**CS320 JAVA**

**Introduction**

Java is an Object-Oriented language. It was designed to allow people to send programs over a network to appliances like VCRs and microwave ovens. It was developed by a research team working for Sun Microcomputers. Sun, later, realized that it could be used on the WWW and it became famous overnight.

Java can be used to create normal programs that you run by typing commands. These are called *applications*. Java is mostly used to describe programs that can be sent across the Internet and executed *safely* on a different computer. These are called *Applets*.

**Running Java**

Java code is put into a file with suffix or extension `.java`. Each file declares one or more *classes*. A file called *Foo*.java should declare the class *Foo*. **Hint:** Notice where the UPPER and lowerCase letters are in Java source code and in file names. File names and Java code must match perfectly and this is toTaLlY SensItIve to case in its filenames on UNIX. HelloWorld is not helloworld.

We use the Sun Java Development Kits (JDKs) at CSUSB and so a Java file is compiled by

```
javac filename
```

After compilation each *class* (in the above file) is placed in a file with name

```
ClassName.class
```

These *class* files are in a special binary code called *bytecode*. They are either published as *applet*s and accesses via the WWW or interpreted by the 'java' command:

```
java ClassName
```

There must be no extension above. The ClassName.class file must be in the right place (see packages and directories below).

Documentation for the classes in a file is generated by

```
javadoc filename
```

and is put into files with names like this

```
ClassName.html
```

The *javadoc* program makes it easy to generate documentation on the WWW for people to read.

**Java Documentation**

Java is still evolving. When Oracle bought Sun it got Java as part of the deal. The WWW is an important tool for keeping up with changes. Sun used to keep up-to-date documentation on the World Wide Web. You should learn to search and use this rather than relying on printouts and text books until we have a standard. See http://www.csci.csusb.edu/dick/samples/java.html for pointers to online information on Java

**Java at CSCI.CSUSB.EDU**

Publishing

Browsers can not access files on private machines. Make sure you copy all compiled classes into your /web directory. For a Java applet to be used via the web it must be referred to in an HTML page.

Quick Compilation, Document and Test

My Q command will compile a java program, generate documentation, and invite you to run a class:

```
~dick/bin/Q ProgramName.java
```

Java in the Lab

I have noticed in our laboratories that a good applet can go badly wrong if viewed on the wrong machines with the wrong browser. The systems admin recommends "Firefox" in JBH358. Applications with output to a graphic "Frame" report a couple of spurious errors before working properly.

Compilation and interpreting Java needs Sun's Java 2 Software Development Kit. We have several. Execute this UNIX command

```
ls -ld  /share/j2sdk*
```

to get a list. Pick the last one of the listed directories , plus "/bin" as prefixes to the java, javac, ... commands. You can add the directory to your PATH (before /share/bin, if any) to let you use the abbreviated commands. For example if the last /share/j2sdk* is

```
/share/j2sdk1.4.2_07
```

then add

```
/share/j2sdk1.4.2_07/bin
```

to your PATH or type

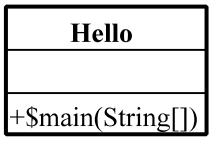
```
/share/j2sdk1.4.2_07/bin/javac filename
/share/j2sdk1.4.2_07/bin/java ClassName
```

**Java Versions**

Java promised to "compile once, run anywhere". The existence of different versions breaks this promise. A user's browser may only recognize classes compiled with older development kits. It pays to avoid the bleeding edge!

**Java Applications**

An *application* is a compiled Java class that can be run on a server or at a terminal. It can come from any ***public*** *class* that has a *main* function declared as follows:

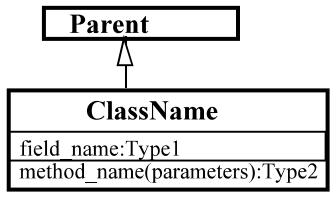



```
public class Hello {
 public static void main(String argv[]){
  System.out.println("Hello, World!");
 }
}
```

(you can change the *italic* bits etc...)

**Java - the language**

Java borrows most of its syntax from an early version of C++. It is not as writeable as C or C++ but it is instantly readable by C and C++ programmers. For example:




in Java looks like this:

**class** *ClassName* **extends** *Parent***{**
    *Type1* *field_name*;
    *Type2* *method_name* ( *parameters* )...
**}**

Java classes form a hierarchy. Each class *extends* precisely one other class. All classes ultimately extend *Object*. Unlike C++, no class can extend more than one other class. If the '**extends** *Parent*' is omitted then the parent is *Object*.

A class contains a set of *field* and *method* declarations.
A *field* declaration defines variable or constant data:

***modifiers*** *type name***;**

or ***modifiers*** *type name*=*value***;**

A *method* declaration defines a function:

***modifiers*** *type name***(***arguments***){**
        *body*
**}**

Functions can declare *local variables*.

Modifiers specify scope and other things:

**public**, **private**, **static**, **final**, **abstract**, ...

These are defined in the glossary later in this document.

**Java Data Types**

All data has defined default initial values. All data have defined precision and size. Java has a dozen primitive data types:

**boolean**, **byte**, **int**, **char**, **long**, **double**,...

A **byte** is an 8-bit ASCII character. A **char** is a 16bit Unicode character.

Java distinguishes primitive data types (like int and double) from classes. *Classes* can define new data types as collections of fields with operations. *Interfaces* are abstract data types. All *arrays* are classes. There are many large libraries of predefined classes and interfaces.

**Java Scoping**

Prior to Java 1.1, Java used simple Object-Oriented scoping. This means searching the local function first, then its class, and then the classes from that the class extended, and so on. The main scopes are: world-wide(**public**), package(*default*), class(**private**), and method/function(local).

Variables and parameters are declared inside methods. Their scope is the whole function. Local variables can *shadow* fields in a class but not formal parameters. In Java `**this**.*v*` is a field in this class even if *v* is also a formal parameter or if it is declared locally as a variable. Declarations of classes and functions could not occur inside functions in Java 1.0.

Java 1.1...2 complicates scoping by allowing classes to be defined inside classes and methods. The search for the right meaning for an identifier can follow the inheritance hierarchy (OO scoping) or the lexical containment hierarchy (Static scoping). Java 1.1 forces you to use the dot notation to resolve ambiguities.

**Parameter passing**

Primitive data like ***char, int*** and ***double*** are passed by value. Objects (instances of classes and arrays) are passed by reference.

**Predefined Classes in Java**

Like Smalltalk and C++, Java comes with a large and growing library of classes including: Object, String, Vector, Number, Integer, Double, System, Applet, Exception, Error, Frame, Graphics, Menu, Button, Enumeration, Thread .... They include a machine and platform independent toolkit for a graphical user interface(GUI) called the *AWT*(Abstract Windowing Toolkit). Java 2 have added more classes (Swing!) to make GUIs easier to program (and slower to run).

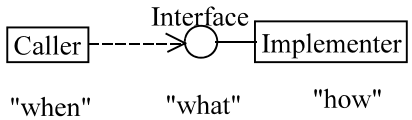
**Java Interfaces**

An *interface* is a list of *abstract methods*(function prototypes in C/C++ terms). If *I* is an interface that lists method *m* and variable *v* is declared to be of type *I* then the compiler assumes *f* can apply to *v*:

*v* . *f*(...).

The compiler makes sure that *v* can only refer to objects that come from classes that *implement I*. The Java interpreter picks the *f* from the class that *v*'s object refers to at that time.

A class can implement any number of *interfaces*. Interfaces provide a "plug and socket" to connect pieces of code. Calling component knows *when* to *use* a function, the Interface defines *what* functions can be called, and implementer components know *how* to do the function's job.



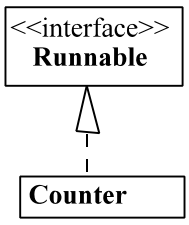

An interface usually has a name ending in "able". The code should contain enough documentation so that a programmer can use or implement the interface after reading the documentation.

An interface used for Multithreaded Code.

```
public  interface Runnable
{   public abstract void run();
}
```

A class *implements* interfaces by *supplying the bodies* of the methods in the interface:

```
public  class Counter implements Runnable{
  private int me;
  public Counter( int i) {me=i;}
  public void run()
 { while( -- me > 0)pause(); }
  public int getMe()
 { return me ;}
  private void pause(){ ... }
}
```




Usage:

```
  Counter countdown = new
Counter(100);
```

Each implementation of an interface can be different because an interface defines only the caller's point of view.

**Java Applets**

Applets are programs that can be sent over a network to be executed on another machine. They are, therefore, not allowed to access information on the machine they are sent to and will fail if they try. An *applet* is called from a HTML page, or another method, or another application. Browsers or Sun's AppletViewer use these HTML pages.

An *applet* must be a public class that *extends* the **class *Applet*** and defines the methods: ***init()*** and ***paint()***. Here is an example:

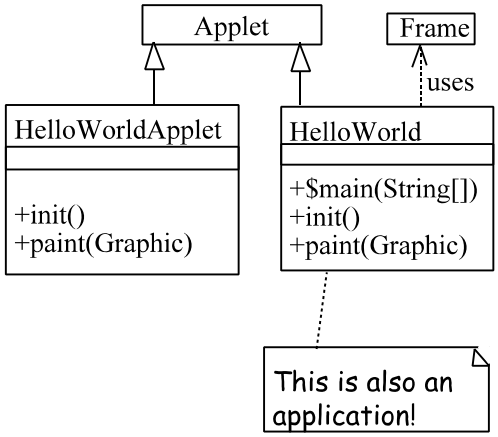
```
public class HelloWorldApplet extends Applet {
   public void init() { setSize(150,25);}
   public void paint(Graphics g) {
      g.drawString("Hello world!", 50, 25);
   }//paint
}//HelloWorld
```

http://web.csusb.edu/public/faculty/dick/Hello.html
http://web.csusb.edu/public/faculty/dick/Hello.java

The *HelloWorldApplet* class has to be inside a file called *HelloWorldApplet.java*. It is compiled by *javac* in the normal way. *Javadoc* generates documentation in HelloWorldApplet.html. But it can **not** be interpreted by the *java* command! It must be accessed from an HTML page with an <APPLET> tag.

It is a pain to create a page to test each Applet!. Debugging with a browser is even more painful. Luckily a class can be *both an applet and application*. All it needs is a *main* method that creates the Graphic that the Applet paints. This is then compiled, documented and run in the normal way. Here is an example of an application/applet on my web site:
http://web.csusb.edu/public/faculty/dick/HelloWorld.java

There is a simplified version on the next page:



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

public class HelloWorld extends Applet {

/** init is called when the applet is loaded. */
   public void init(){ //do nothing
   }//init

/** paint is called whenever the applet
    needs to be painted or repainted.
    It is given a Graphics object to paint in.
*/
   public void paint(Graphics g) {
      g.setColor(Color.pink);
      g.drawString("Hello world!", 50, 25);
   }//paint

/**main is called by an application. This one makes a
window in which a HelloWorld object is put.
*/
   public static void main(String args[]) {
      HelloWorld hi = new HelloWorld();
         hi.init();//initialize applet.
         hi.start();//start applet.
       //The applet hi is ready
      Frame w = new Frame("HelloWorld");
                                // window for hi
         w.add("Center", hi);//put hi in middle of w
         w.setSize(300, 300);// make w big enough
         w.setVisible(True);//let the user see w.
   }//main
}//HelloWorld
```

Note that on some Linux systems a couple run time errors are thrown when the new Frame is created. You can ignore them.

There is a more exciting version on:
http://web.csusb.edu/public/faculty/dick/java.test.html

**Java Packages**

A *package* is a collection of compiled classes in files *in a common directory* somewhere on the WWW. Each file must also state which package it belongs in:

**package** *package_name;*

PAckage scope: the default is to share information only between classes in a single **package**. Java files can refer to classes in packages and on other servers as long as these are named in an "**import**" statement An **import** statement can access any **public class** anywhere on the WWW or any non-**private** class in the *same* package.

**Java Idioms**

Certain patterns of code a common in Java. For example, local variables are declared close to their first use. Here are some more patterns:

We often do things to objects like this:

variable . method ( arguments );
//method of Parent or Child type.

The above can add or delete component parts etc. A call can also return an object

variable = ...object.method (parameters)...;

the above uses a method to get data from an *object*. It may change it. It then uses the return value to calculate a new value for the variable. This idiom can be extended into a sequence of method calls: ....

o.m1(p1).m2(p2).m2(p3). ...;

In the above *o* is an object that reacts to *m1(p1)* and returns another object that reacts to *m2(p2)* which, in turn, reacts to *m3(p3)*, and so on.

We often create new variables and objects like this:

Parent variable = **new** Child( arguments );

The above creates a variable that can refer to objects that fit the *Parent* type. The `**new** *Child*` is a *constructor* that creates a new object of type *Child*. This compiles as long as the *Child* class is extended from and/or implements the *Parent* class (directly or indirectly). You can delay the initialization of the object and change the object that the variable refers to.

You must take extra care with *arrays*: *Declaring an array does not create the storage or the items in the storage*. It takes several steps to do this, and they are all necessary:

```
Parent variable[]; // (1)
  variable = new Parent [2];  // (2)+(3)
  variable[0]= firstChild;
  variable[1]=secondChild; // ( 4)
   ...
```

(1)The variable is declared and is **NULL**, (2) the array is created and (3) the variable is attached to it. (4) The items in the array are defined. If *Parent* is a **class** or **interface** the array elements are NULL until they are attached to objects. The objects can be of any type or class derived from the Parent type. Omitting any step leads to a runtime error. Be careful!

**Binding Calls to Methods**

In Java 2 nearly all calls use late binding. The class of the object determines the code that is executed rather than the location of the code. Only if a function was "final" can you get static binding.

**Java Hints**

* Don't rush to use the latest version of Java unless you know that *all* your colleagues, clients, testers, and users have compatible browsers. Recompiling a class with the latest JDK can stop older browsers loading it!

* When things do not work, check the cAsE of all *file* names vs class identifiers vs names in HTML code. Also check names of directories with package names.

* Make sure you publish everything to the right directories: HTML pages, GIFs, *.class files, and all the imported classes as well.

* Read the official documentation of the library classes for the version of Java that you are using.

* Use Javadoc and Doc-Comments and read the pages they produce. Publish these along with your public classes.

* When something works in an application but not in an applet. (1) Is it **public**? (2) Is it trying to do something that is insecure like (a) finding out about the client's computer, (b)reading it's files, or (c) shutting down the Browser!.

* If an applications can not find a class, check the CLASSPATH variable in your shell!

* In Netscape 4.0 and later click "Reload" to get a new version of the page and Shift+"Reload" to get a new version of a newly compiled class.

* The older Netscape will not download new versions of applet class files until you terminate them and re-execute them. You can often use Sun's Appletviewer or program a test application instead. You can run a Netscape on a test page in foreground and terminate it when done. On Orion 'Q' will compile a file called *P*.java and automatically execute

netscape test.*P*.html

for you if it exists.

* Open the Java Console when testing an Applet with a browser.

* If it runs and crashes with a exception referring to Null ---check all (1) references, (2)arrays, and (3)each elements in arrays. The default initial value for objects in arrays is Null.

* It is a good thing that Java has Threads! It is difficult to write well-behaved applets that are not multithreaded. A single threaded Applet can not be stopped by its browser when you change pages. It must co-operate and give the browser an opportunity to stop it. Similarly, a process generating a graphic must suspend and let the browser display the results!

Write a test page/application for each class FIRST. Then work on the new class.

* It pays to **think** before you code:

- Diagrams help: UML classes, interactions, states....
- Reuse and extend existing classes. Don't reinvent the wheel.
- Plan for Reuse.
- Use known design patterns.

**Java Syntax**
The following is a quick overview for C and C++ programmers. It is based on "The Java Language Specification(1.0Alpha3)" and several other books. For more information,
.See http://www.csci.csusb.edu/dick/samples/java.html

**. Note**
This is a simplified grammar for a Java compilation unit. It uses my XBNF Extended BNF Notation where "|" indicates "or", "(...)" indicates priority. O(_) stands for 0 or 1 occurrences, N(_) for 1 or more occurrence, L(_) for a comma separated list, and #(_) for 0 or more occurrences. Quoted text "+" signifies literal terminals. It also uses:
*Set*(*X*)::="**{**" #( *X* ) "**}**".
For more on XBNF see http://www.csci.csusb.edu/dick/maths/intro_ebnf.html

**. Java extends C**
Much of the Java Syntax is based on the syntax of C and/or C++:
raw_C::=http://www.csci.csusb.edu/dick/samples/c.syntax.html.

The following borrows the structures from the syntax of C but changes some names to those used in the Java syntax.
C::=*raw_C*(expression=>*Expression*, statement=>*Statement*, ...).

**. Lexemes**
The following are defined as in C
*Identifier*::=*C.identifier*.
*Number*::=*C.integer_constant* | *C.float_constant*.
*String*::=*C.string_constant*.
*Character*::=*C.character_constant*.

*comment*::= *C.Comment* | *C++.Comment* | *Doc_Comment*.

| | |
|---|---|
| `// text` | All characters from // to the end of the line are ignored. |
| `/* text */` | All characters from /* to */ are ignored. |
| `/** text */` | These comments are treated specially when they occur immediately before any declaration. They should not be used any where else. The enclosed text is put in automatically generated documentation. |

*Doc_Comment*::=`documentation comment`::= "**/**" `*documentation*` "***/**".

**. Compilation Units**
A Java program consists of one or more compilation units that may be held in separately compiled files.
*Java_Program*::=N(*Compilation_Unit*).
A compilation unit can identify its package, import any number of other packages, classes, or interfaces, and can declare any number of classes and interfaces.
*Compilation_Unit*::= O(*Package_Statement*) #(*Import_Statement*) #(*Type_Declaration*).
Each file can place its classes into at most **one** package. It is named at the start of the file and must also be the name of the file's directory!
*Package_Statement*::= "**package**" *Package_Name* "**;**".
Packages are a collection of comparatively unrelated classes and interfaces that are stored in *`a single directory`*. If the package name is omitted then the file is in a default un-named package. Packages are accessed by using the **import** statement and the dot notation:
**import** mystuff.Fun;
would import the content of a class in file `Fun.class` in directory `mystuff`. The code for `Fun` must be in a file which starts with " **package** mystuff " in directory called "mystuff".

*Import_Statement*::= "**import**" (*Package_Name* "**.***" | O(*Package_Name* "**.**")(*Class_Name* |*Interface_Name* ) ) "**;**".

. **Declarations**
*Type_Name*::= *Class_Name* | *Interface_Name*.
*Type_Declaration*::= *Class_Declaration* | *Interface_Declaration* | "**;**".

*Class_Declaration*::= O(*Doc_Comment*) *Possible_Modifiers* "**class**" *Identifier* O(*Extends*) O(*Implements*) *Set* (*Internal_Declaration*).

*Extends*::="**extends**" *Class_Name*. -- the default is to extend the predefined class called Object. Objects are responsible for garbage collection, synchronization, and knowing how to display themselves as a string.

*Implements*::="**implements**" L(*Interface_Name*). -- A class implements an interface if it provides bodies for the methods listed in the interface.

*Interface_Declaration*::= O(*Doc_Comment*) *Possible_Modifiers* "**interface**" *Identifier* O(*Extends_interfaces*) *Set*( *Abstract_Method_declaration*).

*Extends_interfaces*::= "**extends**" L(*Interface_Name*) .

*Internal_Declaration*::= O(*Doc_Comment*) ( *Method_Declaration* | *Constructor_Declaration* | *Field_Declaration* ) | *Static_Initializer* | "**;**". -- In Java 1.1 There can also be inner class declarations.

*Static_Initializer*::= "**static**" *Block*. -- a block of statements that is invoked when a class is loaded.

*Method_Declaration*::= *Abstract_Method_Declaration* | *Concrete_Method_Declaration*.
*Concrete_Method_Declaration* ::= *Method_header Block*. -- a method with a defined body.
*Abstract_Method_Declaration*::= O("**abstract**") *Method_header* "**;**" . -- The body defined in child class.

*Method_header*::=*Possible_Modifiers Returned_Type Identifier Parameter*s *Possible_Array_Indicators*.

*Parameter*s ::="**(**" O( L(*Parameter*)) ")".

*Returned_Type*:= "**void"** | *Type*.

*Constructor_Declaration*::= *Possible_Modifiers Class_Identifier Parameter*s *Block*. -- this block can start with a special call to the parent classes constructor "super(...);" with suitable arguments.

*Parameter*::= *Type_Specifier Identifier Possible_Array_Indicators*.

*Class_Identifier*::= *Identifier* & `*the name of the class of object being constructed*`.

*Field_Declaration*::= *Possible_Modifiers Type* L( *Field_Declarator*) "**;**". -- a field is an item of data that can be a variable(if not **final**) or a constant (if **final**). The **static** modifier makes the data belong to the class and shared by all objects in the class. Non-**static** identifiers are attached to one object only.
*Field_Declarator*::= *Identifier Possible_Array_Indicators* O("**=**" *Field_Initializer*).
*Field_Initializer* ::= *Expression* | "**{**" O( L(*Field_Initializer*) O("**,**") ) "**}**".

**. Statements**
Java Statements can only appear inside constructors, static initializers and methods. The rules are like those of C Statements -- however the goto-statement has been removed (even if "goto" is still a reserved word ) and three new statements have been added: **try**, **throw**, and **synchronized**. The syntax of **break** and **continue** have been changed to include a label of the statement that is broken or continued.

*Statement*::=*C_based_statement* | *Non_C_Statement*.
*C_based_Statement* ::= *Variable_Declaration* |
*Expression* "**;**" |
*Block* |
"**if**" "**(**" *Expression* "**)**" *Statement* O("**else**" *Statement*) |
"**while**" "**(**" *Expression* "**)**" *Statement* |
"**for**" "**(**" *For_initializer* "**;**" *Expression* "**;**" *Expression* "**)**" *Statement* |
"**do**" Statement "**while**" "**(**" *Expression* "**)**" "**;**" |
"**switch**" "**(**" *Expression* "**)**" *Block* |
"**return**" O(*Expression*) "**;**" |
"**case**" *Expression* "**:**" |
"**default**" "**:**" |
*Identifier* "**:**" *Statement* |
"**break**" O(*Identifier*) "**;**" |
"**continue**" O(*Identifier*) "**;**" |
"**;**".
*Block*::="**{**" N(*Statement*) "**}**".
*For_initializer* ::= *Expression* | *Type Identifier* "=" *Expression*, -- I think. Note that, unlike C++, a variable declared in a for-statement has the whole function as its scope and it may not be declared elsewhere... (This may change?)

*Variable_declaration*::= O( "**final**" ) *Type Identifier* O( "=" *Expression* )"**;**" , -- a variable can only be declared once per function. Final variables are constants.

*Non_C_Statement*::= *Try_statement*
| *Synchronized_statement*
| *Throw_statement*
| *Local_class_declaration. -- only in Java 1.1 and later.*

*Try_statement* ::="**try**" *Block* #("**catch**" "**(**" *Parameter* "**)**" *Block*) O("**finally**" *Block*) .

*Throw_statement.* ::= "**throw**" *Expression* ";" .

*Synchronized_statement* ::= "**synchronized**" "**(**" *Expression* "**)**" *Block* .

*Local_class_declaration*::=*Class_Declaration.* -- Added in Java 1.1also see *Object_creation_expression* below.

**. Compound Names**
*Package_Name*::= *Identifier* | *Package_Name* "**.**" *Identifier*.
*Class_Name*::= *Identifier* | *Package_Name* "**.**" *Identifier*.
*Interface_Name*::= *Identifier* | *Package_Name* "**.**" *Identifier*.

Similarly with variable, field, and method names.

**. Expressions**
Expressions follow rules very like those of C but with some changes to the operators. Here is an abbreviated description of the *abstract syntax* of an *Expression*. An abstract syntax ignores the priorities of operators.

E::= *Literal* | *E Infix E* | *Prefix E* | *E Postfix* | *Conditional_E* | *Other_E.*

*Infix*::= "+" | "-" | "*" | "/" | "%" | "^" | "&" | "|" | "&&" | "||" | "<<" | ">>" | ">>>" | "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "^=" | "&=" | "|=" | "<<=" | ">>=" | ">>>=" | "<" | ">" | "<=" | ">=" | "==" | "!=" | "." | ",".
*Prefix*::= "++" | "--" | "-" | "~" | "!".

*Postfix*::= "++" | "--".
*Conditional_E*::=*E* "**?**" *E* "**:**" *E*.
*Other_E*::= *array* | *cast* | *call* | *parenthesized*.
*array*::=*E Number_of_subscripts* .
*Number_of_subscripts* ::= N( "**[**" *E* "**]**" ).
*parenthesized*::="**(**" *E* "**)**" .
*cast*::= "**(**" *Type* "**)**" *E* .
*call*::= *method_name* "**(**" *Optional_List_of_Expressions* "**)**".
*Optional_List_of_Expressions* ::= O( L( *E* ) ).

*Literal*::= *Boolean_literal* | *Object_literal* | Number | String | Character.
*Boolean_literal*::="**true**" | "**false**".
*Object_literal*::="**null**" | "**super**" | "**this**". -- see Glossary

*Non_C_Expression*::=*Run_time_type_test_expression* | *Object_method_call* | *Object_creation_expression*,

*Run_time_type_test_expression*::=*E* "**instanceof**" *Type_Name* ,

*Object_method_call*::=*E* "**.**" *method_name* "**(**" *Optional_List_of_Expressions* "**)**".
-- *E* **.** *m*(*a*) will compile iff (1) *E* returns an object of type *T*, (2) *m* must be a method in a class that directly or indirectly extends and/or implements *T*, (3) The number and types of the actual arguments *a* must match those of the formal parameters of *m*, and (4) the rules of scope and information hiding for *m* are followed.
-- The actual method call is determined at run time by using the class of the object returned by *E*.

*Object_creation_expression*::= *New_Instance_of_Class* | *New_Array_Instance* |
*New_Object_of_an_Anonymous_New_class.*

*New_Instance_of_Class*::= "**new**" *Class_Name* "(" *Optional_List_of_Expressions* ")" | "**new**" "**(**" *E* "**)**"..
*New_Array_Instance*::= "**new**" *Type_Specifier Number_of_subscripts Possible_Array_Indicators.*

Added in Java 1.1 -- able to create an object that is an instance of an extended class or interface.
*New_Object_of_an_Anonymous_new_class*="**new**" ( *Anonymous_Class* | *Anonymous_Interface*).
*Anonymous_Class* ::= *Class_Name* "("*Optional_List_of_Expressions* ")" *Set* (*Internal_Declaration*).
*Anonymous_Interface*= I*nterface_Name* "()" *Set*( *Abstract_Method_declaration*).

**. Types**
A type is either elementary, or a class or interface, and can indicate an array as well:
*Type*::= *Type_Name Possible_Array_Indicators*.
*Type_Name*::= *Elementary_data_type_name* | *Class_Name* | *Interface_Name.*

*Possible_Array_Indicators*::=#("**[**" "**]**").

*Elementary_data_type_name*::="**boolean**" | "**byte**" | "**char**" | "**short**" | "**int**" | "**float**" | "**long**" | "**double**".

**. Modifiers**
*Possible_Modifiers*::=#(*Modifier*), -- modifiers should not be repeated and certain pairs of modifiers can not both be used at one time (for example **public** and **private**).

*Modifier*::= "**public**" | "**private**" | "**protected**" | "**static**" | "**final**" | "**native**" | "**synchronized**" | "**abstract**" | "**threadsafe**" | "**transient**".
**. Glossary**

http://www.csci.csusb.edu/dick/samples/java.glossary.html

**. Literals**
**null**::Object=`nonexistent object`. -- initial value for all variables that can refer to object but hasn't been attached to one yet. Also a handy value to indicate that some object (like a Thead) is dead and gone.
**super**::=`name of object that *this* extends`.
**this**::=`name of object currently invoked`.

**. Ideas**
deprecate::=`to make a feature obsolete so that it can be removed it in a future version`.

extend::gloss=`Add, define, or redefine fields and methods in an pre-existing class or interface`. If T1 extends T2 and T2 extends T3 then T1 extends T3 indirectly, and so on.

field::=`an attribute or data item in a class`.

implement::gloss=`To provide detailed code that satisfies a particular interface`.

interface::gloss=`A description of *how to use* objects in a set of classes, that does not define *how they work*`.

method::=`operation, function, ...`.

overloading::=`One name with different meanings for different types of object`.

package::=`unrelated collection of classes and interfaces. A file that starts with a *package_specifier* P generates classes C in directory P with a file C.class that can be imported as P.* or P.C`.

**. Modifiers**
default_modifier::=`In the absence of any modifiers fields are associated with objects and are visible in all classes in a package, and are inherited by a subclass in the same package only`.

public::=`A public field or method can be used in any other class, public classes can be imported and used any where on the WWW`. -- note that objects are only usable over the WWW if they are public - hence all applets are public classes.

private::=`A private field or method can only be used its own class.`. Note. To get read only fields have a private field and public accessor method that returns the value.

protected::=` Protected variables and functions are inherited by subclasses and are accessible within a package....only`.

static::=`A variable or function associated with a class rather than with the objects of the class`. If you are in doubt about whether some data should be static or not - ask yourself how many times it occurs: once per class(static) or once per object(not static) or many time per object ( in a different class!).

**Note for C programmers.**

Java Static is like C++ not C. In Java a static *field* is shared by all functions in that class. Whether other classes can access it depends on whether it is *private, public*, and/or *protected*. Static data is shared in common by every object in a class and only stored in one place. A method is declared to be static when it associated with the class of objects rather than a particular object in that class. This means that a static function can not refer any data in an object - because it has no object.

A Java field that is nonstatic is attached to an object -- one object to one field instance. Each object in the

class has its own variable, and all functions refer to that particular variable. The variable is created as part of the object and deleted with it. Variables in inner blocks also belong to the function, not the inner block.

final::=`once initialized it can not be changed`, -- UML( `frozen` ), C++(const).
A variable(a field of a class/object or in a function) is declared `final` when you want it to keep its initial value for ever. It is a constant. A *class* is declared as `final` if it can not be extended. A final *method* can not be over-ridden in any extension. The idea here is to stop people subverting an existing class by extending it (across the internet) and redefining a security sensitive method.

native::=`Preprogrammed in another language and running as machine code`. For a tutorial on a way to integrate C functions into Java methods see http://www.javasoft.com/tutorial/native/index.html

synchronized::=`Only one thread can execute this code/method at a time`.
A method or block of code is marked as `synchronized` if only one thread of control can be in it at a time. It protects resources from interference by multiple accesses at one time. Other threads are locked out. The lock uses a semaphore that is an invisible part of an object. If the semaphore belongs to a thread a new thread can not start the statement. On exit the semaphore is released. When a thread enters code the semaphor belongs to the entering thread.

abstract::method=`A method that must exist for objects in a class but is fully defined only in subclasses`,
abstract::class=`A class with one or more abstract methods`.
abstract::=`not concrete, not yet implemented, deferred to a subclass, prototypical`.
An abstract method is one that is defined in classes derived from this class. All subclasses have a version as defined in the base class or also declare it as abstract. Certain methods can not be abstract: constructors, static, private, those that override superclass methods. An abstract method makes the whole class `abstract`. A class that inherits an abstract method and does not override it is also an abstract class. An abstract class can not be used to declare or create objects. You can only call an abstract method via an object in a class that has extended the abstract class and defined all the abstract methods. An abstract method is a place where functionality can be plugged into an existing framework.

threadsafe::=` If another thread executing this code at the same time can not change the value of a variable then the variable is `threadsafe` and the compiler may do clever things with it to make the code faster or smaller.`.

transient::=`something that does last longer than a function call`.
If an object can exist longer than a given applet....is persistent.... then its transient data does not have to be preserved when a function exits.

**. Terminology**
applet::=`A small program that can be sent across a network and interpreted safely on the receiving machine`.

array::= `an indexed sequence of similar sized objects that are allocated into consecutive locations in memory.`.

API::= Application Programmers Interface. -- for example the AWT and SWING.
AWT::=awt.
awt::=`Abstract windowing toolkit, another windowing toolkit. A set of machine independent classes that make it easier to create graphic user interfaces and output`.

bean::= `a class that follows guidelines about its fields and methods that allow it to be used as a plug compatible component to rapidly develop complex applications`.
beanbox::= `a program that allows someone to combine Java beans into programs by using graphics`.

bytecode::= byte_code.
byte_code::=`a way of describing classes by a stream of bytes that are the machine code for the Java Virtual Machine`.

file::=`A collection of data.  A Java source code file defines one or more classes and interfaces to be added to a package`.

class::=`Defines a collection of knowledge and know how.  Classes are defined as declaring variables(fields) and functions(methods) associated with the objects of that class, and also with the class itself.`.

field::=`A component or member of a class that holds data about any object or about the class, a variable or constant`.

function::computing=`A named piece of code that returns a value and may also do something`

function::java=`A piece of know-how attached to an object or to a class`.

interface::=`Describes a set of classes in terms of what they can do for you, but allows each class to implement these methods in any way that you wish`.

JDK::=`Java Development Kit: the compiler: javac, interpreter: java, and the classes i n the AWT`.  There are at least two major releases so far.  Later called an SDK...

method::=`A piece of "know-how". A procedure or function that is associated with an object or a class`.

object::=`An instance of a class`.  Each object has an identity, class, and attributes.  Its operations come from its class. Most objects are automatically instances of those classes that there own class extends or implements.

procedure::Computer Science=`A named piece of code that does something for a caller'.
procedure::Java=`a void function`.

SDK::tool=`Software Development Toolkit`.

shadow::=`to have the same name as something else`.

type::=`any array, class, interface or elementary data type.`.

static::=`something associated with a class rather than an object`.

Swing::library=`a more advanced library of windowing and graphic user interface tools`.

field::Java=`A piece of data(knowledge) associated with a class or an object that can be a variable or a constant`.

variable::= `a field or  local variable that is not final`.

virtual_machine::=`A hypothetical machine that can be emulated on many different actual machines`.

void::=`word used in place of a type to indicate that a subprogram is a procedure and so does not return a value`. -- introduced in ANSI C and still confusing people 20 years later.

womb::=`Java applets are kept alive inside a running program on the hosts machine - this called the womb.  It stops the applet from doing things that might be insecure`.  Also known as the sandbox.