

# Java



# Java

- Java is object oriented.
- Java works on most platforms. While C/C++ programs are platform specific.
- Java is network enabled. It is trivially simple to write code in Java that works across networks.

# What is JVM?

- It is a hypothetical processor which executes java programs.
- The java compilers produces binary byte code designed to execute on JVM rather than on a PC or Sun Workstation.
- It is abbreviated as Java Virtual Machine.

# Declaring Variables

- Java requires that you *declare* every variable before you can use it.
- You can declare the variables in several ways. Often, you declare several at the same time.

```
int y, m, x //all at once
```

or one at a time.

```
int y; //one at a time
```

```
int m;
```

```
int x;
```

# Declaring Variables (cont...)

- You can also declare the variables as you use them.

```
int x = 4;  
int m = 8;  
int b = -2;
```

- However, you **MUST** declare variables by the time you first refer to them.

# Constants

- Constants can be defined by using the **final** modifier.
- It is a good practice to CAPITALIZE symbols referring to constants.

- Examples

```
final float PI = 3.1416;
```

```
final int NUMBER_OF_DAYS_IN_MONTH = 30;
```

# Data Types

Type	Contents
boolean	true or false
byte	Signed 8-bit value
short	16-bit integer
int	32-bit integer
long	64-bit integer
float	32-bit floating point
double	64-bit floating point
char	16-bit unicode character

# Data Type Conversions

- Any “wider” data type can have a lower data type assigned directly to it and the promotion to the new type will occur automatically.
- For example, If  $y$  is of type **float** and  $j$  is of type **int** then you can write:
  - `float y; //y is of type float`
  - `int j; //j is of type int`
  - `y = j; //convert int to float`to promote an integer to a float.



# Data Type Conversions (cont...)

- You can reduce a wider type to a narrower type using by *casting* it. You do this by putting the data type name in parentheses and putting this name in front of the value you wish to convert:

- For example,

```
j = (int)y; //convert float to integer
```

# Boolean Data Type

- Boolean variables can only take on the values represented by the reserved words **true** and **false**.



Unlike C, you cannot assign numeric values to a boolean variable and you cannot convert between **boolean** and any other type.

# Numeric Data Types

- Any number you type into your program is automatically of type **int** if it has no fractional part or of type **double** if it does.
- For example in a program if a number, say 5.5 is used then it will be of double data type by default.

# Numeric Data Types (cont...)

- If you want to indicate that it is a different type, you can use various suffix and prefix characters to indicate what you had in mind.

- For example,

```
float loan = 1.23f; //float
```

```
long pig = 45L; //long
```

```
long color = 0x12345; //hexadecimal
```

# Simple Java Program

```
import java.io.*  
public class Add2 {  
  
    public static void main(String argv[ ]) {  
        double a, b, c;  
        a = 1.75;  
        b = 3.46;  
        c = a + b;  
  
        // print out sum  
        System.out.println("sum = " + c);  
    }  
}
```

**Output : sum = 5.21**

# import statement

- You must use the **import** statement to define libraries or *packages* of Java code that you want to use in your program
- This is similar to the C and C++ **#include** directive.

# “main” Method

- The program starts from a function called **main** and it must have *exactly* the form shown here:

```
public static void main(String argv[])
```

**or**

```
public static void main(String[] argv)
```

# Comments

- Single line comments start with “//” (double forward slash)
- Multiple lines can be commented by enclosing the required code block between /\* and \*/



# Class Definition

- Every program module must contain one or more classes.
- The class and each function within the class is surrounded by *braces* ( { } ).
- Like most other languages the equals sign is used to represent assignment of data.

# String concatenation

- You can use the “+” sign to combine two strings. The string “sum =” is concatenated with the string representation of the double precision variable `c`.

# Arithmetic Operators

+	Addition
-	Subtraction, Unary Minus
*	Multiplication
/	Division
%	Modulus

# Assignment Operators

=	Assignment
+=	Compound assignment
-=	
*=	
/=	
etc	

# Increment & Decrement Operators

- Java allows you to express incrementing and decrementing of integer variables using the “++” and “--” operators.

- For example

```
i = 5; j = 10;
```

```
x= i++ //x = 5, then i = 6
```

```
y = --j; //y = 9 , j = 9;
```

# Control structures



# Making decisions in Java

- The familiar if-then-else of Visual Basic has its analog in Java. Note that in Java, however, we do not use the “then” keyword. For Example,

```
if ( y > 0 )  
    z = x / y;
```

- Parentheses around the condition are **REQUIRED** in Java.

# Making decisions in java (cont...)

- If you want to have several statements as part of the condition, you must enclose them in braces:

```
if (y > 0) {  
    z = x / y;  
    System.out.println("z = " + z);  
}
```



# Comparison Operators

Java	Meaning
>	Greater than
<	Less than
==	Is equal to
!=	Is not equal to
>=	Greater than or equal to
<=	Less than or equal to
!	Not
<b>Note</b> :- All of these operators return boolean results	

# Logical operators

Operator	Meaning
&	Logical AND
	Logical OR
&&	Shortcut Logical AND
	Shortcut Logical OR

# switch statement

```
switch(expression) {  
    case constant : statements  
    case constant : statements  
    ----  
    ----  
    default : statements  
}
```

# switch statement - example

```
switch(number) {  
    case 1: System.out.println("One");  
            break;  
    case 2: System.out.println("Two");  
            break;  
    case 3: System.out.println("Three");  
            break;  
    default : System.out.println("Invalid");  
}
```

# Loops

<pre>while(boolean-expression)     statement</pre>
<pre>do     statement     while(boolean-expression);</pre>
<pre>for(initialization; boolean-expression; step)     statement</pre>

# break and continue

- You can also control the flow of the loop inside the body of any of the iteration statements by using **break** and **continue**
- **break** quits the loop without executing the rest of the statements in the loop
- **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration
- For nested loops, labels can be used along with these statements to specify the loop

# Classes and Objects



# Class

- It is a basic unit in java programming language.
- It provides a structure for objects.
- Contract – A combination of methods, data and semantics.



# Java Class Structure

```
package <package_name>;  
  
import <other_packages>;  
  
public class ClassName {  
    <variables(also known as fields)>;  
  
    <constructor(s)>;  
  
    <other methods>;  
}
```

# Simple class

```
class BankAccount {  
    private double balance = 0.0;  
}
```

# Class Members

1. Fields
2. Methods
3. Classes – Nested classes
4. Interfaces – Nested interfaces

# Fields

- Class variables are called fields.
- Fields are data variables associated with a class and its objects.
- Instance variables – associated with objects
- Static variables – associated with class

# Field Initialization

- When a field is declared it can be initialized by assigning it a value of the corresponding type
  - `double zero = 0.0; // constant`
  - `double sum = 4.5 + 3.7; // constant expression`
  - `double zeroCopy = zero; // field`
  - `double rootTwo = Math.sqrt(2); // method invocation`
  - `double someVal = sum + 2 * Math.sqrt(rootTwo)`

# Field Initialization *contd..*

- If a field is not initialized a default initial value is assigned to it depending on its type.

Type	Default value
boolean	false
char	'\u0000'
int types	0
float	0.0f
double	0.0
Object ref	null

# Final fields

- A final variable is one whose value cannot be changed after it has been initialized.
- Ex:

```
final double PI = 3.141592;
```

# Methods

- Methods also are members of a class
- Method should have a type
- Type of a method is the type of data it returns
- If the method does not return a value its type is void
- 'this' can be used to refer instance variable explicitly

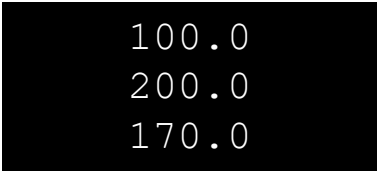


# Method Overloading

- **Rules for method overloading :**
  1. Overloaded methods must change the argument list.
  2. Overloaded methods can change the return type.
  3. Overloaded methods can change the access modifier.

# Method Overloading

```
public static void main(String[] args) {  
    Sales s = new Sales();  
  
    System.out.println(s.computeSales(100));  
    System.out.println(s.computeSales(100,2));  
    System.out.println(s.computeSales(100,2,30));  
}  
  
class Sales {  
    double computeSales(double price) {  
        double sales;  
        sales = price;  
        return sales;  
    }  
    double computeSales(double price, int qty) {  
        double sales;  
        sales = price * qty;  
        return sales;  
    }  
    double computeSales(double price, int qty, double discount) {  
        double sales;  
        sales = (price * qty) - discount;  
        return sales;  
    }  
}
```



100.0  
200.0  
170.0

# Static members

- Variables shared by all objects of a class are called static fields or class variables.

```
static int nextID;
```

- But, when accessed externally it must be accessed via class name.

Ex: System.out

- Methods also can be static

# Access control

- It provides a way to who has access to what members of a class.
  - **private** – accessible only in the class itself.
  - **package** – accessible in
    - classes in the same package
    - class itself
  - **protected** – accessible in
    - subclasses of any package
    - classes in same package,
    - the class itself.
  - **public** – accessible
    - anywhere the class is accessible.

# Creating Objects

- Object of a class is created using the keyword – new

- Ex:-

```
BankAccount anAccount = new BankAccount()  
anAccount.balance = 1000.00;
```

- You never delete objects. JVM manages memory for you using *garbage collection*.

# Constructors

- Constructors are blocks of statements that can be used to initialize an object.
- Constructors have the same name as the class they initialize.
- Constructors take zero or more arguments.

# Constructors (contd)

EX:

```
class BankAccount {  
    double balance = 0.0  
    BankAccount(double initialBalance) {  
        balance = initialBalance  
    }  
}
```

**Using constructor to create objects:**

```
BankAccount anAccount = new BankAccount(1000.00);
```

# Constructors (contd)

- All Java classes have constructors that are used to initialize a new object of that type
- A constructor has the **SAME NAME** as the class
- A constructor **DOESNOT** have return type.
- For example, a no argument constructor for `Stack` class can be

```
public Stack() {  
    items = new Vector(10);  
}
```



# Constructors -Overloading

- Java supports name overloading for constructors so that a class can have any number of constructors, all of which have the same name
- For example, constructors for the stack classes can be
  - `public Stack()` // no argument ctor
  - `public Stack(int initialSize)` // 1 argument ctor

# Default Constructor

- When writing your own class, you **DON'T HAVE** to provide constructors for it
- The default no argument constructor is automatically provided by the runtime system for any class that contains no constructors
- If a constructor with arguments is provided, default constructor is not automatically created by runtime system

# Create Object

- A class provides the blueprint for objects
- Variable of class type is object reference
- Unless assigned with object reference, variable value is null

```
Point p1 = new Point(23, 94);  
Rectangle r1 = new Rectangle(origin_one, 100, 200);  
Point p2;    // value of p2 is null  
P1 = new Point(23,34);  // now p1 refers a new object
```

- **Referencing an Object's Variables (instance variables)**  
objectReference.variableName
- **Calling an Object's Methods (instance methods)**  
objectReference.methodName(argumentList);

# Life Cycle Of an Object

## **Cleaning Up Unused Objects**

- The Java runtime environment deletes objects when it determines that they are no longer used. This process is called *garbage collection*.
- An object is eligible for garbage collection when there are no more references to that object.
- We can explicitly drop an object reference by setting the variable to the special value *null*

## **Garbage Collector**

- JRE has a garbage collector that periodically frees the memory used by objects that are no longer referenced.

# Encapsulation

- Encapsulation is one of the four fundamental object-oriented programming concepts.
- The term *encapsulation* means to enclose in a capsule, or to wrap something around an object to cover it.
- Encapsulation covers, or wraps, the internal workings of a Java object.
  - Data variables, or fields, are hidden from the user of the object.
  - Methods, the functions in Java, provide an explicit service to the user of the object but hide the implementation.
  - As long as the services do not change, the implementation can be modified without impacting the user.

# Public and Private Access Modifiers

- The `public` keyword, applied to fields and methods, allows any class in any package to access the field or method.
- The `private` keyword, applied to fields and methods, allows access only to other methods within the class itself.
- One way to hide implementation details is to declare all of the fields `private` and methods as `public`

# String objects



# String

- The String class represents character strings
- All string literals in Java programs, such as "abc", are implemented as instances of this class.
- Strings are constant; their values cannot be changed after they are created
- The class String includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase
- The Java language provides special support for the string concatenation operator ( + ), and for conversion of other objects to strings.



# String (contd)

- Ways of creating String objects :

```
String s = "Hello"
```

```
String s = new String("hello");
```

```
char[ ] ch = { 'a','b','c'};
```

```
String s = new String(ch);
```

# String methods

## Some of the methods

char charAt(int index)

String concat(String)

boolean equals(Object)

boolean equalsIgnoreCase(String)

int indexOf(String str)

int length()

String replace(char old, char new)

String substring(int begin, int end) // begin to end – 1

String toLowerCase()

String toUpperCase()

String trim()

static String valueOf(alltypes)

# StringBuffer & StringBuilder

Both classes represent mutable string objects

Both have same methods

StringBuffer is threadsafe, StringBuilder is not

Constructors

`StringBuilder()` // initial capacity 16

`StringBuilder(int capacity)`

`StringBuilder(String st)` //create with capacity 16 + length of st

# StringBuffer & StringBuilder

## Some methods

StringBuilder append(alltypes)

int capacity( )

char charAt(int index)

int capacity()

StringBuilder delete(int start, int end) //start to end – 1

StringBuilder deleteCharAt(int index)

int indexOf(String)

StringBuilder insert(int offset, alltypes)

int length( )

StringBuilder replace(int start, int end, String new)

StringBuffer reverse( )

# Arrays

- Arrays provide ordered collections of elements.
- Components of array can be primitive types or references to objects, including references to other arrays.
- Arrays themselves are objects and extend the class `Object`
- Examples:  

```
int[] x = new int[3];  
int y[] = new int[3];
```

# Arrays (cont...)

- An `ArrayIndexOutOfBoundsException` is thrown if the index is out of bounds.
- The index expression must be of type `int`
- Implicit length variable used to know the size of the array
- An array with length zero is said to be an empty array.

# Arrays - example

```
public class ArrayTest {  
    public static void main(String[] args) {  
        int a1[] = {10,34,56,23,67,87};  
        int a2[]; // value is null  
        int a3[] = new int[5];  
        a2 = a1; // a1 and a2 hold same array  
        /* use length to know the size of the array */  
        for(int i=0;i<a1.length;i++)  
            System.out.println(a1[i]);  
    }  
}
```

# Array of objects

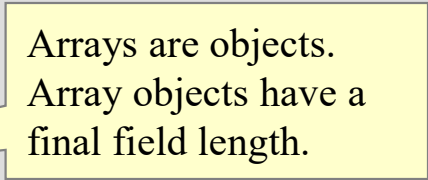
- Array of objects is array of references to the objects as shown in this example

```
public class ArrayOfStringsDemo {  
    public static void main(String[] args) {  
        Test b[] = new Test[5];  
        Test a[] = { new Test("Ramana"), new Test("Surender"),  
                     new Test("Hires"), new Test("Haritha") };  
        for (int i = 0; i < a.length; i++) a[i].show();  
    }  
}  
  
public class Test {  
    private String name;  
    public Test(String s) {  
        name = s;  
    }  
    public void show() {  
        System.out.println(name);  
    }  
}
```



# Arrays and for-each Loop

```
1 public class ArrayOperations {
2     public static void main(String args[]){
3
4         String[] names = new String[3];
5
6         names[0] = "Blue Shirt";
7         names[1] = "Red Shirt";
8         names[2] = "Black Shirt";
9
10        int[] numbers = {100, 200, 300};
11
12        for (String name:names){
13            System.out.println("Name: " + name);
14        }
15
16        for (int number:numbers){
17            System.out.println("Number: " + number);
18        }
19    }
20 }
```



Arrays are objects.  
Array objects have a  
final field length.

# Multi dimensional arrays

- Multidimensional arrays are arrays of references to arrays

```
String[][] cartoons = {  
    { "Flintstones", "Fred", "Wilma", "Pebbles", "Dino" },  
    { "Rubbles", "Barney", "Betty", "Bam Bam" },  
    { "Jetsons", "George", "Jane", "Elroy", "Judy", "Rosie",  
      "Astro" },  
    { "Scooby Doo Gang", "Scooby Doo", "Shaggy", "Velma", "Fred",  
      "Daphne" }  
};
```

```
int x[][] = new int[5][10];
```

```
int y[][] = new int[3][];  
y[0] = new int[5];  
y[1] = new int[10];  
y[2] = new int[2];
```

# Annotations



# Annotations

- Annotations provide data about a program
- An **annotation** is an attribute of a program element.
- As of release 5.0, the platform has a general purpose annotation (metadata) facility that permits to define and use annotation types.
- The facility consists of:
  - a syntax for declaring annotation types
  - a syntax for annotating declarations
  - APIs for reading annotations
  - a class file representation for annotations
  - an annotation processing tool

# Annotation Usage

- Annotations have a number of uses, among them:
  - **Information for the compiler** - Annotations can be used by the compiler to detect errors or suppress warnings
  - **Compiler-time and deployment-time processing** - Software tools can process annotation information to generate code, XML files, and so forth
  - **Runtime processing** - Some annotations are available to be examined at runtime (reflection)

# What can be annotated?

Annotatable program elements:

- package
- class, including
  - interface
  - enum
- method
- field
- only at compile time
  - local variable
  - formal parameter

# Annotations Used by the Compiler

- There are three annotation types that are predefined by the language specification itself:
  - **@Deprecated** – indicates that the marked element is deprecated and should no longer be used
  - **@Override** – informs the compiler that the element is meant to override an element declared in a superclass
  - **@SuppressWarnings** – tells the compiler to suppress specific warnings that it would otherwise generate

# Inheritance





# INHERITANCE

- Inheritance denotes Specialization.
- A *subclass* is a class that extends another class. A subclass inherits state and behavior from all of its ancestors (a.k.a super class(es)).
- A subclass inherits variables and methods from its superclass and all of its ancestors. The subclass can use these members as is, or it **can hide** the member variables or **override** the methods.

# Constructors in inheritance

- While creating subclass objects super class default constructor is automatically invoked
- To invoke argumented constructor of super class use `super()` with arguments
- `super()` should be first statement in subclass constructor

# Overriding

- Overriding a method means replacing the superclass's implementation of a method with one of your own. The signature must be identical
- When a method is overridden it means both the signature and return type are **SAME** as in the superclass.
- If two methods differ only in return type it is an ERROR and the compiler will reject the class.

# Overriding (cont...)

- Overriding methods can have their own access specifiers. A subclass can change the access of a superclass's methods, but **ONLY** to provide **MORE** access.

- For example

```
class Base {  
    Protected void show() {  
    }  
}  
Class Derived extends Base{  
    Public void show() { //this is valid  
    }  
}
```

# Overriding (cont...)

- The Overriding method can be made final but not the method being overridden.
- Overriding method's throws clause CAN BE different from that of the superclass method's as long as every exception type listed in the overriding method is the same or a subtype of the exceptions listed in the superclass's method.
- An overriding method can have **NO** throws clause though the method in superclass has.
- Static method **CANNOT** be overridden.

# Polymorphism

- Super class reference variable can hold sub class object
- When a overridden method is invoked on super class reference variables, the method of the object held by it is invoked

# Polymorphism

Example :

```
class Person {  
    public void display(){ system.out.println("Person"); }  
}
```

```
class Employee extends Person {  
    public void display(){ system.out.println("Employee Data"); }  
}
```

```
class Student extends Person {  
    public void display(){ system.out.println("Student Data"); }  
}
```

# Polymorphism

Example :

```
Person p ;
```

```
p = new Person();
```

```
p.display();           // output : Person
```

```
p = new employee();
```

```
p.display();           // output : Employee Data
```

```
p = new Student();
```

```
p.display();           // output : Student Data
```



# Abstract methods & classes

- Methods whose design is not complete are called abstract methods
- Ex: `public abstract void printIt(int x, String y);`
- Class having abstract methods should be declared as abstract class
- Abstract class cannot be instantiated (cannot create objects)
- Abstract classes are for subclassing
- A class declared as abstract need not have abstract methods

# final class

- Class declared as final cannot be extended
- final class cannot have abstract methods
- Ex:

```
public final class LastOne {
```

```
-----
```

```
-----
```

```
}
```

# Interfaces



# Introduction

- The fundamental unit of OO design is the *type*.
- *Interfaces* define types in an **abstract** form as a collection of methods.
- *Interfaces* contain **no implementation** and you cannot create instances of an interface.
- Classes can expand their own types by implementing interfaces.

# Introduction (contd...)

- Classes can implement more than one interface.
- In a given class, the classes that are extended and the interfaces that are implemented are collectively called the *supertypes*, the new class is a *subtype*.

# Interface example

- An example of a simple interface :

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

# Declarations

- An *interface* is declared using the keyword **interface**, giving the interface a name and listing the interface members between braces.
- An interface can have
  1. Constants
  2. Methods (only signature)
- Interface members are implicitly *public*.

# Interface constants

- An interface can declare named constants.
- These constants are *implicitly* *public*, *static*, and *final*.
- *interface* Verbose {  
    int SILENT = 0;  
    int NORMAL = 1;  
    int VERBOSE = 3;  
}



# Interface methods

- The methods declared in an interface are implicitly **abstract** and **public**, no other modifiers are permitted
- Methods **cannot be** *static* – because static methods cannot be abstract.

# Extending interfaces

- Interfaces can be extended using the extends keyword.

```
public interface SerializableRunnable  
    extends Serializable, Runnable {  
    //.....  
}
```

- The interfaces that are extended are the *superinterfaces* and the new interface is a *subinterface*

# Implementing interfaces

- A class can implement one or more interfaces using *implements* keyword
- ```
public class XXXX implements  
Comparable {  
    public int compareTo(Object o )  
        // Implementation details  
}
```

# Marker interfaces

- Some interfaces do not declare any methods. These are marker interfaces.
- Marker interfaces simply mark a class as having some general property.
- Examples of marker interfaces:
  - `Serializable`
  - `Cloneable`
  - `java.rmi.Remote`

# Packages

- Packages are convenient ways of grouping related classes according to their functionality, usability as well as category they should belong to.
- Classes under different packages **CAN HAVE** same names.
- Packaging help us to avoid class name collision when we use the same class name as that of others

# How to create Packages?

1. Suppose we have a file called **HelloWorld.java**, and we want to put this file in a package **world** then add the package definition in the top of the file as shown below...

```
// only comment are allowed before this definition  
package world;  
public class HelloWorld {  
    //...  
}
```

2. Create subdirectories to represent package hierarchy of the class. In our case, we have the **world** package, which requires only one directory. So, we create a directory **world** and put our HelloWorld.java into it.

# How to use Packages?

- There are 2 ways in order to use the public classes stored in package.
1. Declare the fully-qualified class name. For example,

```
world.HelloWorld hw = new world.HelloWorld();
world.moon.HelloMoon hm = new world.moon.HelloMoon();
String holeName = helloMoon.getHoleName();
```
  2. Use an "import" keyword:

```
import world.*;
import world.moon.*;
HelloWorld helloWorld = new HelloWorld();
HelloMoon helloMoon = new HelloMoon();
```

# CLASSPATH

- Environment variable CLASSPATH should be set to search for packages
- Specify a series of folders in classpath
- Ex:  
**Set CLASSPATH=c:\;d:\javaprg;c:\test\classes**
- Packages / classes are searched for in c:\ , d:\javaprg and c:\test\classes in that order
- Classes with no package statement belong to default package



# Using Access Control

- There are four access levels that can be applied to data fields and methods. The following table illustrates access to a field or a method marked with the access modifier in the left column.

| Modifier<br>(keyword)  | Same Class | Same<br>Package | Subclass in<br>Another<br>Package | Universe |
|------------------------|------------|-----------------|-----------------------------------|----------|
| <code>private</code>   | Yes        |                 |                                   |          |
| <code>default</code>   | Yes        | Yes             |                                   |          |
| <code>protected</code> | Yes        | Yes             | Yes *                             |          |
| <code>public</code>    | Yes        | Yes             | Yes                               | Yes      |

# Points to Note

- Classpath not required for JDK classes
- While using JDK classes, package to which the class belongs should be identified for import statement
- java.lang package classes need not be imported
- Object, String, StringBuffer, Runtime, System classes and wrapper classes belong to java.lang package

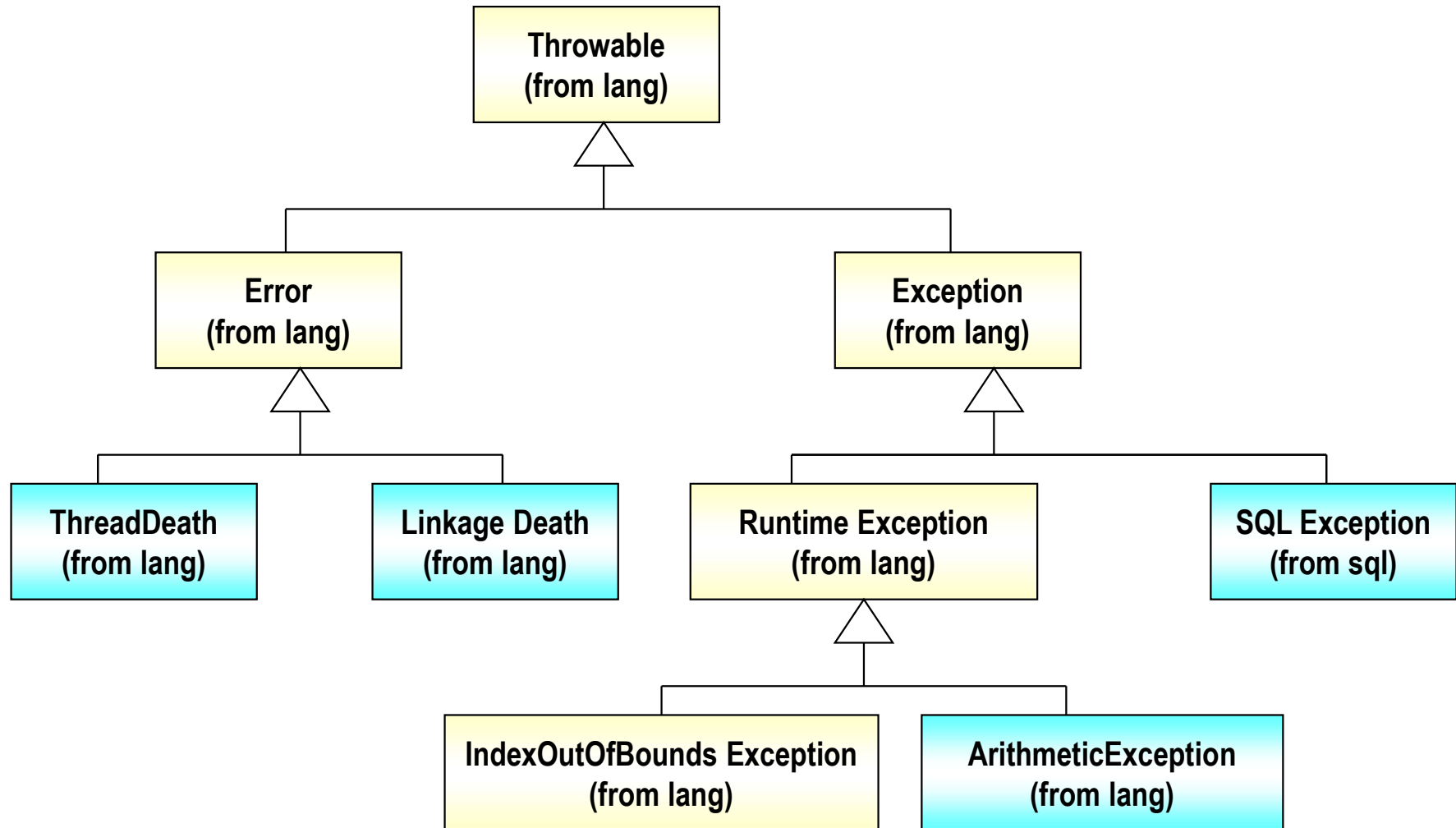
# Exceptions



# Exception Handling

- Exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class files you are interested in loading are missing

# Exception class hierarchy



# Exception Handling

- Checked Exceptions
  - Extends the `java.lang.Exception` class.
  - Needs to be caught or specified.
- Unchecked Exceptions
  - Extends the `java.lang.RuntimeException` class
  - Need not be caught or specified.

# Exception Handling

- Methods should either catch or specify all checked exceptions that can be thrown within the scope of that method
- A method can catch an exception by providing an exception handler for that type of exception
- If a method chooses not to catch an exception, the method must specify that it can throw that exception
- Callers of a method must know about the exceptions that a method can throw

# Dealing with Exceptions

- Three components of an exception handler

**try, catch, and finally blocks**

## **try Block**

- Enclose the statements that might throw an exception within a try block
- Defines the scope of any exception handlers

## **catch Block**

- Associate exception handlers with a try block by providing one or more catch blocks directly after the try block

## **finally Block**

- Allows the method to clean up after itself regardless of what happens within the try block



# Exceptions - example

```
public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(
            new FileWriter("OutFile.txt"));

        for (int i = 0; i < size; i++)
            out.println("Value at: " + i + " = " +
                victor.elementAt(i));
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Array refererror " + e.getMessage());
    } catch (IOException e) {
        System.out.println("IOException: " + e.getMessage());
    }
    finally {
        if (out != null)
            out.close();
    }
}
```

# Specifying the Exceptions Thrown by a Method

- To let a method transfer the exception to the caller, specify throws
- Caller is responsible to handle this exception

Ex:

```
public void writeList() throws IOException {  
    PrintWriter pw =  
        new PrintWriter(new FileWriter("File.txt"));  
    for (int i = 0; i < size; i++)  
        pw.println(b[i]);  
    pw.close();  
}
```

# Defining Exceptions

## **throw statement**

**Before you can catch an exception, some Java code somewhere must throw one**

```
throw someThrowableObject;
```

**Throwable objects should be subclass of the Throwable class**

# Defining Exceptions - example

```
public class ApplicationException extends Exception{
    public ApplicationException(String msg) {
        super(msg);
    }
}

public void readFile(String fileName) throws
    ApplicationException {
    try {
        FileInputStream fis = new FileInputStream(fileName);
        .....
    }
    catch (FileNotFoundException e) {
        throw new ApplicationException("File Not Found",
            e);
    }
}
```

# Classes of java.lang package



# Object class

- Every java class extends Object class either directly or indirectly
- Object class provides useful methods required in every class
- Some of the methods need to be overwritten
- If a class does not extend any class, it automatically extends Object
- Every java object is instanceof Object

# Object class

## Some Object class methods

### `boolean equals (Object obj)`

Decides whether two objects are meaningfully equivalent.

### `void finalize( )`

Called by garbage collector when the garbage collector sees that the object cannot be referenced

### `int hashCode( )`

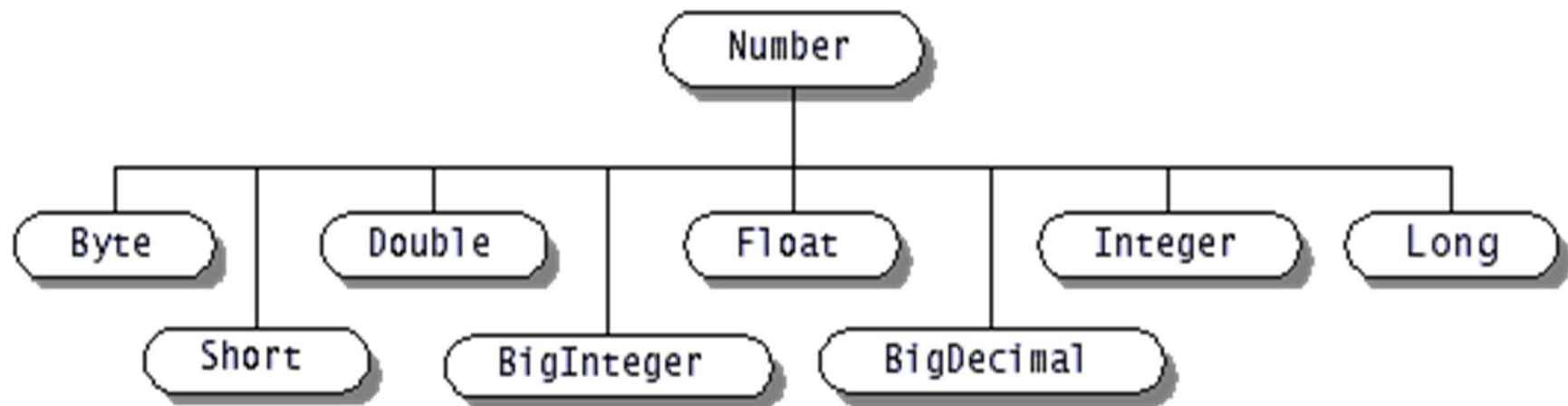
Returns a hashcode int value for an object, so that the object can be used in Collection classes that use hashing, including Hashtable, HashMap, and HashSet

### `String toString( )`

Returns a “text representation” of the object

# Wrapper classes

- Wrapper classes available to create objects for all primitive types
- These classes provide methods to convert wrap primitive types and also convert back to primitives



- Also available, Boolean and Character classes



# Wrapper classes

- Designed to convert primitives into objects
- Wrapper objects are immutable
- provide methods for conversion of primitives to/from String objects to different bases
- All wrapper class names map to primitives they represent except Integer and Character
- Byte, Short, Integer, Long, Float, Double are sub classes of Number
- constructors overloaded to take primitives as well as their String representation

# Wrapper classes

- Common methods of wrapper classes

## Methods of Number

byte    byteValue( )  
short   shortValue( )  
int     intValue( )  
long    longValue( )  
float   floatValue( )  
double doubleValue( )

## Character

char charValue( )

## Boolean

boolean booleanValue( )

# Auto Boxing Unboxing

- ☞ Before Java 5, wrapping and unwrapping was done explicitly
- ☞ For example to perform arithmetic on a wrapped value involved unwrapping and re-wrapping after operation

```
Integer x = new Integer(45);
```

```
int y = x.intValue( );
```

```
y = y + 10 ;
```

```
x = new Integer(y) ;
```

- ☞ java 5 provides auto boxing / unboxing
- ☞ In that wrapping and unwrapping done automatically based on the operation

```
Integer x = 45;
```

```
x = x + 10 ;
```

```
x++; // and so on
```

# Auto Boxing Unboxing

All these are possible now !!!

```
Integer x = 36; // wrap it  
x++;           // unwrap and re-wrap
```

```
List l = new ArrayList();  
l.add(0,36); //wrap and add
```

```
Integer a = 30; //wrap  
Integer b = 20; //wrap  
Integer c = a + b; //unwrap, add, wrap the sum
```

```
if (a.equals(30) ) //unwrap and compare
```

# Math class

- Math class provides many arithmetic, trigonometric and logarithmic methods
- All these methods are static
- It is not possible to create Math class object
- Example methods:

static double sqrt(double)

static double sin(double)

static double random( )

# Date and time handling

- An instance of java.util.Date represents a specific instant in time with millisecond precision
- java.util.Calendar is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object
- Subclasses of Calendar can implement specific calendar systems such as Gregorian calendar, Lunar Calendar and Jewish calendar
- Currently, java.util.GregorianCalendar for the Gregorian calendar is supported in the Java API
- You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time
- use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date. The month parameter is 0-based, i.e., 0 is for January

# Date and time handling

- **Calendar.getInstance()** returns current date
- **getTime()** of Calendar object returns Date object
- **get(field)** method of calendar object returns individual elements of the date&time
- **add(field, number)** method of calendar object can be used to perform date calculations
- **set(y,m,d)** of calendar object can be used to change field values
- **set(field,value)** of calendar object used to change any field value

# Calendar - Examples

```
SimpleDateFormat sdf = new SimpleDateFormat("dd/MMM/yyyy");  
Calendar cal = new GregorianCalendar(2013,0,31);  
System.out.println(sdf.format(cal.getTime()));
```

```
cal.set(2010, 6, 21);  
System.out.println(sdf.format(cal.getTime()));
```

```
Calendar calendar = Calendar.getInstance();
```

```
int year          = calendar.get(Calendar.YEAR);  
int month         = calendar.get(Calendar.MONTH); // Jan = 0, dec = 11  
int dayOfMonth    = calendar.get(Calendar.DAY_OF_MONTH);  
int dayOfWeek     = calendar.get(Calendar.DAY_OF_WEEK);  
int weekOfYear    = calendar.get(Calendar.WEEK_OF_YEAR);  
int weekOfMonth   = calendar.get(Calendar.WEEK_OF_MONTH);
```

```
calendar.add(Calendar.MONTH, 1);  
calendar.add(Calendar.DAY_OF_MONTH, -10);
```



# Working with Local Date and Time

- The `java.time` API defines two classes for working with local dates and times (without a time zone):
  - `LocalDate`:
    - Does not include time
    - A year-month-day representation
    - `toString` – ISO 8601 format (YYYY-MM-DD)
  - `LocalTime`:
    - Does not include date
    - Stores hours:minutes:seconds.nanoseconds
    - `toString` – (HH:mm:ss.SSSS)

# LocalDate: Example

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

    public static void main(String[] args) {
        LocalDate now, bDate, nowPlusMonth, nextTues;
        now = LocalDate.now();
        out.println("Now: " + now);
        bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
        out.println("Java's Bday: " + bDate);
        out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
        out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
        out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
        nowPlusMonth = now.plusMonths(1);
        out.println("The date a month from now: " + nowPlusMonth);
        nextTues = now.with(next(TUESDAY));
        out.println("Next Tuesday's date: " + nextTues);
    }
}
```

next method

TUESDAY

LocalDate objects are  
immutable – methods  
return a new instance.

# Working with `LocalTime`

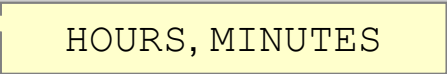
`LocalTime` stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with `LocalTime`
  - When is my lunch time?
  - Is lunch time in the future or past?
  - What is the time 1 hour 15 minutes from now?
  - How many minutes until lunch time?
  - How many hours until bedtime?
  - How do I keep track of just the hours and minutes?

# LocalTime: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
    public static void main(String[] args) {
        LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
        now = LocalTime.now();
        out.println("The time now is: " + now);
        nowPlus = now.plusHours(1).plusMinutes(15);
        out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
        nowHrsMins = now.truncatedTo(MINUTES);
        out.println("Truncate the current time to minutes: " + nowHrsMins);
        out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
        lunch = LocalTime.of(12, 30);
        out.println("Is lunch in my future? " + lunch.isAfter(now));
        long minsToLunch = now.until(lunch, MINUTES);
        out.println("Minutes til lunch: " + minsToLunch);
        bedtime = LocalTime.of(21, 0);
        long hrsToBedtime = now.until(bedtime, HOURS);
        out.println("How many hours until bedtime? " + hrsToBedtime);
    }
}
```



HOURS, MINUTES

| Prefix            | Example                                                                   | Use                                                                            |
|-------------------|---------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| now               | <code>today = LocalDate.now()</code>                                      | Creates an instance using the system clock                                     |
| of                | <code>meet = LocalTime.of(13, 30)</code>                                  | Creates an instance by using the parameters passed                             |
| get               | <code>today.get(DAY_OF_WEEK)</code>                                       | Returns part of the state of the target                                        |
| with              | <code>meet.withHour(12)</code>                                            | Returns a copy of the target object with one element changed                   |
| plus, minus       | <code>nextWeek.plusDays(7)</code><br><code>sooner.minusMinutes(30)</code> | Returns a copy of the object with the amount added or subtracted               |
| to                | <code>meet.toSecondOfDay()</code>                                         | Converts this object to another type. Here returns <code>int</code> seconds.   |
| at                | <code>today.atTime(13, 30)</code>                                         | Combines this object with another; returns a <code>LocalDateTime</code> object |
| until             | <code>today.until</code>                                                  | Calculates the amount of time until another date in terms of the unit          |
| isBefore, isAfter | <code>today.isBefore(lastWeek)</code>                                     | Compares this object with another on the timeline                              |
| isLeapYear        | <code>today.isLeapYear()</code>                                           | Checks if this object is a leap year                                           |

# Period

- The Period class models a date-based amount of time, such as five days, a week or three years.
- Duration class models a quantity or amount of time in terms of seconds and nanoseconds. It is used represent amount of time between two instants.
- Following table shows important and common methods of both:

| Method    | Uses                                                                |
|-----------|---------------------------------------------------------------------|
| between   | Use to create either Period or Duration between LocalDates.         |
| of        | Creates Period/Duration based on given year, months & days.         |
| ofXXX()   | Creates Period/Duration based on specified factors.                 |
| getXXX () | Used to return various part of Period/Duration.                     |
| plusXXX() | Add the specified factor and return a LocalDate.                    |
| minusXXX( | Subtracts the specified factor and return a LocalDate.              |
| isXXX()   | Performs checks on LocalDate and returns Boolean value.             |
| withXXX() | Returns a copy of LocalDate with the factor set to the given value. |

# Java I/O



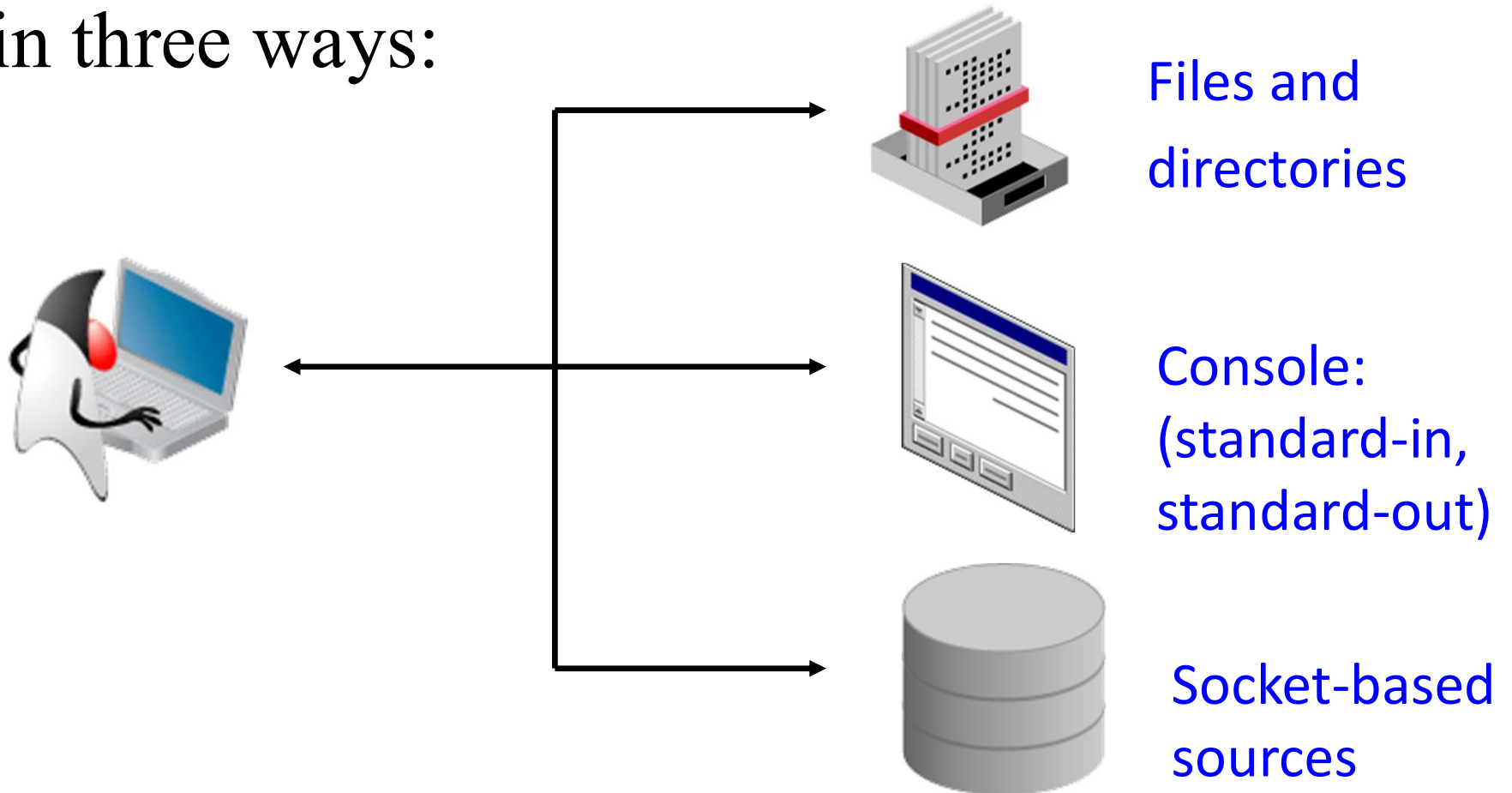
# I/O Streams

- A program uses an input stream to read data from a source, one item at a time.
- A program uses an output stream to write data to a destination (sink), one item at time.



# I/O Application

- Typically, a developer uses input and output in three ways:



# Data Within Streams

- Java supports two types of streams: character and byte.
- Input and output of character data is handled by readers and writers.
- Input and output of byte data is handled by input streams and output streams:
  - Normally, the term *stream* refers to a byte stream.
  - The terms *reader* and *writer* refer to character streams.

| Stream         | Byte Streams | Character Streams |
|----------------|--------------|-------------------|
| Source streams | InputStream  | Reader            |
| Sink streams   | OutputStream | Writer            |

# I/O Stream classes

|                | Byte-based Input                       | Byte-based Output                    | Character-based Input       | Character-based Output       |
|----------------|----------------------------------------|--------------------------------------|-----------------------------|------------------------------|
| Basic          | InputStream                            | OutputStream                         | Reader<br>InputStreamReader | Writer<br>OutputStreamWriter |
| Arrays         | ByteArrayInputStream                   | ByteArrayOutputStream                | CharArrayReader             | CharArrayWriter              |
| Files          | FileInputStream<br>RandomAccessFile    | FileOutputStream<br>RandomAccessFile | FileReader                  | FileWriter                   |
| Pipes          | PipedInputStream                       | PipedOutputStream                    | PipedReader                 | PipedWriter                  |
| Buffering      | BufferedInputStream                    | BufferedOutputStream                 | BufferedReader              | BufferedWriter               |
| Filtering      | FilterInputStream                      | FilterOutputStream                   | FilterReader                | FilterWriter                 |
| Parsing        | PushbackInputStream<br>StreamTokenizer |                                      |                             |                              |
| Strings        |                                        |                                      | StringReader                | StringWriter                 |
| Data           | DataInputStream                        | DataOutputStream                     |                             |                              |
| Data-Formatted |                                        | PrintStream                          |                             | PrintWriter                  |
| Objects        | ObjectInputStream                      | ObjectOutputStream                   |                             |                              |
| Utilities      | SequenceInputStream                    |                                      |                             |                              |

# InputStream Methods

- The three basic read methods are:

```
int read()           // return -1 at end of file
int read(byte[] buffer) // return no of bytes read or -1
int read(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close();           // Close an open stream
int available();        // Number of bytes available
long skip(long n);      // Discard n bytes from stream
```

# OutputStream Methods

- The three basic `write` methods are:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- Other methods include:

```
void close(); // close stream
void flush(); // Force a write to the stream
```

# Reader Methods

- The three basic read methods are:

```
int read()  
int read(char[] cbuf)  
int read(char[] cbuf, int offset, int length)
```

- Other methods include:

```
void close()  
boolean ready()  
long skip(long n)  
boolean markSupported()  
void mark(int readAheadLimit)  
void reset()
```

# Writer Methods

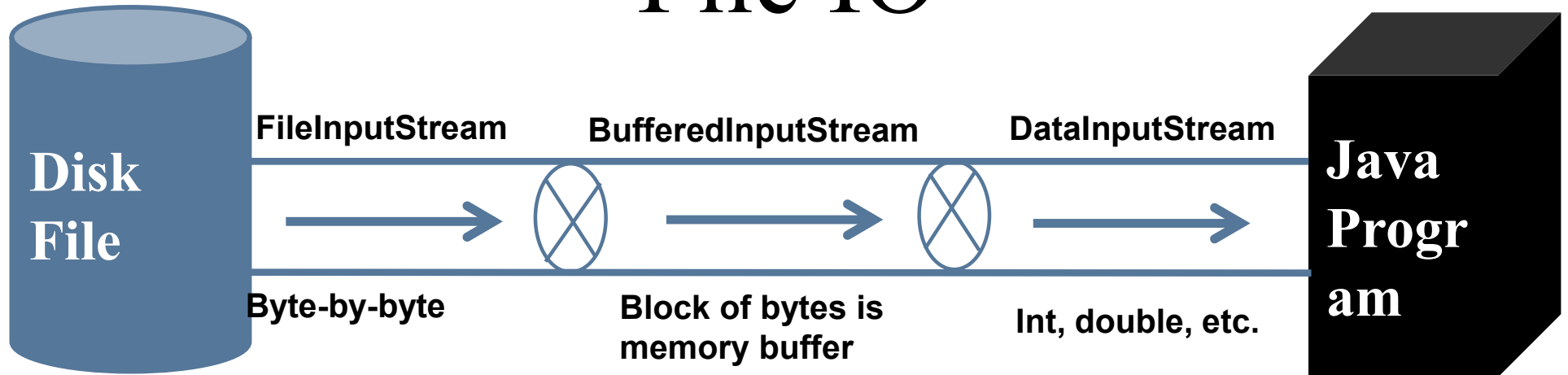
- The basic write methods are:

```
void write(int c)
void write(char[] cbuf)
void write(char[] cbuf, int offset, int length)
void write(String string)
void write(String string, int offset, int length)
```

- Other methods include:

```
void close()
void flush()
```

# File IO



Java enables reading and writing to files:

```
DataInputStream objDataInputStream = new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("inputFile.txt")));
```



# File Reading / writing

- The following classes used for binary data (byte streams)

FileInputStream

FileOutputStream

- The following classes used for character data

FileReader

FileWriter

## FileReader & FileWriter Example

```
import java.io.*;

public class CopyText {

    public static void main(String[] args) throws
        IOException
    {
        FileReader in = new FileReader("source.txt");
        FileWriter out = new FileWriter("Target.txt");
        int c;
        while (true) {
            c = in.read();
            if(c == -1)
                break;
            out.write(c);
        }
        in.close();
        out.close();
    }
}
```

# Binary files

```
import java.io.*;

class CopyBinary{
    public static void main(String args[]){
        int i;
        FileInputStream fin;
        FileOutputStream fout;

        try{
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);

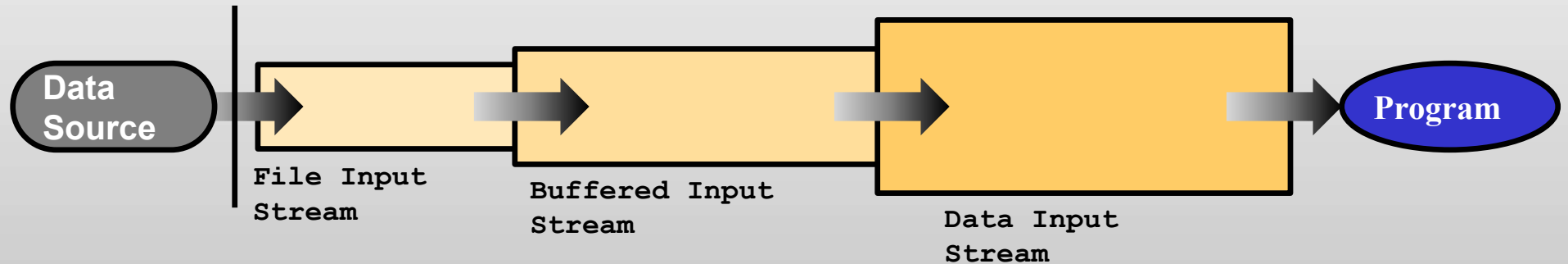
            while (true) {
                i = fin.read();
                if(i == -1)
                    break;
                fout.write(i);
            }
        }
    }
}
```

# Binary files (Contd.).

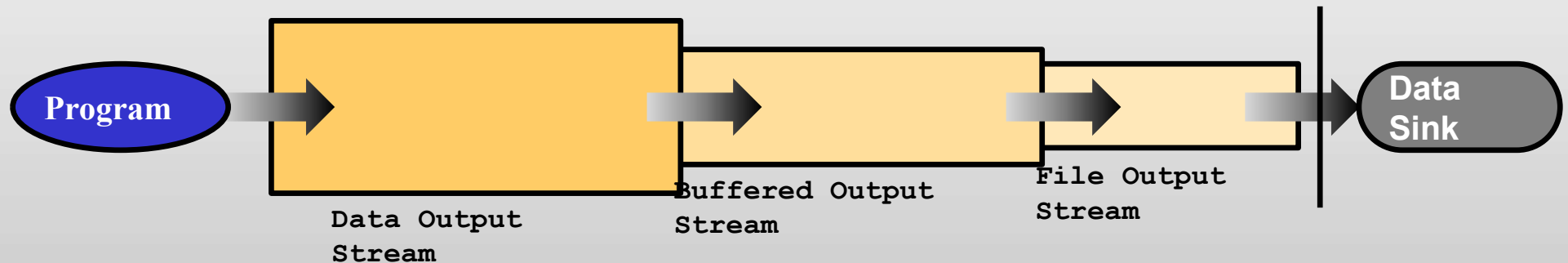
```
        fin.close();  
        fout.close();  
    } // try block  
  
    catch (FileNotFoundException e) {  
        System.out.println("File Not Found");  
    }  
  
    catch (IOException e) {  
        System.out.println("File Error"+e);  
    }  
}  
  
}
```

# I/O Stream Chaining

## Input Stream Chain



## Output Stream Chain



# Chained Streams: Example

```
1  import java.io.BufferedReader; import java.io.BufferedWriter;
2  import java.io.FileReader; import java.io.FileWriter;
3  import java.io.FileNotFoundException; import java.io.IOException;
4
5  public class BufferedStreamCopyTest {
6      public static void main(String[] args) {
7          try (BufferedReader bufInput
8              = new BufferedReader(new FileReader(args[0]));
9              BufferedWriter bufOutput
10                 = new BufferedWriter(new FileWriter(args[1]))) {
11              String line = "";
12              while ((line = bufInput.readLine()) != null) {
13                  bufOutput.write(line);
14                  bufOutput.newLine();
15              }
16          } catch (FileNotFoundException f) {
17              System.out.println("File not found: " + f);
18          } catch (IOException e) {
19              System.out.println("Exception: " + e);
20          }
21      }
22  }
```

A `FileReader` chained to a `BufferedReader`: This allows you to use a method that reads a `String`.

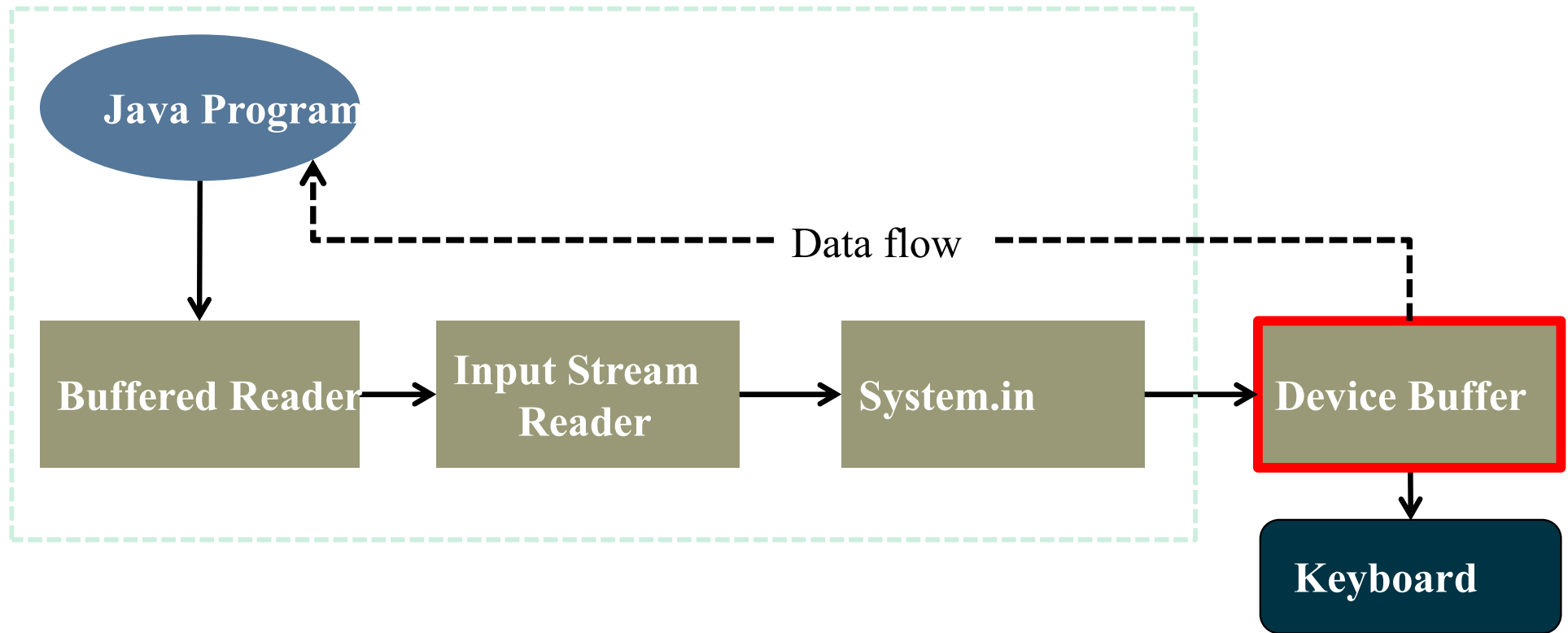
The character buffer replaced by a `String`. Note that `readLine()` uses the newline character as a terminator. Therefore, you must add that back to the output file.

# Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output*.
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input*.
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error*.

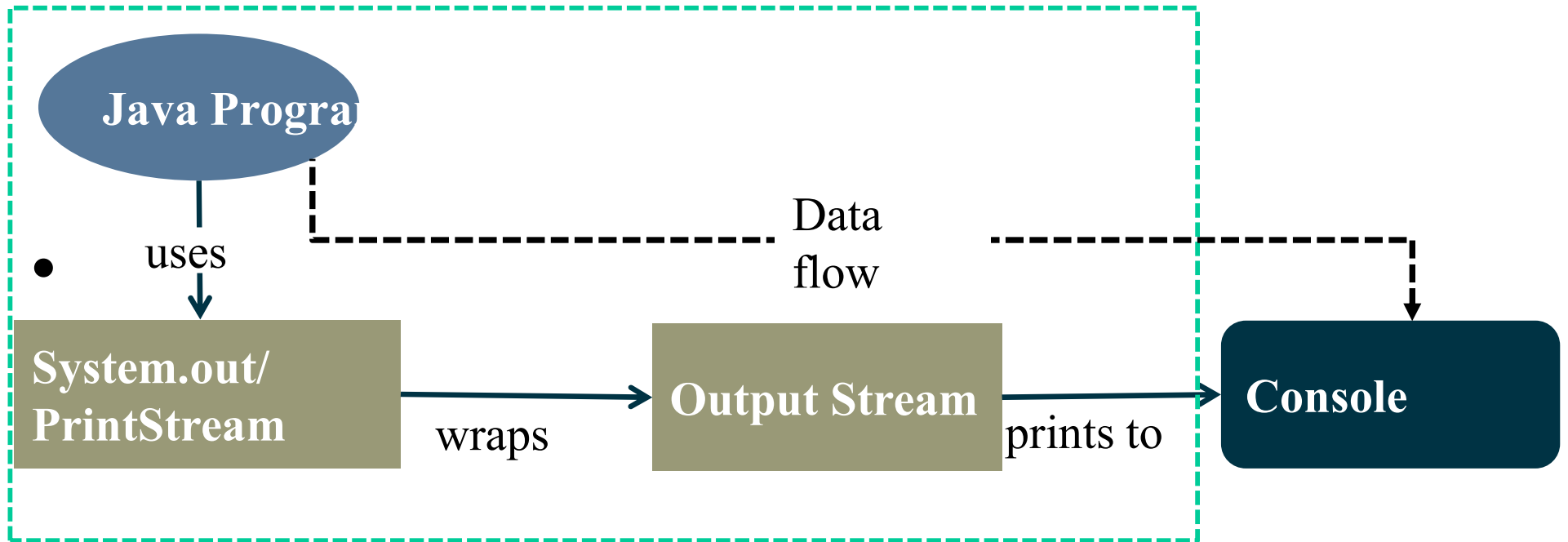
# Console input



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



# Console output



These are statements for displaying simple text messages to the user.

```
System.out.print("Welcome to Codington")  
System.out.println ("Welcome to Codington")
```

# Console I/O example

```
package m10.io;
import java.io.*;

public class BRReadLine{

    public static void main (String args[]) {
        String str;
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        System.out.println("Enter line, 'stop' to
quit");
        do {
            str = br.readLine( );
            System.out.println ( str );
        }while (!str.equals( "stop"));
    }
}
```

# Console Input Using the **Scanner** Class

- Starting with version 5.0, Java includes a class for doing simple keyboard input named the **Scanner** class
- The following line creates Scanner object linked to keyboard

```
Scanner sc = new Scanner(System.in) ;
```

- Once a **Scanner** object has been created, a program can then use that object to perform keyboard input using methods of the **Scanner** class

# Console Input Using the **Scanner** Class

- The method **nextInt** reads one **int** value typed in at the keyboard and assigns it to a variable:

```
int num = scan.nextInt();
```

- The method **nextDouble** reads one **double** value typed in at the keyboard and assigns it to a variable:

```
double d1 = scan.nextDouble();
```

- Similarly, it has methods for all other data types

**next()**                      for String

**nextLong()**      for long value

**nextShort()**      for short value

..... etc

# Console Input Using the **Scanner** Class

- The method **nextLine** reads an entire line of keyboard input
- The code

```
String line = sc.nextLine();
```

reads in an entire line and places the string that is read into the variable line

# Command line arguments

- To supply data while starting the program we can send command line arguments
- example :

```
java MyProgram string1 string2 string3
```

- These values (string1, string2, string3) will be supplied to the main function as String array
- Example:

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

# Using the command line

- However, we can give this array values by providing command line arguments when we start a program running

```
java MyProgram string1 string2 string3
```

```
args[0] args[1] args[2]
```

- To know the number of arguments we can use **args.length**
- For numeric data we have to parse the strings into corresponding data types

# Using the command line

- Sample program:

```
public class Echo {  
    public static void main(String [] args) {  
        System.out.println("first value: " + args[0]);  
        System.out.println("second value: " + args[1]);  
    }  
}
```

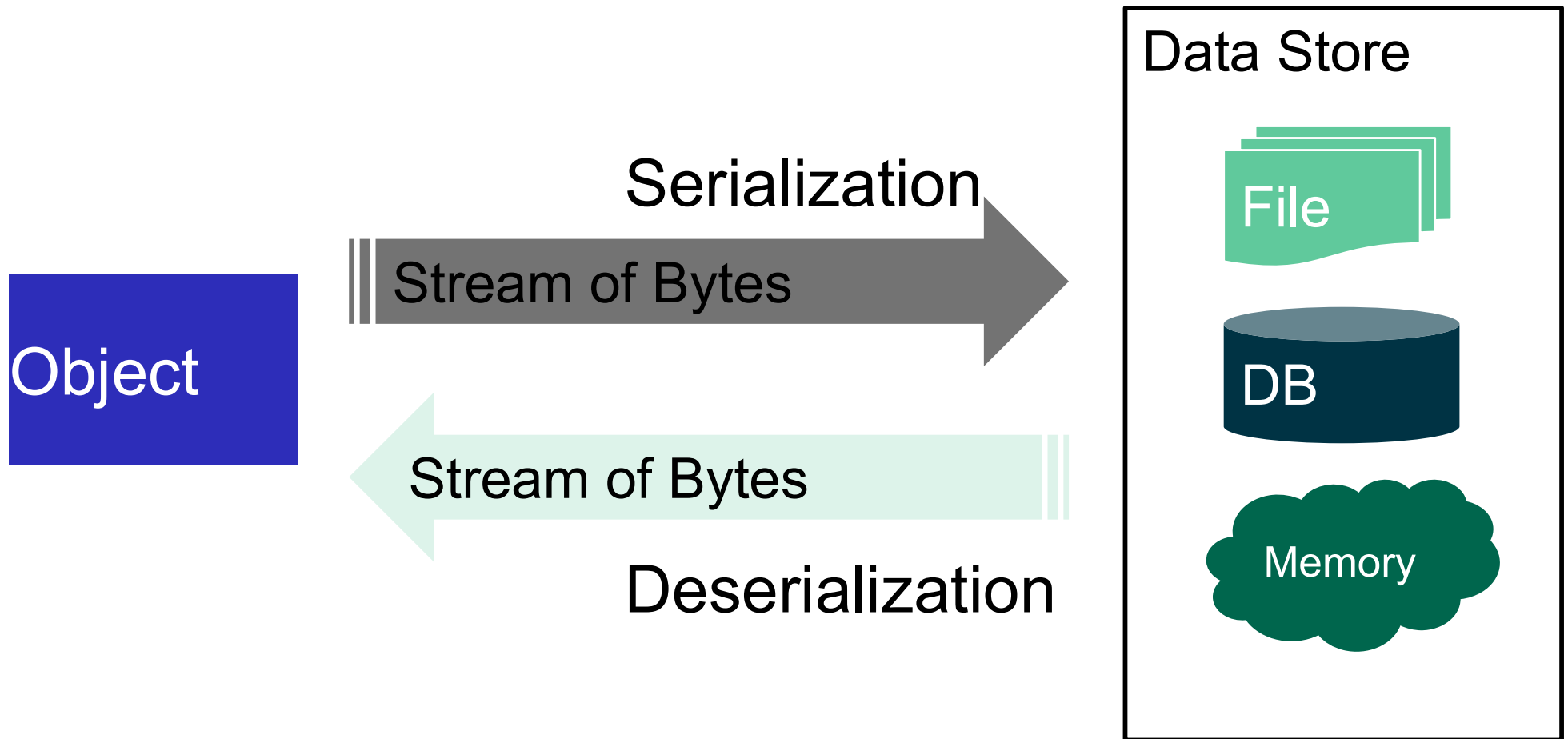
```
java Echo Amir Khan  
first value: Amir  
second value: Khan
```



# Persistence

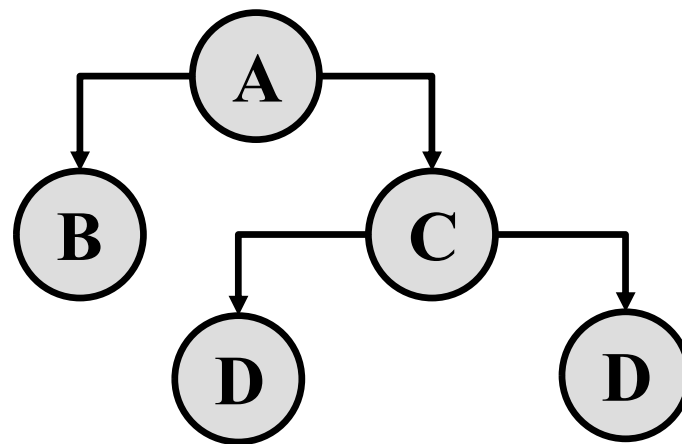
- Saving data to some type of permanent storage is called persistence. An object that is persistent-capable can be stored on disk (or any other storage device), or sent to another machine to be stored there.
  - A non-persisted object exists only as long as the Java Virtual Machine is running.
  - Java serialization is the standard mechanism for saving an object as a sequence of bytes that can later be rebuilt into a copy of the object.
  - To serialize an object of a specific class, the class must implement the `java.io.Serializable` interface.

# Serialization/Deserialization



# Serialization and Object Graphs

- When an object is serialized, only the fields of the object are preserved.
- When a field references an object, the fields of the referenced object are also serialized, if that object's class is also serializable.
- The tree of an object's fields constitutes the *object graph*.



# Serializing Objects

- How to Write to an **ObjectOutputStream**

```
FileOutputStream out = new FileOutputStream("theTime");  
ObjectOutputStream oos = new ObjectOutputStream(out);  
Employee emp = new Employee(-----); // Employee class should  
   // implement Serializable  
  
oos.writeObject( emp);  
oos.close();
```

- How to Read from an **ObjectInputStream**

```
FileInputStream in = new FileInputStream("theTime");  
ObjectInputStream ois = new ObjectInputStream(in);  
Employee x = (Employee) ois.readObject();  
System.out.println(x);
```

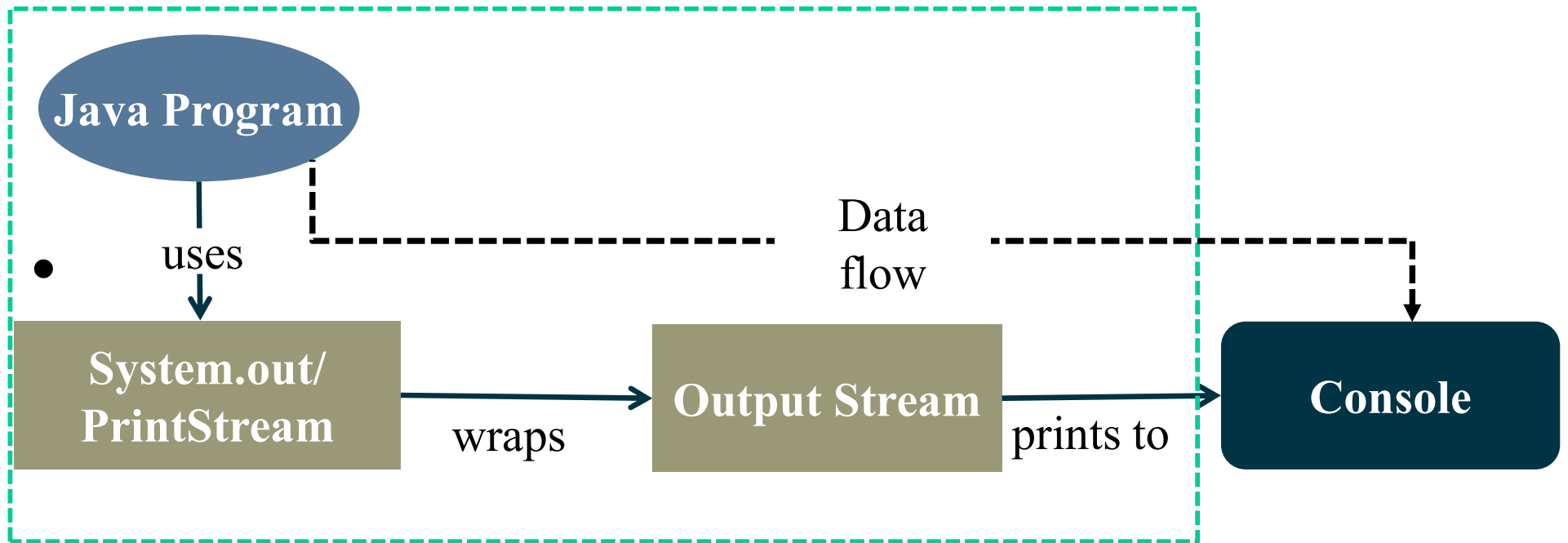
# SQL

# Console I/O

The `System` class in the `java.lang` package has three static instance fields: `out`, `in`, and `err`.

- The `System.out` field is a static instance of a `PrintStream` object that enables you to write to *standard output*.
- The `System.in` field is a static instance of an `InputStream` object that enables you to read from *standard input*.
- The `System.err` field is a static instance of a `PrintStream` object that enables you to write to *standard error*.

# Console output



These are statements for displaying simple text messages to the user.

```
System.out.print("Welcome to Codington")  
System.out.println ("Welcome to Codington")
```

# Console Input Using the **Scanner** Class

- Starting with version 5.0, Java includes a class for doing simple keyboard input named the **Scanner** class
- The following line creates Scanner object linked to keyboard

```
Scanner scan = new Scanner(System.in) ;
```

- Once a **Scanner** object has been created, a program can then use that object to perform keyboard input using methods of the **Scanner** class



# Console Input Using the **Scanner** Class

- The method **nextInt** reads one **int** value typed in at the keyboard and assigns it to a variable:

```
int num = scan.nextInt();
```

- The method **nextDouble** reads one **double** value typed in at the keyboard and assigns it to a variable:

```
double d1 = scan.nextDouble();
```

- Similarly, it has methods for all other data types

**next()**                      for String

**nextLong()**      for long value

**nextShort()**      for short value

..... etc

# Console Input Using the **Scanner** Class

- The method **nextLine** reads an entire line of keyboard input
- The code

```
String line = next.nextLine();
```

reads in an entire line and places the string that is read into the variable line

# Command line arguments

- To supply data while starting the program we can send command line arguments
- example :

```
java MyProgram string1 string2 string3
```

- These values (string1, string2, string3) will be supplied to the main function as String array
- Example:

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

# Using the command line

- However, we can give this array values by providing command line arguments when we start a program running

```
java MyProgram string1 string2 string3
```

```
args[0] args[1] args[2]
```

- To know the number of arguments we can use **args.length**
- For numeric data we have to parse the strings into corresponding data types

# Using the command line

- Sample program:

```
public class Echo {  
    public static void main(String [] args) {  
        System.out.println("first value: " + args[0]);  
        System.out.println("second value: " + args[1]);  
    }  
}
```

```
java Echo Amir Khan  
first value: Amir  
second value: Khan
```

# Multi Thread Programming

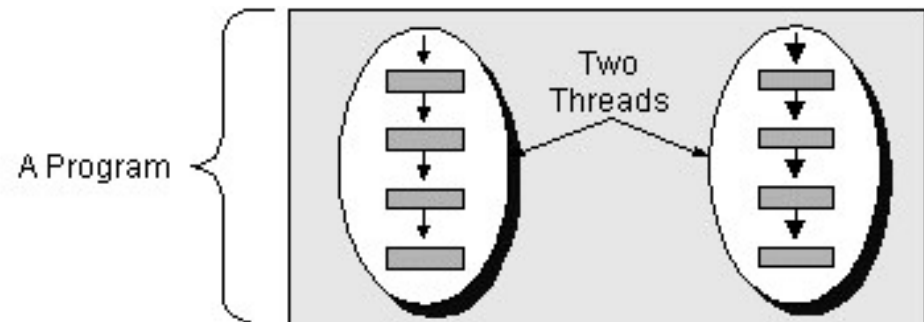
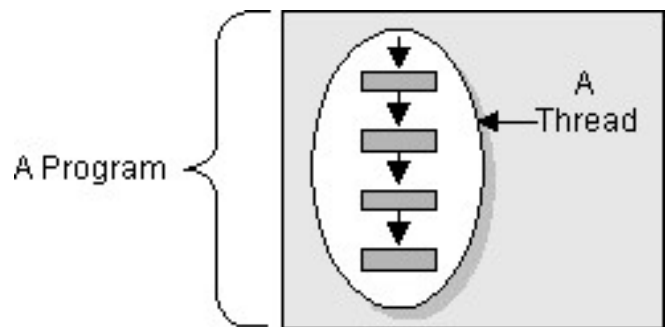


# What is Multithreading?

- In multithreading, the thread is the smallest unit of code that can be dispatched by the thread scheduler
- A single program can perform two tasks using two threads
- Only one thread will be executing at any given point of time given a single-processor architecture

# Threads

**A thread is a single sequential flow of control within a program**





# Threads

- **A thread is an object of type Thread defined in java.lang package**
- **Multi-threading functionality in Java is supported in four places**
  - **Thread class**
  - **Runnable interface**
  - **Object class**
  - **Java virtual machine**
- **There are two ways to create a “Thread”**
  - **Subclassing Thread and provide run() method**
  - **Implementing the Runnable Interface**

# The Thread class

- Java's multithreading feature is built into the **Thread** class
- The **Thread** class has two primary thread control methods:
  - public void start()** - The **start()** method starts a thread execution
  - public void run()** - The **run()** method actually performs the work of the thread and is the entry point for the thread
- The thread *dies* when the **run()** method terminates
- You never call **run()** explicitly
- The **start()** method called on a thread automatically initiates a call to the thread's **run()** method

# Extending Thread

- Instantiate the class that extends Thread
- This class must override run() method
- The code that should run as a thread will be part of this run() method
- We must call the start() method on this thread
- start( ) in turn calls the thread's run( ) method

# Extend Thread - Example

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i);  
            try {  
                sleep((long) (Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! ");  
    }  
}
```

## Activating the thread

```
SimpleThread tt = new SimpleThread("thread 1");  
tt.start();
```

# Implementing Runnable

- Create a class, which must implement the **interface Runnable**
- A thread can be constructed on any object that implements the **Runnable** interface.
- To implement **Runnable**, a class need implement only a single method called **run( )**
- After defining the class that implements Runnable, we have to create an object of type Thread passing the Runnable object to Thread constructor.
- This is mandatory because a thread object confers multithreaded functionality to the object from which it is created.
- Therefore, at the moment of thread creation, the thread object must know the reference of the object to which it has to confer multithreaded functionality.

# Implementing Runnable - Example

```
public class SimpleThread implements Runnable {  
    String name ;  
    public SimpleThread(String str) {  
        name = str;;  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i);  
            try {  
                sleep((long) (Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE! ");  
    }  
}
```

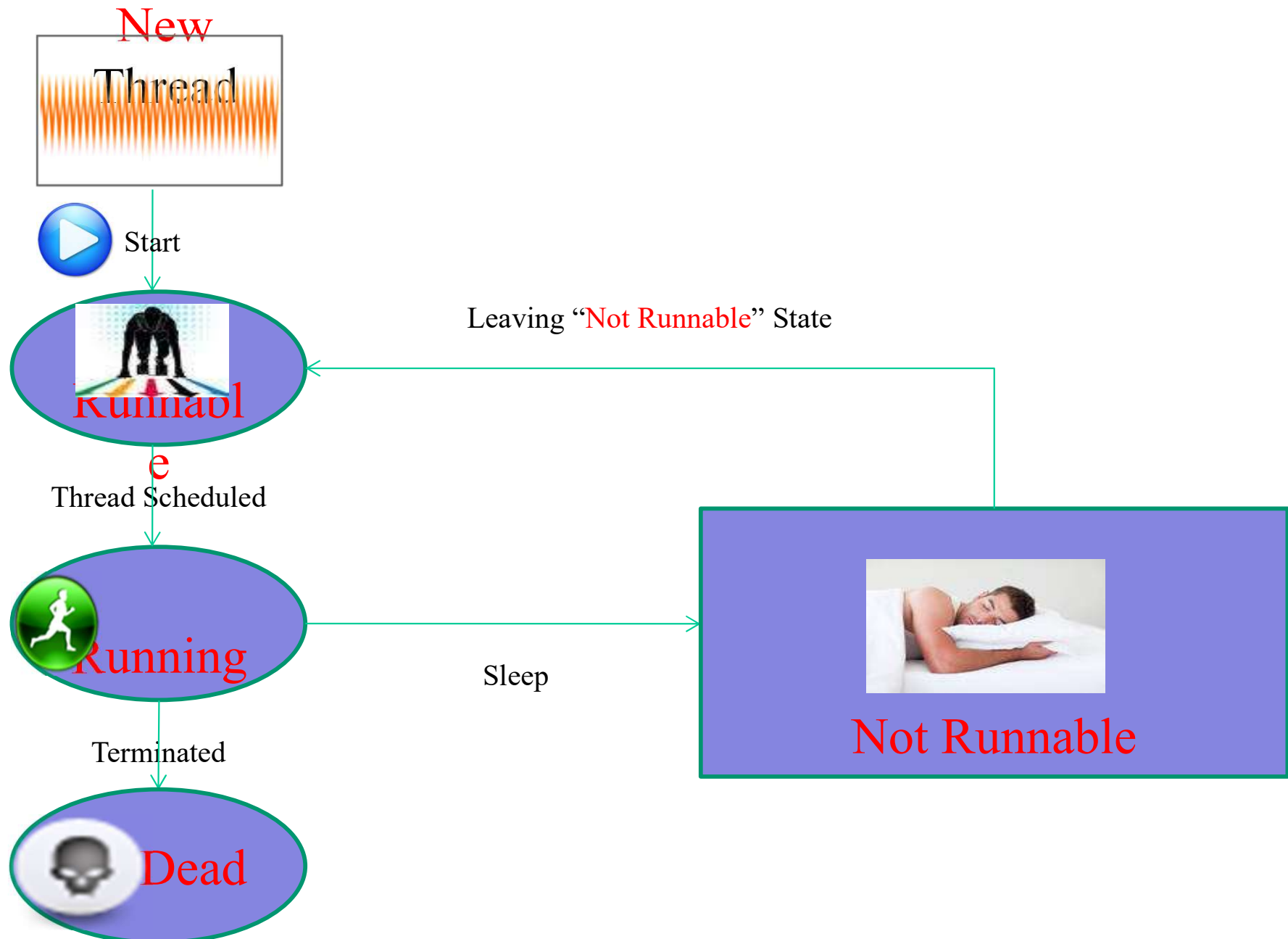
## Activating the thread

```
SimpleThread st = new SimpleThread("Thread 1");  
Thread tt = new Thread(st);  
tt.start();
```

# Control Thread Execution

- Two ways exist by which you can determine whether a thread has finished:
- The **isAlive( )** method will return true if the thread upon which it is called is still running; else it will return false
- The **join( )** method waits until the thread on which it is called terminates.

# Different States of a Thread





# Synchronization

- It is normal for threads to be sharing objects and data
- Different threads shouldn't try to access and change the same data at the same time
- Threads must therefore be synchronized
- For example, imagine a Java application where one thread deleting characters in a StringBuffer object, while a second thread inserting characters at the same place of the StringBuffer object

# Synchronization

- The current thread operating on the shared data structure, must be granted mutually exclusive access to the data
- The current thread gets an exclusive lock on the shared data structure, or a **mutex**
- A **mutex** is a concurrency control mechanism used to ensure the integrity of a shared data structure

# Synchronized Methods

- Synchronized methods are an elegant way of locking the object
- Implementation of concurrency control mechanism is very simple because every Java object has its own implicit monitor associated with it
- If a thread wants to enter an object's monitor, it has to just call the synchronized method of that object
- While a thread is executing a synchronized method, all other threads that are trying to invoke that particular synchronized method or any other synchronized method of the same object, will have to wait

# Synchronization

- Every object in Java has a lock
- Using *synchronization* enables the lock and allows only one thread to access that part of code
- Synchronization can be applied to:
  - A method  
`public synchronized void withdraw(){...}`
  - A block of code  
`synchronized (objectReference){...}`
- Synchronized methods in subclasses use same locks as their superclasses

# Thread Messaging

- In Java, you need not depend on the OS to establish communication between threads
- All objects have predefined methods, which can be called to provide inter-thread communication

# Inter-Thread Communication

- Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request.
- The words wait and notify encapsulate the two central concepts to thread communication
  - A thread waits for some condition or event to occur.
  - You notify a waiting thread that a condition or event has occurred.
- To avoid polling, Java's elegant inter-thread communication mechanism uses:
  - `wait( )`
  - `notify( )`, and `notifyAll( )`

## Inter-Thread Communication (Contd.).

- `wait( )`, `notify( )` and `notifyAll( )`
  - Are declared as final in Object
  - Hence, these methods are available to all classes
  - These methods can only be called from a synchronized context
- `wait( )` directs the calling thread to surrender the monitor, and go to sleep until some other thread enters the monitor of the same object, and calls `notify( )`
- `notify( )` wakes up the other thread which was waiting on the same object(*that had called wait() previously on the same object*)

# Inter-Thread Communication - example

```
public class MessageBox {  
    String message=null;  
    synchronized String get() {  
        if (message==null)  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                System.out.println("InterruptedException caught");  
            }  
        String newMessage=message;  
        message=null;  
        notify();  
        return newMessag;  
    }  
}
```



# Inter-Thread Communication - example

```
synchronized void put(String message) {  
    if (this.message != null)  
        try {  
            wait();  
        } catch (InterruptedException e) {  
            System.out.println("InterruptedException caught");  
        }  
    this.message = message;  
    System.out.println("Put: " + message);  
    notify();  
}  
}
```

# Inter-Thread Communication - example

```
public class Consumer implements Runnable {
```

```
    MessageBox box;
```

```
public Consumer(MessageBox box) {
```

```
    this.box = box;
```

```
}
```

```
public void run() {
```

```
    int i = 0;
```

```
    while (++i < 6) {
```

```
        String msg=box.get();
```

```
        System.out.println("Got "+msg);
```

```
    }
```

```
}
```

```
}
```

# Inter-Thread Communication - example

```
Public class Producer implements Runnable {
```

```
    MessageBox myBox;
```

```
    Producer(MessageBox box) {
```

```
        myBox = box;
```

```
    }
```

```
    public void run() {
```

```
        int i = 0;
```

```
        while (++i < 6) {
```

```
            myBox.put("Message " + i);
```

```
        }
```

```
    }
```

```
}
```

# Inter-Thread Communication - example

```
public class Main {  
  
    public static void main(String args[]) {  
        MessageBox box = new MessageBox();  
        Producer prod = new Producer(box);  
        Consumer con = new Consumer(box);  
        new Thread(con).start();  
        new Thread(prod).start();  
    }  
}
```

# DeadLock

- A simple deadlock situation is one in which a two threads are waiting
- Each thread waiting for a resource which is held by the other waiting thread
- this resource is usually the object lock obtained by the synchronized keyword

# DeadLock - Example

```
class Account {  
    private int id;  
    private int balance;  
  
    public void deposit(int amount) {  
        this.balance += amount;  
    }  
  
    public void withdraw(int amount) {  
        this.balance -= amount;  
    }  
  
    public static void transfer(Account from, Account to, int amount) {  
        synchronized (from) {  
            synchronized (to) {  
                from.withdraw(amount);  
                to.deposit(amount);  
            }  
        }  
    }  
}
```