

# *Software Design*

---

## *Introduction to the Java Programming Language*

Material drawn from [JDK99,Sun96,Mitchell99,Mancoridis00]

# *Java Features*

---

- “Write Once, Run Anywhere.”
- Portability is possible because of Java virtual machine technology:
  - Interpreted
  - JIT Compilers
- Similar to C++, but “cleaner”:
  - No pointers, typedef, preprocessor, structs, unions, multiple inheritance, goto, operator overloading, automatic coercions, free.

# *Java Subset for this Course*

---

- We will focus on a subset of the language that will allow us to develop a distributed application using CORBA.
- Input and output will be character (terminal) based.
- For detailed treatment of Java visit:
  - <http://java.sun.com/docs/books/tutorial/index.html>

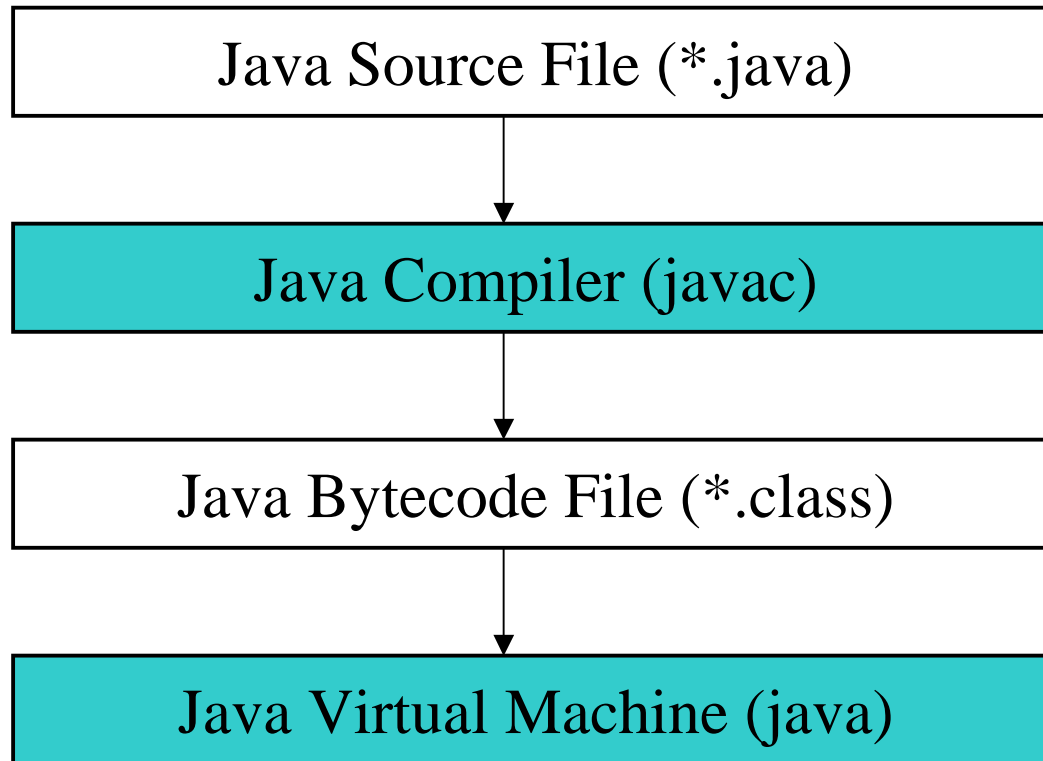
# *Java Virtual Machine*

---

- Java programs run on a Java Virtual Machine.
- Features:
  - Security
  - Portability
  - Superior dynamic resource management
  - Resource location transparency
  - Automatic garbage collection

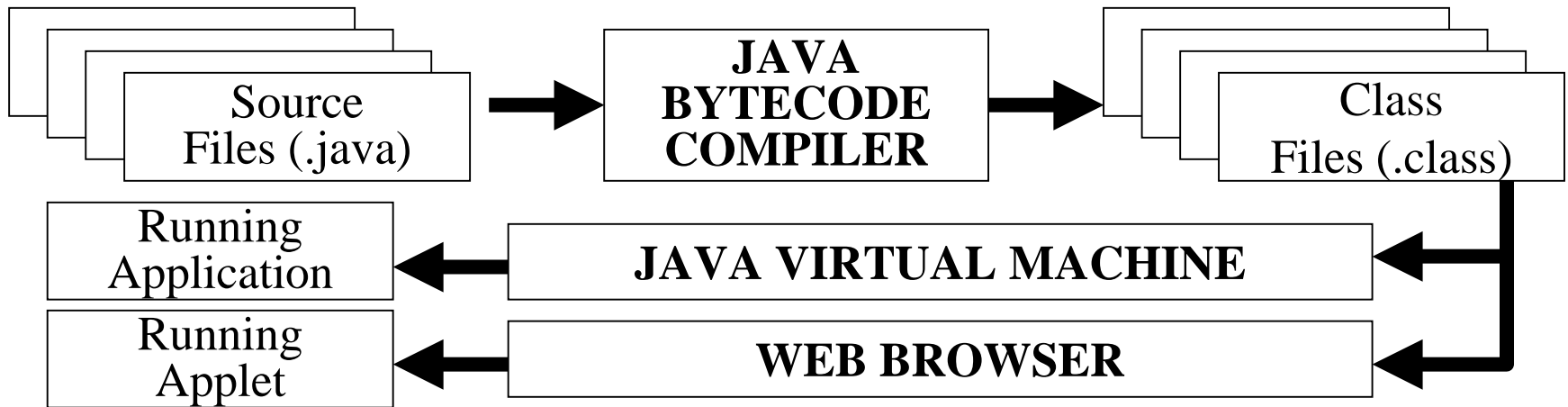
# *The Java Environment*

---



# Program Organization

---



# *Program Organization Standards*

---

- Each class is implemented in its own source file.
- Include one class per file:
  - Name of the Java file is the same as the class name.
- Java applications must include a class with a **main** method. *E.g.*,
  - `public static void main(String args[])`

# *Structure of a simple Java Program*

---

```
class HelloWorld
{
    public static void main(String [] args)
    {
        System.out.println("Hello World!");
    }
}
```

- Everything must be in a class.
- Java applications (not Applets) must have a **main()** routine in one of the classes with the signature:
  - **public static void main(String [] args)**



# *Compiling and Running the “Hello World” Program*

---

- Compile:
  - javac HelloWorld.java
- Run:
  - java HelloWorld
- Output:
  - “Hello World”

# *Comments*

---

- Java support 2 commenting styles
  - // Line Comments
  - /\* Block Comments \*/

# *Data Types*

---

- Basic types:
  - byte, boolean, char, short, int, long, float, double.
  - New types cannot be created in Java. Must create a Java class.
- Java arrays are supported as classes. *E.g.*,
  - `int[] i = new int[9]`
  - size of an array `i.length`

# *Scalar Data Types In Java*

---

```
<data type> variable_name;  
int x;
```

- Standard Java data types:
  - byte            1 byte
  - boolean        1 byte
  - char            2 byte (Unicode)
  - short           2 byte
  - int             4 byte
  - long            8 byte
  - float           4 byte
  - double          8 byte

# *Java is Strongly Typed*

---

- Java is a strongly typed language.
- Strong typing reduces many common programming errors.
- Seems inconvenient to C/C++ programmers.

# *Java is Strongly Typed (Cont'd)*

---

- Assignments must be made to compatible data types.
- To contrast, in C:
  - No difference between int, char, and “boolean”
  - Implicit type casts (automatic type promotion) between many types.

# Variables

---

- Variables may be tagged as constants (**final** keyword).
- Variables may be initialized at creation time
  - **final** variables **must** be initialized at creation time
- Objects are variables in Java and must be dynamically allocated with the **new** keyword.
  - *E.g.*, `a = new ClassA();`
- Objects are freed by assigning them to `null`, or when they go out of scope (automatic garbage collection).
  - *E.g.*, `a = null;`

# *Variables (Cont'd)*

---

```
int n = 1;
```

```
char ch = 'A';
```

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

```
Long L = new Long(100000);
```

```
boolean done = false;
```

```
final double pi = 3.14159265358979323846;
```

```
Employee joe = new Employee();
```

```
char [] a = new char[3];
```

```
Vector v = new Vector();
```



# *Pointers & References Variables*

---

- Java does not support pointers.
- All variables are passed by value except objects.
- Java classes either:
  - Reference an object (new keyword)
  - Alias an object (assign to another object)

# *Expressions*

---

- Java supports many ways to construct expressions (in precedence order):
  - ++,--    Auto increment/decrement
  - +,-        Unary plus/minus
  - \*,/        Multiplication/division
  - %          Modulus
  - +,-        Addition/subtraction

# *Examples of Expressions*

---

```
int x,y,z;
```

```
x = 0;
```

```
x++;
```

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

```
y = y % 5;
```

```
z = 9 / 5;
```

# Assignment Operators

---

- Assignment may be simple
  - $x = y$
- Or fancy with the following operators:
  - $*=, /=$
  - $\%=$
  - $+=, -=$
  - $\&=$  (bitwise AND)
  - $|=$  (bitwise OR)
  - $\^=$  (bitwise exclusive OR)

# *Examples of Assignment Operators*

---

```
int i = 5;
```

```
i += 10;           // i = 15
```

```
i %= 12;          // i = 3
```

# *Conditional Logic*

---

- Conditional logic in Java is performed with the **if** statement.
- Unlike C++ a logic expression does not evaluate to 0 (FALSE) and non-0 (TRUE), it evaluates to either **true** or **false**
- **true, false** are values of the **boolean** data type.
- Building compound conditional statements
  - **&&** (And), **||** (Or), **!** (Not), **<**, **>**, **==**, **!=**, **<=**, **>=**, etc.

# *Example of Conditional Logic*

---

```
int i = 8;
if ((i >= 0) && (i < 10))
    System.out.println(i + " is between 0 and 9");
else
    System.out.println(i + " is larger than 9 or less than 0");
```

# *Code Blocks*

---

- Java, like many other languages, allows compound code blocks to be constructed from simple statements.
- Simply enclose the block of statements between braces. *E.g.*,

```
{  
Statement1;  
Statement2;  
Statement3;  
}
```



# *Looping Constructs*

---

- Java supports three looping constructs:
  - while
  - do...while
  - for

# *Examples of Looping Constructs*

---

```
for (int i = 0; i < 10; i++)  
{  
    System.out.println(i);  
}
```

---

```
int i = 0;  
while(i < 10)  
{  
    System.out.println(i++); //prints i before  
}                          //applying i++
```

---

```
int i = 0;  
do  
{  
    System.out.println(i++);  
} while(i < 10)
```

# *Java Exception Handling*

---

- An *exception* is an object that defines an unusual or erroneous situation.
- An exception is *thrown* by a program or a runtime environment and can be *caught* and handled appropriately.
- Java supports user-defined and predefined exceptions:
  - ArithmeticException
  - ArrayIndexOutOfBoundsException
  - FileNotFoundException
  - InstantiationException

# *Java Exception Handling (Cont'd)*

---

- Exception handling allows a programmer to divide a program into a *normal execution flow* and an *exception execution flow*.
- Separation is a good idea especially since 80% of execution is in 20% of the code (normal flow).

# *Java Exception Handling (Cont'd)*

---

- If an exception is not handled the program will terminate (abnormally) and produce a message.

```
public class DivideBy0 {  
    public static void main (String[] args) {  
        System.out.println(10 / 0);  
    }  
}
```

```
Java.lang.ArithmeticException: / by zero  
at DivideBy0.main(DivdeBy0:3)
```

# *Try and Catch statements*

---

```
try {  
    statement-list1  
} catch (exception-class1 variable1) {  
    statement-list2  
} catch (exception-class2 variable2) {  
    statement-list3  
} catch ....
```

- If an exception is thrown in statement-list1, control is transferred to the appropriate (with same exception class) catch handler.
- After executing the statements in the catch clause, control transfers to the statement after the entire try statement.

# *Exception Propagation*

---

- If an exception is not caught and handled where it occurs, it propagates to the calling method.

```

class Demo {
    static public void main (String[] args) {
        Exception_Scope demo = new Exception_Scope();
        System.out.println("Program beginning");
        demo.L1();
        System.out.println("Program ending");
    }
}

class Exception_Scope {
    public void L3 () {
        System.out.println("Level3 beginning");
        System.out.println(10/0);
        System.out.println("Level3 ending");
    }
    public void L2 () {
        System.out.println("Level2 beginning");
        L3();
        System.out.println("Level2 ending");
    }
    public void L1 () {
        System.out.println("Level1 beginning");
        try { L2 ();
        } catch (ArithmeticException problem) {
            System.out.println(problem.getMessage( ));
            problem.printStackTrace ( );
        }
        System.out.println("Level1 ending");
    }
}

```

### **OUTPUT:**

```

Program beginning
Level1 beginning
Level2 beginning
Level3 beginning
/ by zero
Java.lang.ArithmeticException: / by zero
    at Exception_Scope.L3(Demo.java:18)
    at Exception_Scope.L2(Demo.java:24)
    at Exception_Scope.L1(Demo.java:31)
    at Exception_Demo.main(Demo.java:7)
Level1 ending
Program ending

```



# Throwing an Exception

---

```
import java.io.IOException;

public class Demo {
    public static void main (String[] args) throws Doh {
        Doh problem = new Doh ("Doh!");
        throw problem;
        // System.out.println("Dead code");
    }
}
```

```
class Doh extends IOException {
    Doh (String message) {
        super(message);
    }
}
```

## OUTPUT:

```
Doh: Doh!
    at Demo.main(Demo.java:4)
```

- The exception is thrown but not caught.

# *Finally* clause

---

- A *try* statement may have a *finally* clause.
- The *finally* clause defines a section of code that is executed regardless of how the *try* block is executed.

```
try {  
    statement-list1  
} catch (exception-class1 variable1) {  
    statement-list2  
} catch ...  
  
} finally {  
    statement-list3  
}
```

# *I/O*

---

- Java supports a rich set of I/O libraries:
  - Network
  - File
  - Screen (Terminal, Windows, Xterm)
  - Screen Layout
  - Printer

# *I/O (Cont'd)*

---

- For this course we only need to write to or read from the terminal screen or file.
- Use:
  - `System.out.println()`
  - `System.out.print()`
- May use '+' to concatenate

```
int i, j;  
i = 1;  
j = 7;  
System.out.println("i = " + i + " j = " + j);
```

# *I/O Example with Reading and Writing from or to the Screen*

---

```
import java.io.*;

public class X {
    public static void main(String args[])
    {
        try{
            BufferedReader dis = new BufferedReader(
                new InputStreamReader(System.in));
            System.out.println("Enter x ");
            String s = dis.readLine();
            double x= Double.valueOf(s.trim()).doubleValue();
            // trim removes leading/trailing whitespace and ASCII control chars.
            System.out.println("x = " + x);
        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

# *I/O Example with Reading and Writing from or to the File*

---

```
import java.io.*;
import java.util.*;

// Program that reads from a file with space delimited name-pairs and
// writes to another file the same name-pairs delimited by a tab.
class P
{
    public static void main(String [] args)
    {
        BufferedReader reader_d;
        int linecount = 0;
        Vector moduleNames = new Vector();

        try {

            reader_d = new BufferedReader(new FileReader(args[0]));
```

// continued on next page



# *I/O Example with Reading and Writing from or to the File (Cont'd)*

---

```
// ... continued from previous page.
```

```
while (true) {  
    String line = reader_d.readLine();  
    if (line == null) {  
        break;  
    }  
  
    StringTokenizer tok = new StringTokenizer(line, " ");  
    String module1 = tok.nextToken();  
    String module2 = tok.nextToken();  
    moduleNames.addElement(module1 + "\t" + module2);  
  
    linecount++;  
} // end while
```

# *I/O Example with Reading and Writing from or to the File (Cont'd)*

---

```
// ... continued from previous page.
```

```
catch (Exception e) {  
    e.printStackTrace();  
    System.exit(1);  
}
```

```
BufferedWriter writer_d;
```

```
try {
```

```
writer_d = new BufferedWriter(new FileWriter(args[0] + ".tab"));
```



# *I/O Example with Reading and Writing from or to the File (Cont'd)*

---

```
        for(int i = 0; i < moduleNames.size(); i++) {
            String modules = (String) moduleNames.elementAt(i);
            writer_d.write(modules, 0, modules.length()-1);
            writer_d.newLine();
        } // end for
        writer_d.close();
    } // end try
    catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

        System.out.println("Number of lines: " + linecount);
    } // end main
} // end P
```

# *Classes*

---

- A class can implement an abstract data type (ADT).
- Classes protect data with functions (methods) that safely operate on the data.
- Classes are created with the **new** keyword.
- The **new** keyword returns a reference to an object that represents an instance of the class.
- All instances of classes are allocated in a garbage-collected heap.

# *Classes (Cont'd)*

---

- In Java, everything is a class:
  - Classes you write
  - Classes supplied by the Java specification
  - Classes developed by others
  - Java extensions

```
String s = new String("This is a test")
if (s.startsWith("This") == true)
    System.out.println("Starts with This");
else
    System.out.println("Does not start with This");
```

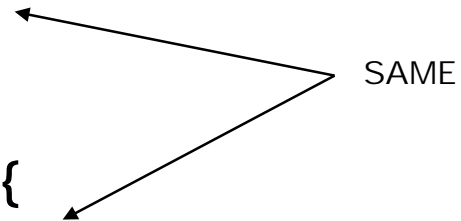
# Classes (Cont'd)

---

- All classes are derived from a single root class called **Object**.
- Every class (except **Object**) has exactly one immediate super class.
- Only single inheritance is supported.

```
Class Dot {  
    float x, y;  
}
```

```
class Dot extends Object {  
    float x, y;  
}
```



# *Casting Classes*

---

- Assume that class *B* extends class *A*.
- An instance of *B* can be used as an instance of *A* (implicit cast, widening).
- An instance of *A* can be used as an instance of *B* (explicit cast, narrowing).
- Casting between instances of sibling classes in a compile time error.

# *Methods*

---

- Methods are like functions.
- Methods are defined inside of a class definition.
- Methods are visible to all other methods defined in the class.
- Methods can be overridden by a subclass.
- Method lookup is done at run-time.
  - But lexical scoping on names.

# *Methods (Cont'd)*

---

```
class Arithmetic
{
    int add(int i, int j){
        return i + j;
    }

    int sub(int i, int j){
        return i - j;
    }
}
```

...

```
Arithmetic a = new Arithmetic();
```

```
System.out.println(a.add(1,2));
```

```
System.out.println(a.sub(5,3));
```

# *Anatomy of a Method*

---

- **Visibility identifier:**
  - **public:** Accessible anywhere by anyone.
  - **private:** Accessible only from within the class where they are declared.
  - **protected:** Accessible only to subclasses, or other classes in the same package.
  - **Unspecified:** Access is public in the class's package and private elsewhere.
- **Return type:**
  - Must specify a data type of the data that the method returns.
  - Use the **return** keyword.
  - Use **void** if the method does not return any data.



# *Anatomy of a Method (Cont'd)*

---

- Method name
- Argument list:
  - List of parameters.
  - Each parameter must have a data type.
- Java semantics (IMPORTANT)
  - *Object references are passed by reference, all other variables are passed by value.*

# *Class (Instance) Variables*

---

- In Java, we can declare variables that are global to the object.
- Define class variables at the top of the class.
- May be defined as public, private or protected.

# *Class Variables (Cont'd)*

---

```
class Add
{
    int i;    // class variable
    int j;    // class variable
              // scope public in the package, private
              // elsewhere.

    int add(){
        return i + j;
    }
}
...
```

```
Add a = new Add();
```

```
System.out.println(a.add(4,6));
```

# *Class Variables (Cont'd)*

---

- Class variables are defined for each instance of the class. Thus, each object gets its own copy.
- Class variables may be initialized. They are set to 0 or null otherwise. Local variables (inside methods) must be initialized before they can be used.
- May use the **static** keyword to make a data element common among all class instances.

# *Class Variables (Cont'd)*

---

```
class Incrementor
{
    private static int i = 0;

    void incrCount(){
        i++;
    }

    int getCount() {
        return i;
    }
}
```

```
Incrementor f1, f2;

f1 = new Incrementor();
f2 = new Incrementor();

f1.incrCount();
System.out.println(f1.getCount());
f2.incrCount();
System.out.println(f1.getCount());
```

**What is the output?**

**What is the output if *i* is not defined as static?**

# *this* and *super*

---

- Inside a method, the name *this* represents the current object.
- When a method refers to its instance variables or methods, *this* is implied.
- The *super* variable contains a reference which has the type of the current object's super class.

# Constants

---

- In C++, constants are defined using **const** or **#define**.
- In Java, it is somewhat more complicated:
  - Define a class data element as:
    - `public final <static> <datatype> <name> = <value>;`
  - Examples:
    - `public final static double PI = 3.14;`
    - `public final static int NumStudents = 60;`
- Constants may be referenced but not modified.

# *Arrays*

---

- Arrays in Java are objects.
- Arrays must be allocated.
- Arrays are passed by reference.
- Arrays are 0-based.

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

```
double [] j = new double[50];
```

```
String [] s = new String[10];
```

```
int a[][] = new int [10][3];
```

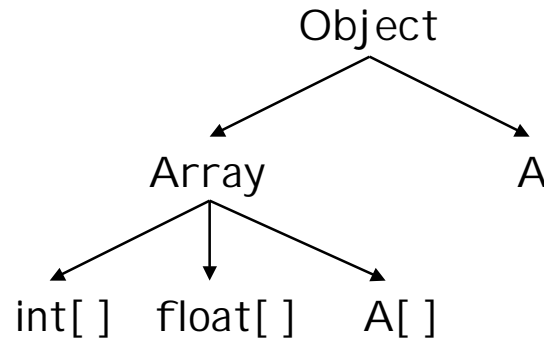
```
println(a.length);    // prints 10
```

```
println(a[0].length); // prints 3
```



# Arrays (Cont'd)

---



// You can assign an array to an Object  
// (implicit upcast)

```
Object o;  
int a[ ] = new int [10];  
o = a;
```

// You can cast an Object to an array  
// (explicit downcast)

```
Object o = new Object();  
a = (int [ ]) o;
```

# *Method Overloading*

---

- In Java, we may create methods with the same name.
  - Parameter lists must be different. Not enough to have a different return type.
- Allows for the method to be used based on the called parameters.

# *Method Overloading (Cont'd)*

---

```
class Adder
```

```
{  
    int add(int i, int j) {  
        return i + j;  
    }  
  
    double add(double i, double j) {  
        return i + j;  
    }  
}
```

```
...
```

```
int i = 4, j = 6; double l = 2.1, m = 3.39;
```

```
Adder a = new Adder();
```

```
System.out.println(a.add(i, j) + "," + a.add(l,m));
```

Software Design (Java Tutorial)

© SERG



# *Class Constructors*

---

- Each class may have one or more constructors.
- A constructor is a “method” (cannot invoke, does not return anything) that is defined with the same name as its class.
- A constructor is automatically called when an object is created using **new**.
- Constructors are valuable for initialization of class variables.

# *Class Constructors (Cont'd)*

---

- By default, the **super** class constructor with no parameters is invoked.
- If a class declares no constructors, the compiler generates the following:

```
Class MyClass extends OtherClass {  
    MyClass () {  
        super(); // automatically generated  
    }  
}
```

# *Class Constructors (Cont'd)*

---

```
class Batter extends Player {
    float slugging;
    Batter (float slugging) {
        // super is called here
        this.slugging = slugging;
    }
    Batter () {
        this((float).450); // does not call super first
    }
}
```

```
class Pitcher extends Batter {
    float era;

    Pitcher(float era) {
        super((float).300);
        this.era = era;
    }
}
```

# *Packages*

---

- A package is a loose collection of classes.
- Packages are like libraries.
- Packages may be created using the **package** keyword.
- Packages are located by the `CLASSPATH` environment variable.
- In order to use a package you use the **import** keyword.

# *Some Common Java Packages*

---

- `java.applet`
- `java.javax`
- `java.io`
- `java.lang`
- `java.net`
- `java.util`
- `java.math`
- `java.security`



# *Importing a Package is Convenient*

---

```
import java.io.*;  
DataInputStream dis = new DataInputStream();
```

or

```
java.io.DataInputStream dis = new java.io.DataInputStream();
```

# *The String Class*

---

- Used to create and manipulate strings in Java.
- Better than a null terminated sequence of characters.
- Automatically sized (automatic storage management).

# *The String Class (Cont'd)*

---

- Rich set of functions:
  - Concatenation
  - Lowercase/Uppercase data conversion
  - Sub-strings
  - Strip leading and trailing characters
  - Length
  - Data conversion to/from an array of character representation

# *The Vector Class*

---

- The vector class may be used as a dynamic array.
- The vector class can hold any Java object.

```
import java.util.Vector;
```

```
Vector v = new Vector();
```

```
v.removeAllElements();
```

```
v.addElement(new String("1"));
```

```
v.addElement(new String("2"));
```

```
v.addElement(new String("3"));
```

```
v.addElement(new String("4"));
```

```
for (int i = 0; i < v.size(); i++)
```

```
    System.out.println("Data = " + v.elementAt(i));
```

# *Data Conversion Functions*

---

- Java contains a robust set of classes to convert from one data type to another.
- There is a data encapsulation class for each scalar type:
  - **Long** for long
  - **Integer** for int
  - **Float** for float
  - **Double** for double
  - **Boolean** for boolean

# *Data Conversion Functions*

## *(Cont'd)*

---

- Typical functionality includes:
  - Convert to/from bases
  - Convert to/from a string
  - Equality checking

# *Java Events and GUI Programming*

---

*Tutorial for Beginners by SERG*

# *Create the Worker*

---

- We start by creating a worker which will actually do the work
- The example worker is computing all the multiplications of  $i$  and  $j$  from 1 to 10



```
package Hello;
import java.util.Vector;
import java.util.Enumeration;

public class claculator {
    private int prog;

    public claculator() {
        prog = 0;
    }

    public void work()
    {
        for(int i=1;i<11;i++)
        {
            for (int j=1;j<11;j++) {
                prog = i*j;
                notifyListeners(prog, false);
            }
        }
    }
}
```

# *Creating the Main Program*

---

- The main program will simulate the user interface that we will create later
- The main program is responsible to start the worker and receive the results

```
package Hello;
```

```
public class Hello_main
```

```
{
```

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

```
    {
```

```
        Hello_main hm = new Hello_main();
```

```
        System.out.println("Start");
```

```
        hm.workNoThread();
```

```
        System.out.println("End");
```

```
    }
```

```
    public Hello_main() {
```

```
    }
```

```
    public void workNoThread()
```

```
    {
```

```
        final claculator cc = new claculator();
```

```
        System.out.println("No Thread");
```

```
        cc.work();
```

```
    }
```

```
}
```

# *Why Isn't This Enough*

---

- So far we have a main program that starts a worker.
- The main program can receive no information from the worker until the worker had finished the calculation.
- This situation creates a feeling that the program is stuck and the user is likely to restart it.

# *Adding the Event*

---

- The next step is to add an event. The event object will carry the information from the worker to the main program.
- The event should be customized to fit your needs.
- Our event includes:
  - Progress: let the main program know the progress of the worker.
  - Finished flag: let the main program know when the worker is done.

```
package Hello;
```

```
public class IterationEvent {
```

```
    private int value_;
```

```
    private boolean finished_;
```

```
    public IterationEvent(int value, boolean finished) {
```

```
        value_ = value;
```

```
        finished_ = finished;
```

```
    }
```

```
    public int getValue()
```

```
    {
```

```
        return value_;
```

```
    }
```

```
    public boolean isFinished()
```

```
    {
```

```
        return finished_;
```

```
    }
```

```
}
```

# *Adding the Listener*

---

- Now that we have an event object we can build an interface of a listener.
- A listener will be any object that can receive events.
- The interface should include any function interface that will be implemented in the actual listening object.

```
package Hello;  
public interface IterationListener {  
    public void nextIteration(IterationEvent e);  
}
```



# *The Main Program Revisited*

---

- Now that we have the listener interface we should change the main program to implement the interface.
- We need to add the implementation of the interface and all the functions that we declared.

Note: don't try to run the new program before you change the worker as well.

```
package Hello;

public class Hello_main

    implements IterationListener

{
    public static void main(String args[])
    {
        Hello_main hm = new Hello_main();
        System.out.println("Start");
        hm.workNoThread();
        System.out.println("End");
    }
}
```

```
public Hello_main() {
}
```

```
public void workNoThread()
{
    final claculator cc = new claculator();
    cc.addListener(this);
    System.out.println("No Thread");
    cc.work();
    cc.removeListener(this);
}
```

```
public void nextIteration(IterationEvent e)
{
    System.out.println("From Main: "+e.getValue());

    if (e.isFinished()) {
        System.out.println("Finished");
    }
}
}
```

# *The Worker Revisited*

---

- As you might have noticed there are new functions that we need to add to the worker object:
  - `addListener(IterationListener listener)`
    - Add a new listener to the worker the listener will receive messages from the worker (you might want more than one).
  - `removeListener(IterationListener listener)`
    - Remove a listener from the worker.
  - `notifyListeners(int value, boolean finished)`
    - The function that will actually notify the listener

```

package Hello;
import java.util.Vector;
import java.util.Enumeration;

public class claculator {
    private int prog;
    private Vector listeners_ = new Vector();

    public claculator() {
        prog = 0;
    }

    public void work()
    {
        for(int i=1;i<11;i++)
        {
            for (int j=1;j<11;j++) {
                prog = i*j;
                notifyListeners(prog, false);
            }
        }
        notifyListeners(prog, true);
    }
}

```

```

public void addListener(IterationListener listener)
{
    listeners_.addElement(listener);
}

public void removeListener(IterationListener listener)
{
    listeners_.removeElement(listener);
}

private void notifyListeners(int value, boolean finished)
{
    IterationEvent e = new IterationEvent(value, finished);
    for (Enumeration listeners = listeners_.elements();
         listeners.hasMoreElements(); )
    {
        IterationListener l = (IterationListener) listeners.nextElement();
        l.nextIteration(e);
    }
}
}

```

# *I Am Confused...*

---

- Lets see what are the new things mean:
  - **addListener(IterationListener listener).**
    - This function adds a new vector element to the listeners vector (you may have as many listeners as you wish).
  - **removeListener(IterationListener listener).**
    - This function removes a listener from the listeners vector this will free the listener object once you don't need it anymore.
  - **notifyListeners(int value, boolean finished).**
    - This function is the function that sends the information to the main program.
    - The for loop runs through all the listener elements in the listeners vector and uses the **nextIteration(e)** function that is implemented in the listener it self.
    - When the worker call the **notifyListeners** function the main program runs the **nextIteration(e)** function and updates it self.

# *How About Another Listener*

---

- As was said before we can have more than one listener, lets see how we can do that.
- Remember that the listener needs to implement the interface and the functions of that interface.

```
package Hello;
import java.util.Calendar;
import java.io.*;

public class Printer
    implements IterationListener
{
    public void nextIteration(IterationEvent e) {
        Calendar Now = Calendar.getInstance();
        try {
            FileWriter fos = new FileWriter("C:\\Ron Projects\\EvensInSwting\\printer.log", true);
            fos.write("From Printer - " + Now.getTime() + " Value: " + e.getValue()+"\n");
            //System.out.println("From Printer - " + Now.getTime() + " Value: " + e.getValue());
            if (e.isFinished()) {
                fos.write("From Printer - "+Now.getTime()+" Finished\n");
                //System.out.println("Finished");
            }
            fos.close();
        } catch (java.io.IOException ex) {
            System.out.println("Opps: " + ex);
        }
    }
}
```

# *The Printer Object*

---

- The printer object is just another listener that will update a log file for the main program.
- Notice that this listener should be added from the main program but is completely independent from the main program
  - `cc.addListener(new Printer());`



# *Threading the Worker*

---

- If you want to thread the worker without changing it the next code segment that is part of the main program shows how to thread the worker from within the main program.
- We add the function `workThread()` to the main program.

```
public void workThread()
{
    final claculator cc = new claculator();
    cc.addListener(this);
    cc.addListener(new Printer());
    System.out.println("With Thread");
    Thread t = new Thread() {
        public void run() {
            cc.work();
        }
    };
    t.start();
    try {
        t.join();
    }
    catch (Exception e) {
    }
    cc.removeListener(this);
}
```

# *Real Threads*

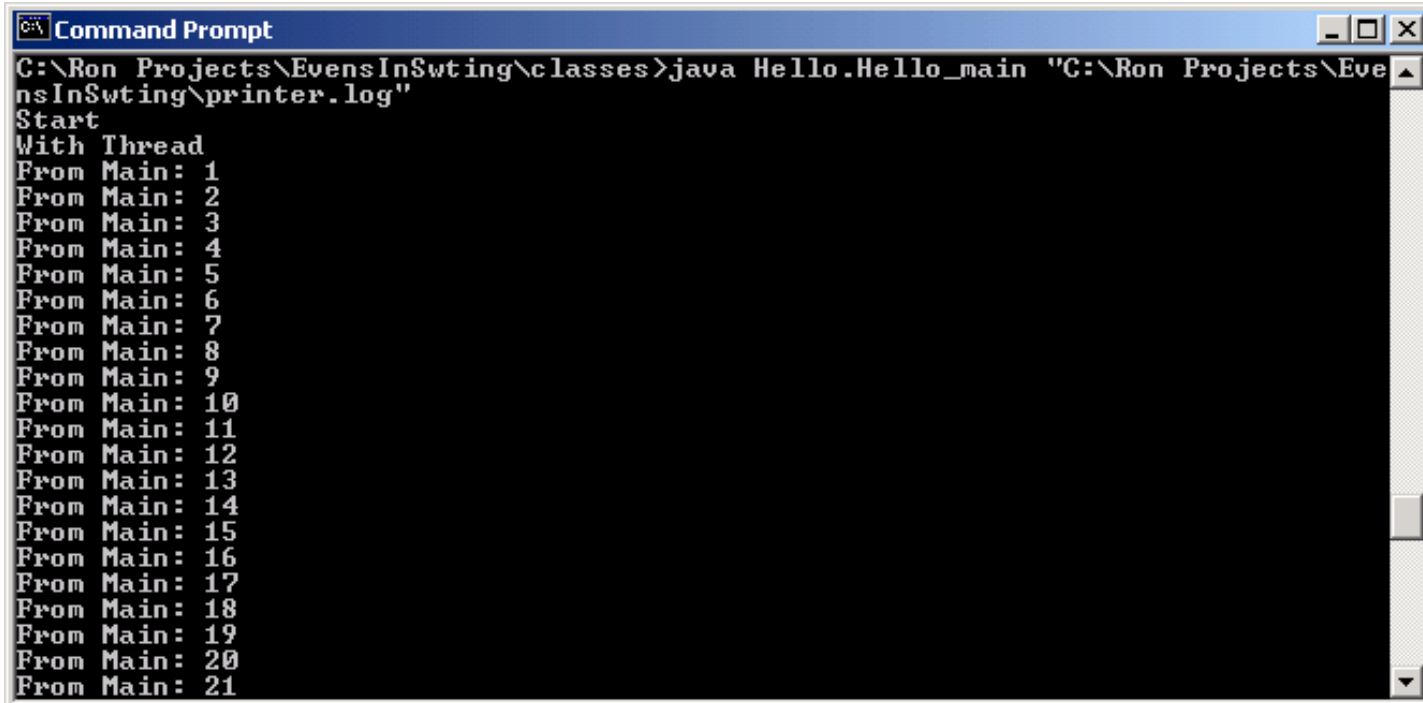
---

- If you want more information on how to build a real threaded program visit this URL:

<http://www.mcs.drexel.edu/~bmitchel/course/mcs720/java.pdf>

# *The Output From the Main Program*

---



```
C:\Ron Projects\EvensInSwting\classes>java Hello.Hello_main "C:\Ron Projects\EvensInSwting\printer.log"
Start
With Thread
From Main: 1
From Main: 2
From Main: 3
From Main: 4
From Main: 5
From Main: 6
From Main: 7
From Main: 8
From Main: 9
From Main: 10
From Main: 11
From Main: 12
From Main: 13
From Main: 14
From Main: 15
From Main: 16
From Main: 17
From Main: 18
From Main: 19
From Main: 20
From Main: 21
```

As it says the printouts are from the main program

# *The Content of the Log File*

---

*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 1*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 2*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 3*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 4*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 5*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 6*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 7*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 8*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 9*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 10*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 11*  
*From Printer - Tue Aug 22 13:18:17 EDT 2000 Value: 12...*

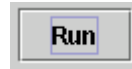
The log file was created at the same time as the screen printouts

# *Adding GUI*

---

- Lets see how we can add a nice graphical user interface to our little example.
- The UI will need to display:

– Start button:



– Progress bar:



– Current value:



– Status bar:



# *Creating Main Frame*

---

- The Main Frame is extending Swing JFrame.
- We will implement IterationListener again to get the information from the worker.

# *Main Frame Cont.*

---

- The following code segments show how to declare the frame:
  - Create the content pane
  - Create the layout
  - Create different fields (buttons, text, progressbar etc.)



```

package Hello;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.io.File;
import javax.swing.UIManager;

public class Hello_Frame1
    extends JFrame implements IterationListener
{
    JPanel contentPane;
    JLabel statusBar = new JLabel();
    BorderLayout borderLayout1 = new
BorderLayout();
    JPanel jPanel1 = new JPanel();
    GridBagLayout gridBagLayout1 = new
GridBagLayout();
    JLabel jLabel1 = new JLabel();
    JButton jButtonRun = new JButton();
    JLabel jLabelCurrentValue = new JLabel();
    JProgressBar jProgressBar = new
JProgressBar(0,100);
    JLabel jLabelProgress = new JLabel();
    JLabel jLabel2 = new JLabel();

```

```

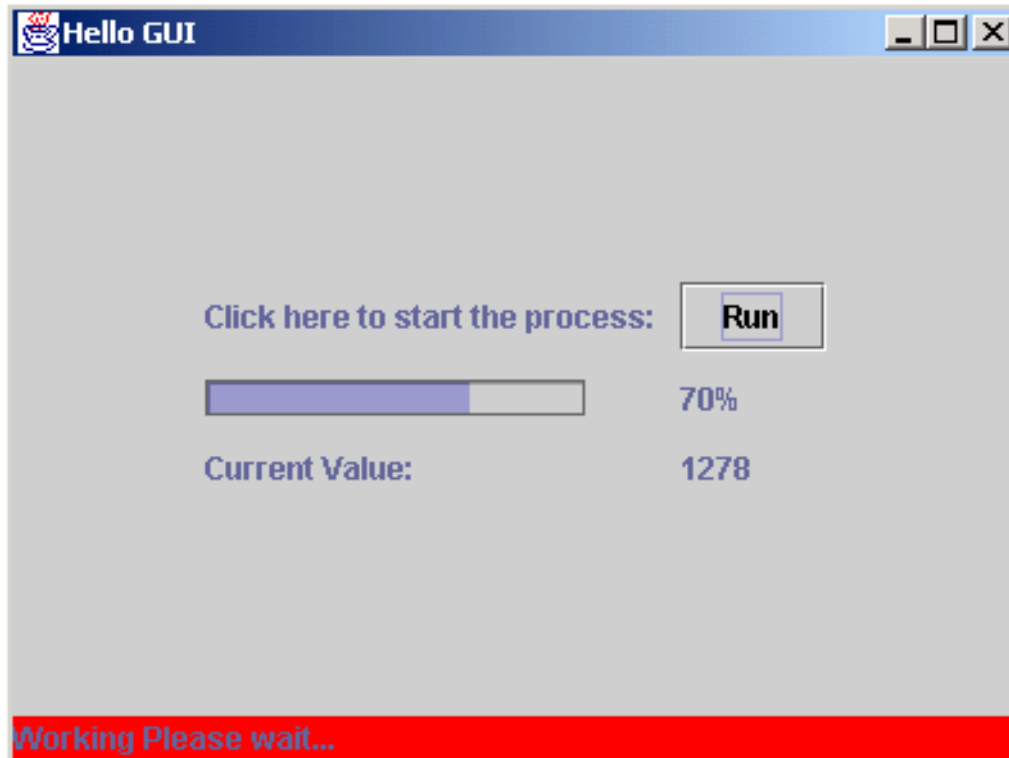
public Hello_Frame1() {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("Hello GUI ");
    statusBar.setOpaque(true);
    statusBar.setText("Ready");
    jPanel1.setLayout(gridBagLayout1);
    jLabel1.setText("Click here to start the process:");
    jLabelCurrentValue.setText("Not Running");
    jButtonRun.setText("Run");
    jProgressBar.setValue(0);
    jLabelProgress.setText(((String)
        (new Integer(jProgressBar.getValue())).toString()+"%");
    jLabel2.setText("Current Value:");

```

```
contentPane.add(statusBar, BorderLayout.SOUTH);
contentPane.add(jPanel1, BorderLayout.CENTER);
jPanel1.add(jLabel1, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
jPanel1.add(jButtonRun, new GridBagConstraints(1, 0, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
jPanel1.add(jProgressBar, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
jPanel1.add(jLabelProgress, new GridBagConstraints(1, 1, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
jPanel1.add(jLabel2, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
jPanel1.add(jLabelCurrentValue, new GridBagConstraints(1, 2, 1, 1, 0.0, 0.0
    ,GridBagConstraints.WEST, GridBagConstraints.NONE, new Insets(5, 5, 5, 5), 0, 0));
}
```

# The Main Frame With the Controls

---



# *Button Functionality*

---

- We have only one button and we need to add a listener to it.
- We can add an `addActionListener` and implement it later.
- The following code segments show how to add the listener, create the class and implement the function.

# *Button Functionality Cont.*

---

- After the declaration of the listener we implement the actionPerformed function in a new function:
  - jButtonRun\_actionPerformed.
- In this function we will run the calculator in a similar way to the non-GUI way.
- We add the Frame as the listener and implement how it will listen later.

```
jButtonRun.addActionListener(new java.awt.event.ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        jButtonRun_actionPerformed(e);
    }
});
```

```
void jButtonRun_actionPerformed(ActionEvent e)
{
    change_status(Color.red, "Working Please wait...");
    final claculator cc = new claculator();
    final I terationListener listener = this;

    //cc.addListener(new Printer());
    Thread t = new Thread() {
        public void run() {
            cc.addListener(listener);
            cc.work();
            cc.removeListener(listener);
        }
    };
    t.start();
}
```

# *GUI nextIteration Implementation*

---

- The new function will need to update the GUI with the status of the worker:
  - Change the status bar to a working mode.
  - Update:
    - Current value field.
    - Status Bar.
    - Complete percentage.
  - When done open the ‘Done’ dialog box and reset everything to the starting state.

```
public void nextIteration(I IterationEvent e)
{
    //System.out.println(e.getProgress());
    progressBar.setValue(e.getProgress());
    jLabelProgress.setText((String) (new Integer(progressBar.getValue())).toString()+"%");
    jLabelCurrentValue.setText((String) (new Integer(e.getValue()).toString() ));

    if (e.isFinished()) {
        change_status(Color.getColor("204,204,204"), "Ready");
        //Start the finished dialogBox
        Hello_FrameAbout dlg = new Hello_FrameAbout(this);
        Dimension dlgSize = dlg.getPreferredSize();
        Dimension frmSize = getSize();
        Point loc = getLocation();
        dlg.setLocation((frmSize.width - dlgSize.width) / 2 + loc.x, (frmSize.height - dlgSize.height) / 2 + loc.y);
        dlg.setModal(true);
        dlg.show();
        progressBar.setValue(0);
        jLabelProgress.setText("0%");
        jLabelCurrentValue.setText("Not Running");
    }
}
```



# Summary

---

- I hope you gained some knowledge of how to create and manage Swing applications and events.
- Source code the complete example as well as the compiled version is available to download.
- Usage:
  - `Java -classpath Hello.jar Hello.Hello_main filename`
  - `Java -classpath Hello.jar Hello.Hello_GUI`

# *References*

---

- [Mitchell99] B. S. Mitchell class lecture notes.
- [Sun96] The Java Language Tutorial, Sun Microsystems.
- [JDK99] The Online Java Tutorial  
<http://java.sun.com/docs/books/tutorial/index.html>