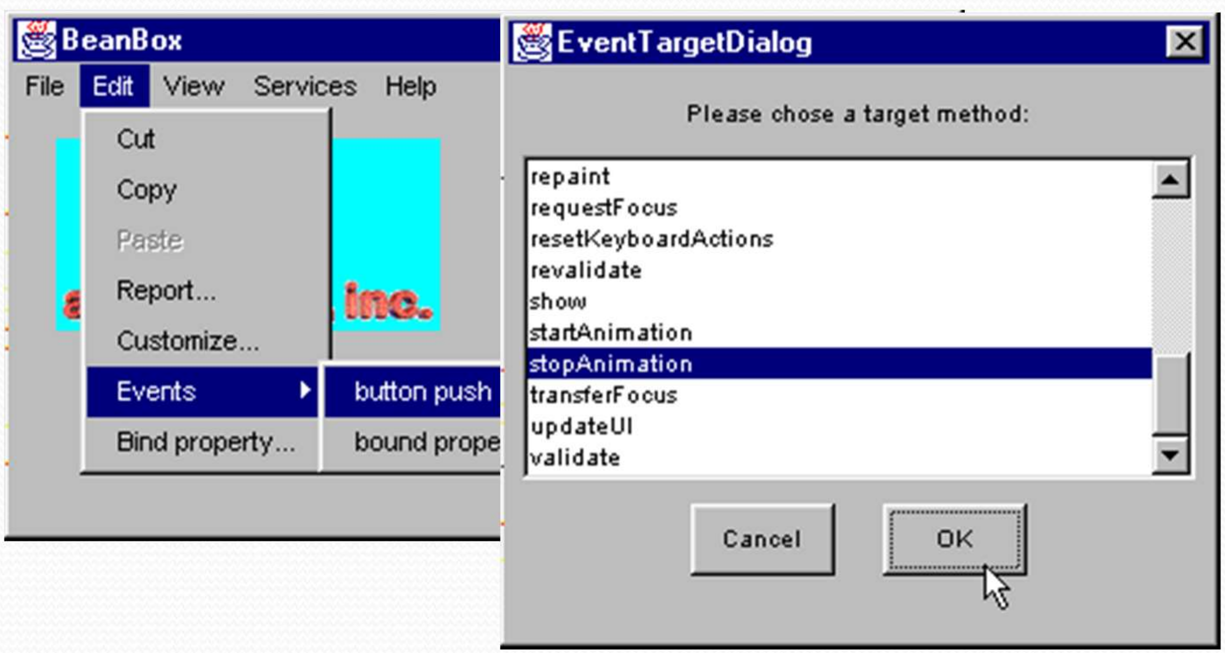
- **Properties** window
  - Shows properties of **`LogoAnimator`**
    - Properties inherited from **`JPanel`**





Connecting Beans with Events in the BeanBox

- Connecting Beans with events
  - **`LogoAnimator`** has methods **`stopAnimation`** and **`startAnimation`**
  - Connect two **`ExplicitButton`**s to **`LogoAnimator`**
    - **`Change label`**
    - **Edit -> Events -> button push -> actionPerformed**



Adding Properties to a JavaBean

- Add **`animationDelay`** property
  - Control animation speed
  - Extend **`LogoAnimator`** and create **`LogoAnimator2`**
  - Read/write property of bean
    - Defined as set/get method pair of format:

`public void set`*PropertyName***`(`** *DataType* **`value )`**

`public` *DataType* **`get`***PropertyName***`()`**

    - Property set and get methods
    - If using boolean, use **`is`***PropertyName***`()`** instead of **`get`**
  - We use **`setAnimationDelay`** and **`getAnimationDelay`**

Adding Properties to a JavaBean

- Properties
  - When builder tool examines bean, looks for pairs of set/get methods
    - Introspection
    - If found, used as property
- Creating bean
  - Must wrap **`LogoAnimator2`** class
  - Compile: **`javac -d . LogoAnimator2.java`**
  - Create **`manifest.tmp`**

```
Main-Class: jhtp3beans.LogoAnimator2

Name: jhtp3beans/LogoAnimator2.class
Java-Bean: True
```

Adding Properties to a JavaBean

- Creating bean
  - Package into JAR file

```
jar cfm LogoAnimator2.jar
manifest.tmp jhtp3beans\*.*
```

  - Load **`LogoAnimator2`** bean, can change **`animationDelay`** property