

Java Tutorial



Write Once, Run Anywhere



Java - General

- Java is:
 - platform independent programming language
 - similar to C++ in syntax
 - similar to Smalltalk in mental paradigm
- Pros: also ubiquitous to net
- Cons: interpreted, and still under development (moving target)

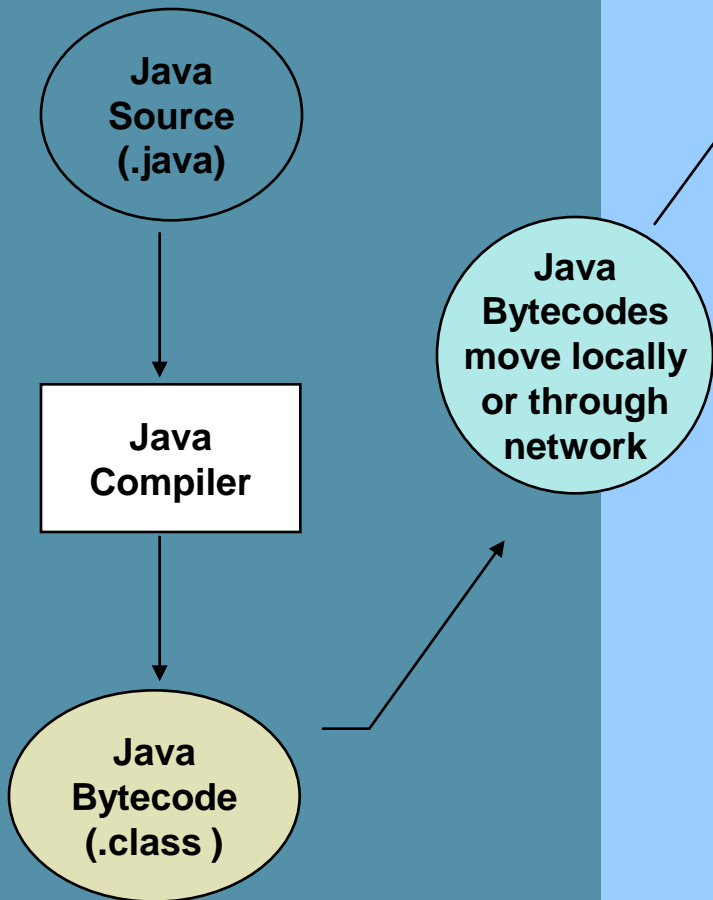


Java - General

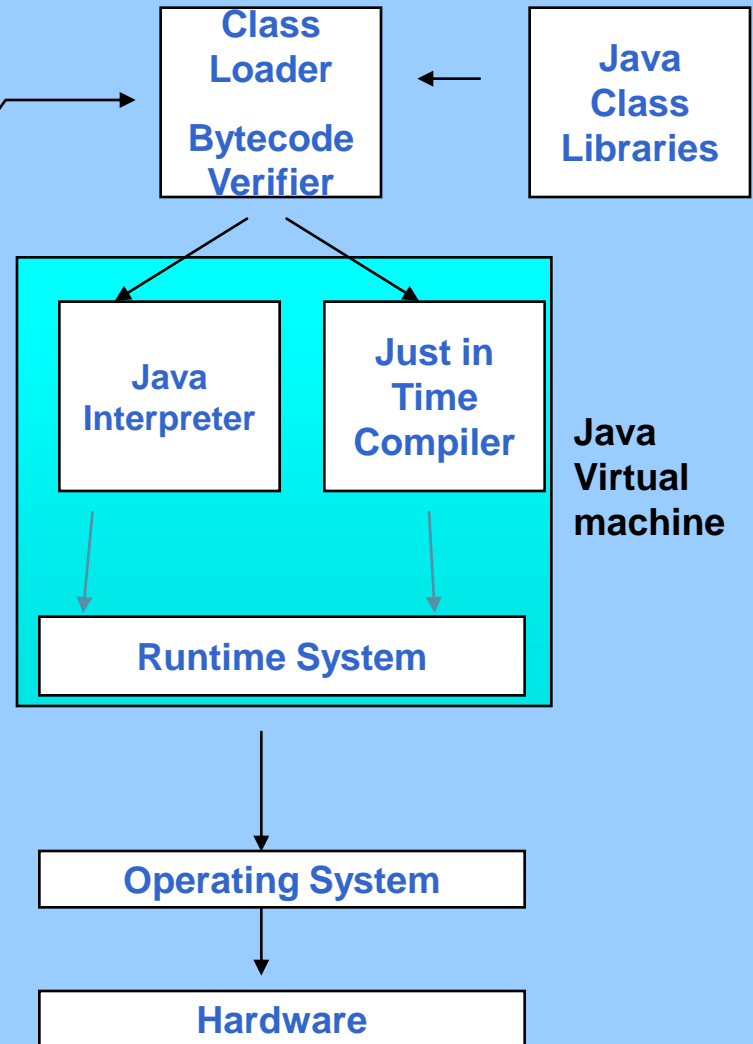
- Java has some interesting features:
 - automatic type checking,
 - automatic garbage collection,
 - simplifies pointers; no directly accessible pointer to memory,
 - simplified network access,
 - multi-threading!

How it works...!

Compile-time Environment



Compile-time Environment



Java Bytecodes move locally or through network

Java Virtual machine



How it works...!

- Java is independent only for one reason:
 - Only depends on the Java Virtual Machine (JVM),
 - code is compiled to *bytecode*, which is interpreted by the resident JVM,
 - JIT (just in time) compilers attempt to increase speed.



Java - Security

- Pointer denial - reduces chances of virulent programs corrupting host,
- Applets even more restricted -
 - May not
 - run local executables,
 - Read or write to local file system,
 - Communicate with any server other than the originating server.



Object-Oriented

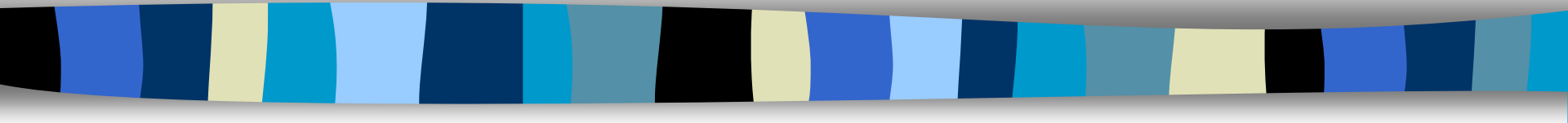
- Java supports OOD
 - Polymorphism
 - Inheritance
 - Encapsulation
- Java programs contain nothing but definitions and instantiations of classes
 - Everything is encapsulated in a class!



Java Advantages

- Portable - Write Once, Run Anywhere
- Security has been well thought through
- Robust memory management
- Designed for network programming
- Multi-threaded (multiple simultaneous tasks)
- Dynamic & extensible (loads of libraries)
 - Classes stored in separate files
 - Loaded only when needed

Basic Java Syntax





Primitive Types and Variables

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

- These basic (or primitive) types are the only types that are not objects (due to performance issues).

- This means that you don't use the new operator to create a primitive variable.

- Declaring primitive variables:

```
float initVal;
```

```
int retVal, index = 2;
```

```
double gamma = 1.2, brightness
```

```
boolean valueOk = false;
```



Initialisation

- If no value is assigned prior to use, then the compiler will give an error
- Java sets primitive variables to zero or false in the case of a boolean variable
- All object references are initially set to null
- An array of anything is an object
 - Set to null on declaration
 - Elements to zero false or null on creation



Declarations

```
int index = 1.2;           // compiler error
boolean retOk = 1;        // compiler error
double fiveFourths = 5 / 4; // no error!
float ratio = 5.8f;       // correct
double fiveFourths = 5.0 / 4.0; // correct
```

- 1.2f is a float value accurate to 7 decimal places.
- 1.2 is a double value accurate to 15 decimal places.



Assignment

- All Java assignments are right associative

```
int a = 1, b = 2, c = 5
```

```
a = b = c
```

```
System.out.print(
```

```
“a= “ + a + “b= “ + b + “c= “ + c)
```

- What is the value of a, b & c
- Done right to left: `a = (b = c);`

Basic Mathematical Operators

- `*` `/` `%` `+` `-` are the mathematical operators
- `*` `/` `%` have a higher precedence than `+` or `-`

```
double myVal = a + b % d - c * d / b;
```

- Is the same as:

```
double myVal = (a + (b % d)) -  
                ((c * d) / b);
```



Statements & Blocks

- A simple statement is a command terminated by a semi-colon:

```
name = "Fred";
```

- A block is a compound statement enclosed in curly brackets:

```
{  
    name1 = "Fred"; name2 = "Bill";  
}
```

- Blocks may contain other blocks



Flow of Control

- Java executes one statement after the other in the order they are written
- Many Java statements are flow control statements:

Alternation: if, if else, switch

Looping: for, while, do while

Escapes: break, continue, return



If – The Conditional Statement

- The if statement evaluates an expression and if that evaluation is true then the specified action is taken

```
if ( x < 10 ) x = 10;
```

- If the value of x is less than 10, make x equal to 10
- It could have been written:

```
if ( x < 10 )
```

```
  x = 10;
```

- Or, alternatively:

```
if ( x < 10 ) { x = 10; }
```



Relational Operators

==	Equal (careful)
!=	Not equal
>=	Greater than or equal
<=	Less than or equal
>	Greater than
<	Less than



If... else

- The if ... else statement evaluates an expression and performs one action if that evaluation is true or a different action if it is false.

```
if (x != oldx) {  
    System.out.print("x was changed");  
}  
else {  
    System.out.print("x is unchanged");  
}
```



Nested if ... else

```
if ( myVal > 100 ) {  
    if ( remainderOn == true) {  
        myVal = mVal % 100;  
    }  
    else {  
        myVal = myVal / 100.0;  
    }  
}  
else  
{  
    System.out.print("myVal is in range");  
}
```

else if

■ Useful for choosing between alternatives:

```
if ( n == 1 ) {  
    // execute code block #1  
}  
else if ( j == 2 ) {  
    // execute code block #2  
}  
else {  
    // if all previous tests have failed,  
    execute code block #3  
}
```



A Warning...

WRONG!

```
if( i == j )
    if ( j == k )
        System.out.print(
            "i equals k");
else
    System.out.print(
        "i is not equal
        to j");
```

CORRECT!

```
if( i == j ) {
    if ( j == k )
        System.out.print(
            "i equals k");
}
else
    System.out.print("
    i is not equal to
    j");    // Correct!
```



The switch Statement

```
switch ( n ) {  
    case 1:  
        // execute code block #1  
        break;  
    case 2:  
        // execute code block #2  
        break;  
    default:  
        // if all previous tests fail then  
        //execute code block #4  
        break;  
}
```



The **for** loop

- Loop n times

```
for ( i = 0; i < n; i++ ) {  
    // this code body will execute n times  
    // i from 0 to n-1  
}
```

- Nested for:

```
for ( j = 0; j < 10; j++ ) {  
    for ( i = 0; i < 20; i++ ){  
        // this code body will execute 200 times  
    }  
}
```




while loops

```
while(response == 1) {  
    System.out.print( "ID =" +  
        userID[n]);  
    n++;  
    response = readInt( "Enter ");  
}
```

What is the minimum number of times the loop is executed?

What is the maximum number of times?



do { ... } while loops

```
do {  
    System.out.print( "ID =" + userID[n] );  
    n++;  
    response = readInt( "Enter " );  
}while (response == 1);
```

What is the minimum number of times the loop is executed?

What is the maximum number of times?



Break

- A break statement causes an exit from the innermost containing `while`, `do`, `for` or `switch` statement.

```
for ( int i = 0; i < maxID, i++ ) {  
    if ( userID[i] == targetID ) {  
        index = i;  
        break;  
    }  
} // program jumps here after break
```



Continue

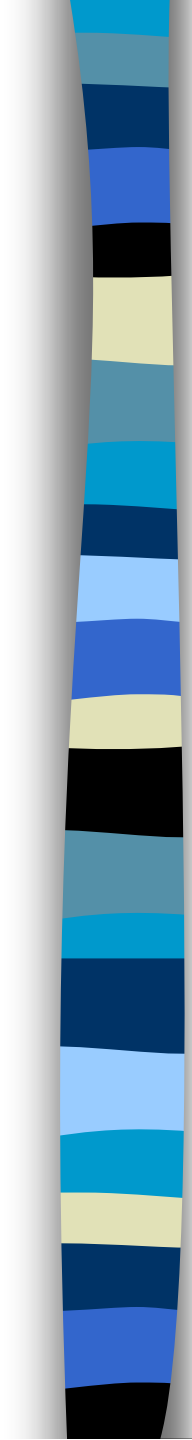
- Can only be used with while, do or for.
- The continue statement causes the innermost loop to start the next iteration immediately

```
for ( int i = 0; i < maxID; i++ ) {  
    if ( userID[i] != -1 ) continue;  
    System.out.print( "UserID " + i + " : " +  
        userID);  
}
```



Arrays

- An array is a list of similar things
- An array has a fixed:
 - name
 - type
 - length
- These must be declared when the array is created.
- Arrays sizes cannot be changed during the execution of the code



myArray =

3	6	3	1	6	3	4	1
0	1	2	3	4	5	6	7

myArray has room for 8 elements

- the elements are accessed by their index
- in Java, array indices start at 0



Declaring Arrays

```
int myArray[];
```

declares *myArray* to be an array of integers

```
myArray = new int[8];
```

sets up 8 integer-sized spaces in memory,
labelled *myArray[0]* to *myArray[7]*

```
int myArray[] = new int[8];
```

combines the two statements in one line



Assigning Values

- refer to the array elements by index to store values in them.

```
myArray[0] = 3;
```

```
myArray[1] = 6;
```

```
myArray[2] = 3; ...
```

- can create and initialise in one step:

```
int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};
```




Iterating Through Arrays

- *for* loops are useful when dealing with arrays:

```
for (int i = 0; i <
    myArray.length; i++) {
    myArray[i] = getsomevalue();
}
```



Arrays of Objects

- So far we have looked at an array of primitive types.
 - integers
 - could also use doubles, floats, characters...
- Often want to have an array of objects
 - Students, Books, Loans
- Need to follow 3 steps.



Declaring the Array

1. Declare the array

```
private Student studentList[];
```

- this declares studentList

2 .Create the array

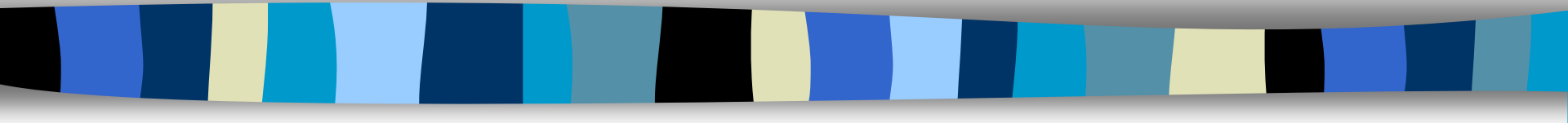
```
studentList = new Student[10];
```

- this sets up 10 spaces in memory that can hold references to Student objects

3. Create Student objects and add them to the

```
array: studentList[0] = new  
Student("Cathy", "Computing");
```

Java Methods & Classes





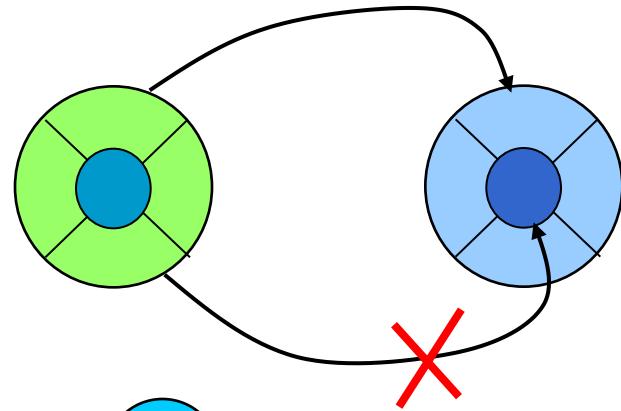
Classes ARE Object Definitions

- OOP - object oriented programming
- code built from objects
- Java these are called ***classes***
- Each class definition is coded in a separate .java file
- Name of the object must match the class/object name

The three principles of OOP

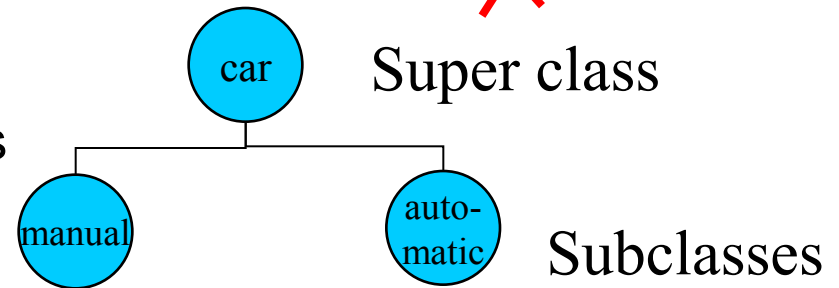
■ Encapsulation

- Objects hide their functions (**methods**) and data (**instance variables**)



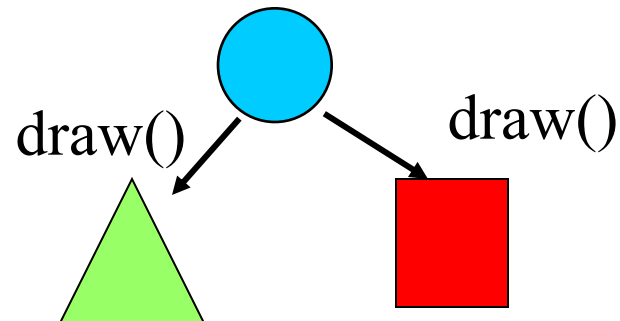
■ Inheritance

- Each **subclass** inherits all variables of its **superclass**



■ Polymorphism

- Interface same despite different data types





Simple Class and Method

```
Class Fruit{  
    int grams;  
    int cals_per_gram;  
  
    int total_calories() {  
        return(grams*cals_per_gram);  
    }  
}
```



Methods

- A method is a named sequence of code that can be invoked by other Java code.
- A method takes some parameters, performs some computations and then optionally returns a value (or object).
- Methods can be used as part of an expression statement.

```
public float convertCelsius(float tempC) {  
    return( ((tempC * 9.0f) / 5.0f) + 32.0 );  
}
```




Method Signatures

- A method signature specifies:
 - The name of the method.
 - The type and name of each parameter.
 - The type of the value (or object) returned by the method.
 - The checked exceptions thrown by the method.
 - Various method modifiers.
 - *modifiers type name (parameter list) [throws exceptions]*
- ```
public float convertCelsius (float tCelsius) {}
public boolean setUserInfo (int i, int j, String name) throws
 IndexOutOfBoundsException {}
```



# Public/private

- Methods/data may be declared ***public*** or ***private*** meaning they may or may not be accessed by code in other classes ...
- Good practice:
  - keep data private
  - keep most methods private
- well-defined interface between classes - helps to eliminate errors



# Using objects

- Here, code in one class creates an instance of another class and does something with it

...

```
Fruit plum=new Fruit();
int cal;
cal = plum.total_calories();
```

- ***Dot operator*** allows you to access (public) data/methods inside Fruit class



# Constructors

- The line

```
plum = new Fruit();
```

- invokes a constructor method with which you can set the initial data of an object
- You may choose several different type of constructor with different argument lists

```
eg Fruit(), Fruit(a) ...
```



# Overloading

- Can have several versions of a method in class with different types/numbers of arguments

```
Fruit() {grams=50;}
```

```
Fruit(a,b) { grams=a; calcs_per_gram=b;}
```

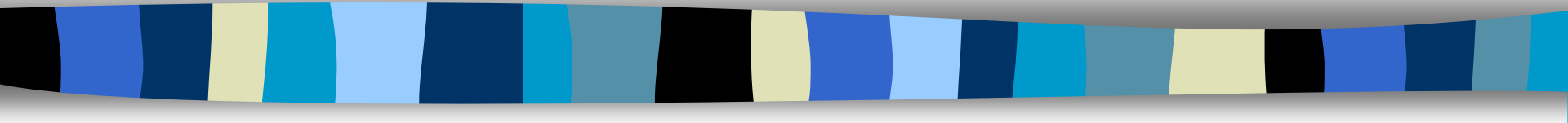
- By looking at arguments Java decides which version to use



# Java Development Kit

- javac - The Java Compiler
- java - The Java Interpreter
- jdb - The Java Debugger
- appletviewer - Tool to run the applets
  
- javap - to print the Java bytecodes
- javaprof - Java profiler
- javadoc - documentation generator
- javah - creates C header files

# Stream Manipulation





# Streams and I/O

- basic classes for file IO

- FileInputStream, for reading from a file
- FileOutputStream, for writing to a file

- Example:

Open a file "myfile.txt" for **reading**

```
FileInputStream fis = new FileInputStream("myfile.txt");
```

Open a file "outfile.txt" for **writing**

```
FileOutputStream fos = new FileOutputStream ("myfile.txt");
```



# Display File Contents

```
import java.io.*;
public class FileToOut1 {
 public static void main(String args[]) {
 try {
 FileInputStream infile = new FileInputStream("testfile.txt");
 byte buffer[] = new byte[50];
 int nBytesRead;
 do {
 nBytesRead = infile.read(buffer);
 System.out.write(buffer, 0, nBytesRead);
 } while (nBytesRead == buffer.length);
 }
 catch (FileNotFoundException e) {
 System.err.println("File not found");
 }
 catch (IOException e) { System.err.println("Read failed"); }
 }
}
```



# Filters

- Once a stream (e.g., file) has been opened, we can attach filters
- Filters make reading/writing more efficient
- Most popular filters:
  - For basic types:
    - `DataInputStream`, `DataOutputStream`
  - For objects:
    - `ObjectInputStream`, `ObjectOutputStream`

# Writing data to a file using Filters

```
import java.io.*;
public class GenerateData {
 public static void main(String args[]) {
 try {
 FileOutputStream fos = new FileOutputStream("stuff.dat");
 DataOutputStream dos = new DataOutputStream(fos);
 dos.writeInt(2);
 dos.writeDouble(2.7182818284590451);
 dos.writeDouble(3.1415926535);
 dos.close(); fos.close();
 }
 catch (FileNotFoundException e) {
 System.err.println("File not found");
 }
 catch (IOException e) {
 System.err.println("Read or write failed");
 }
 }
}
```

# Reading data from a file using filters

```
import java.io.*;
public class ReadData {
 public static void main(String args[]) {
 try {
 FileInputStream fis = new FileInputStream("stuff.dat");
 DataInputStream dis = new DataInputStream(fis);
 int n = dis.readInt();
 System.out.println(n);
 for(int i = 0; i < n; i++) { System.out.println(dis.readDouble());
 }
 dis.close(); fis.close();
 }
 catch (FileNotFoundException e) {
 System.err.println("File not found");
 }
 catch (IOException e) { System.err.println("Read or write failed");
 }
 }
}
```



# Object serialization

Write objects to a file, instead of writing primitive types.

Use the `ObjectInputStream`, `ObjectOutputStream` classes, the same way that filters are used.

# Write an object to a file

```
import java.io.*;
import java.util.*;
public class WriteDate {
 public WriteDate () {
 Date d = new Date();
 try {
 FileOutputStream f = new FileOutputStream("date.ser");
 ObjectOutputStream s = new ObjectOutputStream (f);
 s.writeObject (d);
 s.close ();
 }
 catch (IOException e) { e.printStackTrace(); }

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

# Read an object from a file

```
import java.util.*;
public class ReadDate {
 public ReadDate () {
 Date d = null;
 ObjectInputStream s = null;
 try { FileInputStream f = new FileInputStream ("date.ser");
 s = new ObjectInputStream (f);
 } catch (IOException e) { e.printStackTrace(); }
 try { d = (Date)s.readObject (); }
 catch (ClassNotFoundException e) { e.printStackTrace(); }
 catch (InvalidClassException e) { e.printStackTrace(); }
 catch (StreamCorruptedException e) { e.printStackTrace(); }
 catch (OptionalDataException e) { e.printStackTrace(); }
 catch (IOException e) { e.printStackTrace(); }
 System.out.println ("Date serialized at: "+ d);
 }
 public static void main (String args[]) { new ReadDate (); }
}
```