# COP 3330: Object-Oriented Programming Summer 2007

## Introduction to Object-Oriented Programming


Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida


# Course Topics

- Software Development, Object Oriented Software Development
- Introduction to Java
  - First Application and Applet Programs
- Simple Java Statements
  - Variables, Declarations, Assignment Statements, Simple I/0, Creating Objects
  - Control Statements, Boolean Expressions, Loops, Arrays, Strings



# Course Topics (cont.)

- Writing Classes
  - Methods, Parameter Passing, Static Modifier, Constructors
  - Interfaces, Events and Listeners
- Inheritance
  - Extending Classes, Designing Classes, Class Hierarchies
- Exceptions, I/O Streams
- Graphical User Interface (GUI)
  - Containers, Components, Layout Managers
- Design by Abstraction (UML Diagrams)



# Programming

- A *program* is a set of instructions to solve a given task.
- These instructions are not given in English. They are in a form that a computer can understand.
- Before we write a computer program to solve a problem, we should organize its solution. (*problem solving*)
- Normally we are good in problem solving, but we should apply certain methods to solve problems (especially when we solve large problems)



# Programming (cont.)

- Good problem solving steps make life easier when we write a computer program to solve a given problem. We will talk about top-down approach (divide and conquer) when we organize solutions for problems.
- We will also talk about object-orient software development techniques.



# Programming Languages

- Instructions to solve a problem can be written in many different programming languages.

- Some of them can directly understandable by the computers and others need to be translated into instructions that the computer can understand.



# Programming Languages (cont.)

- Programming languages may be divided into: Machine Languages, Assembly Languages, High-Level Languages
- Any computer can directly understand its own machine language. (patterns of 0s and 1s). Machine languages are machine dependent and cumbersome for humans.
- Assembly languages are English like abbreviations of machine instructions Assembly programs are translated into machine languages using assemblers.



# Programming Languages (cont.)

- Some of High-Level Languages: Pascal, ALGOL, FORTRAN, Basic, C, C++, Java, Lisp, Prolog
- Compilers convert programs written in high-level languages into machine languages



# Software Development

- Steps of developing a software to solve a problem:

  1. *Problem Understanding*
     - Read the problem carefully and try to understand what is required for its solution.
  2. *Analysis and Design*
     - Identify problem inputs and outputs.
     - Identify the data structures to model the data which is required for the problem.
     - Develop a list of steps (algorithm) to solve the problem
     - Refine steps of this algorithm. (Divide and Conquer)
     - Verify that the algorithm solves the problem, i.e. the algorithm is correct



# Software Development (cont.)

3. *Implementation*
   - Implement the algorithm as a (java) program.
   - You have to know a specific programming language (java).
   - Convert steps of the algorithm into programming language statements.
4. *Testing and Verification*
   - Test the completed program, and verify that it works as expected .
   - Use different test cases (not one) including critical test cases.



# Desirable Qualities of Software Systems

- Usefulness
  - Should adequately address the needs of their intended users in solving problems and providing services.
- Timeliness
  - Should be completed and shipped in a timely manner.
- Reliability
  - Should perform as expected by users in terms of the correctness of the functions being performed, and an acceptable level of failures.
- Maintainability
  - Should be easily maintained, easy to make corrections.



# Desirable Qualities (cont.)

- Reusability
  - Components of software systems should be designed as general solutions (not ad hoc)
- User-Friendliness
  - Should provide user friendly interfaces.
- Efficiency
  - Should not waste the system resources (time,memory and disk space)



# Components of Software Systems

- A software system usually consists of two components:
  - ***Model*** **–** represents the organization of the required data
  - ***Algorithm*** **–** computations involved in the processing the data represented by the model.
- In the analysis and design phase of a software development, the required data for that software should be organized as a model, and the algorithm which manipulates the data should be developed.



# Components of Software Systems (cont.)

- In the classical software development, the emphasis is on the algorithm part of the software.
- In the object-oriented software development, a balanced view of the data and the computations is to be captured.



# Object-Oriented Software Development

- Object-Oriented models are composed of **objects**.
- Objects contain data and make computations.
- The decomposition of a complex system is based on the structure of classes, objects, and the relationship among them. (divide-and-conquer).
- When we divide our problems into sub-problems, we will try to design classes to solve these sub-problems.
- We will use graphical notation to describe object-oriented analysis and design models. This notation is based on *Unified Language Modelling (UML).*



# Classes and Objects

- **Objects** and **classes** are two fundamental concepts in the object-oriented software development.
- An *object* has a unique *identity*, a *state*, and *behaviors*. In the real life, an object is anything that can be distinctly identified.
- A *class* characterizes the structure of states and behaviors that shared by all its instances.
- The terms *object* and *instance* are often interchangeable.



# Classes and Objects (cont.)

- The *features* of an object are the combination of the *state* and *behavior* of that object.
  - The state of an object is composed of a set of *attributes* (fields) and their current values.
  - The behavior of an object is defined by a set of *methods* (operations, functions, procedures).
- A class is a template for its instances. Instead of defining the features of objects, we define features of the classes to which these objects belong.



# Classes in Java

- A class in Java can be defined as follows:

```
class Rectangle {
    int length, width;
    public int area() {……}
    public void changeSizes(int x, int y)
     { …… }
```

- The name of the class is: Rectangle. Its attributes are: length, width. Its methods are: area, changeSizes

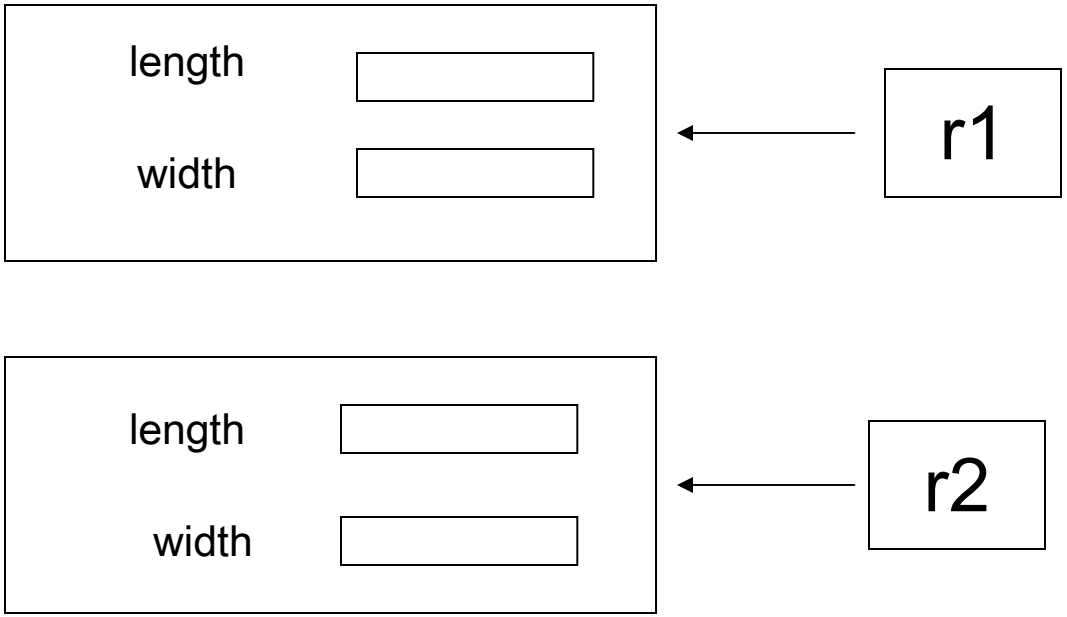
- This Rectangle class is a template for all rectangle objects. All instances of this class will have same structure.



# Objects in Java

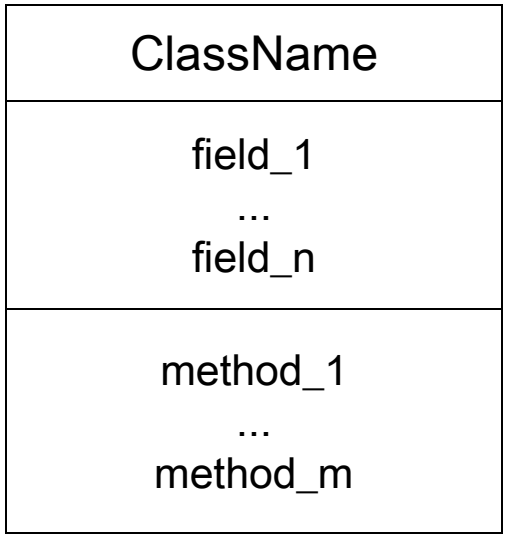
- An object in Java is created from a class using new operator.

```
Rectangle r1 = new Rectangle();
Rectangle r2 = new Rectangle();
```






# Graphical Representation of Classes




ClassName is the name of the class

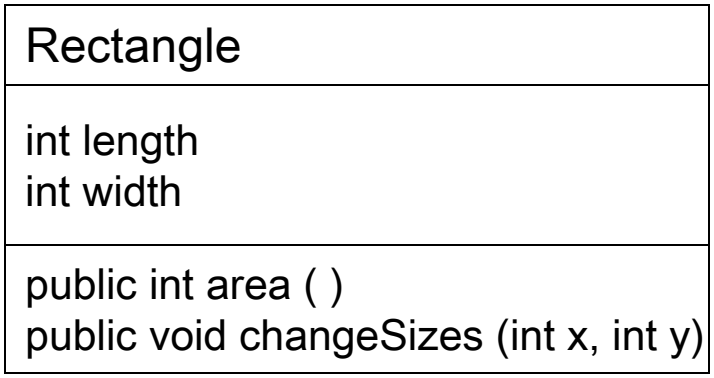
Each field is

[Visibility][Type] identifier [=initialvalue]

Each method is

[Visibility][Type] identifier ( [parameter-list] )

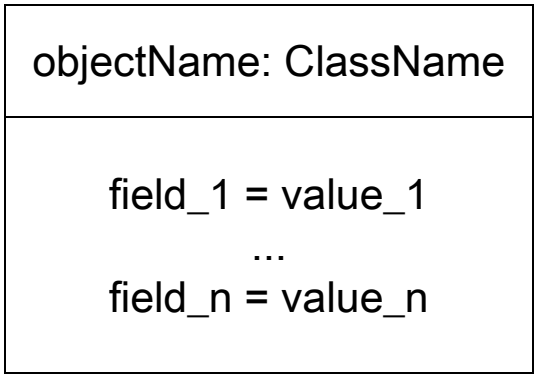
Example:



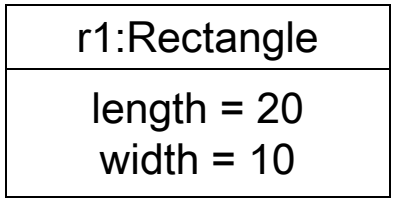

- We may not give field, methods parts



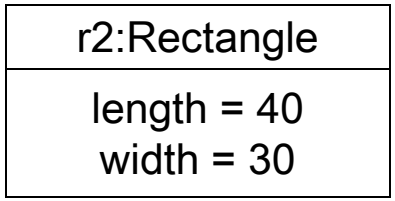
# Graphical Representations of Objects




- We may omit ClassName, and just use objectName. In this case the class of the object is no interest for us.
- We may omit objectName, and just use :ClassName. In this case, the object is an anonymous object.




```
Rectangle r1 = new Rectangle();
r1.length = 20;
r1.width = 10;
```




```
Rectangle r2 = new Rectangle();
r2.length = 40;
r2.width = 30;
```



# Message Passing

- A **message** consists of a *receiving object (recipient)*, *the method to be invoked*, and *arguments* to the method.
- Message passing is also known as *method invocation*.

`r1.changeSizes(4,3)` to change the size of r1 object

`r2.area()` to calculate the area of r2 object



# Modularity

- A complex system should be decomposed into a set of highly cohesive but loosely coupled *modules*.
- A system may be extremely complex in its totality, but a modular decomposition of the system aims to break it down into modules so that:
  - Each module is relatively small and simple (*highly cohesive*)
  - The interactions among modules are relatively simple (*loosely coupled*).



# Modularity (cont.)

- Modular decompositions are hierarchical (module may contain sub-modules).
- *Cohesion* refers to the functional relatedness of the entities within a module.
- *Coupling* refers to the interdependency among different modules.



# Abstraction and Encapsulation

- *Abstraction* and *encapsulation* are powerful tools for deriving modular decomposition of systems.
- Abstraction means to separate the essential from the non-essential characteristics of an entity.
- **Abstraction**: The behaviors and functionalities of a module should be characterized in precise description known as the *contractual interface* of the module.



# Abstraction and Encapsulation (cont.)

- **Encapsulation**: The clients (users of the module) need know nothing more than the service contract (contractual interface) while using the service (where the module is the service provider).
- The implementation of the module should be separated from its contractual interface and hidden from the clients of that module (information hiding).
- Encapsulation tries to reduce the coupling among the modules.



# Interface

- A contractual interface without any implementation associated with it is known as an *interface* (or as an abstract data type) in Java terminology.
- A module can be represented by two separate entities:
  - An interface that describes the contractual interface of the module.
  - A class that implements the contractual interface.
- In Java, we define input/output behavior methods (without implementing those methods) using interfaces.
- A class implementing an interface gives the implementation of all the methods.



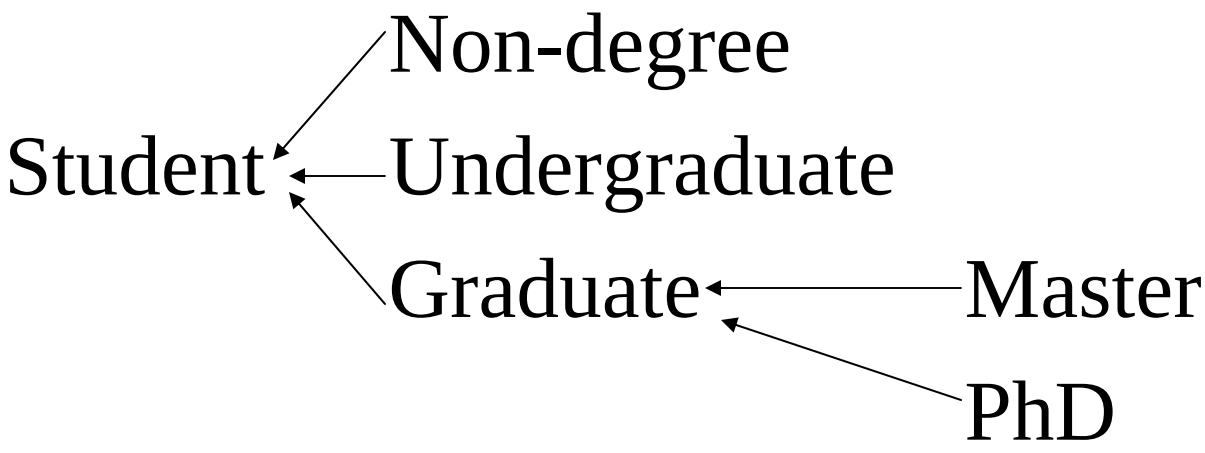
# Inheritance

- Inheritance defines a relationship among classes.
- A class C2 inherits from (extends) another class C1.
- C2 is the sub-class (child) of C1;  C1 is the super-class (parent) of C2.
- C2 inherits attributes and methods defined in C1. In addition, it may also define new attributes and methods.



# Inheritance (cont.)

- Inheritance allows the fields and methods of a super-class to be shared by its sub-classes.






# Class Diagrams

- To design a software in an object-oriented environment, first we design our *model*.
- Our model consists of the classes will be used in our software, their hierarchies, and their relationships among them.
- As an example consider the University environment illustrated in the next slide.



# Class Diagram Example

- We may have a Student class and this class may have sub-classes. There will be a hierarchy among all student classes (Student, NonDegree, UnderGrad, Grad, Master, PhD). This will be accomplished by inheritance mechanism.
- We may also have Course, Department and Faculty classes.
- Each Faculty will be a member of a Department, Each Student may be student of one ore more Departments. A Faculty will be the chairman of a Department.
- Different Students may enroll to different Courses, a Faculty will teach a Course.
- A Faculty can be advisor of many Students.

- All of these relations can be shown as class diagrams in UML notation.



# Algorithms

- After we designed the model for our software, we design algorithms for our software.
- We give the order of operations and the creation order of objects.
- In short, we design an algorithm (steps of our solution for the given problem) for the dynamic behavior of our software.
- We may use different notations to specify our algorithms:
  - classical representation – list of steps, conditional execution, repetitions, ...
  - a sequence diagram – indicating order of operations.
  - a pseudo code
  - flow charts



# Java & Object-Oriented Programming

- Java is an object-oriented programming language that was developed at Sun Microsystems (by a team lead by James Gosling).
- Java is one of many object-oriented programming languages (others: C++, Smalltalk, Objective C,...).
- Java not only supports object-oriented programming, but it also prohibits many bad programming styles and practices (pointers, goto are not available in Java).
- Java is platform-independent. Java programs are compiled into Java byte-codes and these byte-codes are interpreted by Java byte-code interpreters available in different platforms.



# Executing Programs

- Programs in the most of programming languages (such as C, Pascal,...) are *compiled* into the executable files, and these executables files are directly executed by the operating system and the hardware.
- These executable files are platform-dependent. They can only run on the intended platforms.
- So, if we want to run our software on different platforms, we have to prepare its versions for all platforms.
- Since the executable files are in machine codes, they can run fast.



# Executing Programs (cont.)

- Another approach to execute programs is *interpretation*.
- A source program in a programming language is directly interpreted by the interpreter of that programming language (such as some versions of LISP, Smalltalk).
- The interpretation approach is platform-independent.
- The execution speeds of programs will be much slower when compared with compiled executables.



# Java Execution Model

- Java execution model compromises between conventional compilation and interpretation approaches.
- Java programs are compiled into Java byte-code (Machine Code of Java Virtual Machine). The Java byte-code is independent from the machine code of any architecture. But they are close to machine code.
- These generated Java byte-codes are interpreted by Java interpreters available in different platforms.
- So, the generated byte-codes are portable among different systems.
- The execution of Java programs are slower because we still use the interpretation approach.



# Preparing a Java Program

- We are going to use JDK environment of Sun Microsystems.
- JDK environment is simple to use and free.
- You can JDK environment for your own computer from the Sun website:

  http://java.sun.com



# Preparing a Java Program (cont.)

- Editing the Program
  - Create a file containing a Java program.
  - You may use any text editor.
    - You may use the textpad editor to create your .java files. You can get the textpad editor from the website http://www.textpad.com/
    - On the Olympus system you can use emacs, or vi.
  - This file will have a **.java** extension
    - (Example: Example1.java)



# Preparing a Java Program (cont.)

- Compiling
  - Use Java compiler to compile the Java program. ( javac Example1.java )
  - Java compiler will create a file with a **.class** extension. This file will contain the Java byte-codes of your Java program.
    - Example: Example1.class.
  - You can run this file in different platforms.
  - The other compilers produce executable files.



# Executing a Java Program

- **Executing:**
  - Execute the byte-codes of your Java program by a Java interpreter.
  - We will see that there are two types of Java programs:
    - **Application**
    - **Applet**
  - If our program is an application, we will execute it as follows:

```
java Example1
```

this will interpret Example1.class file



# Executing a Java Program (cont.)

- If our program is an applet:
  1. First we will create a **`.html`** file containing a link to our **`.class`** file.
  2. Then we will run our applet using appletviewer:

```
appletviewer Example1.html
```

- We may run our applets under web-browsers too.



# A Simple Console Application Program

```
//Developer:  Mark Llewellyn          Date: May 15, 2007
//
//A Java application program that prints "Hello there, World!"

public class Example1
{       public static void main (String args[ ])
        {       //print message
                System.out.println("Hello there, World!");
        }  //end main
}  //end class Example1
```

## A Simple Console Application Program (cont.)

- Every Java application program defines a class.
  - Use the keyword **class** to declare a class.
  - The name of a class is an identifier. Uppercase and lowercase letters are different.
  - **The name of the class must be same as the the file holding that class (Example1.java)**

# A Simple Console Application Program (cont.)

- Our class contains only one method.
  - Every console application program should contain a *main* method.
  - It may contain other methods too.
  - The method main should be declared using *public* and *static* keywords.
  - Our main method contains only one executable statement
    - This is a method invocation statement (print statement)
  - // single line comments
  - /* ... */ multi line comments
- Use white space between words/lines/blocks to make easier to read programs.



# A Simple Console Application Program (cont.)

- Using a text editor, put this application program into the file `Example1.java`
- To compile: `javac Example1.java`
  - creates `Example1.class` file (if there are no syntax errors in the code)
- To run: `java Example1`
  - java interpreter will interpret the byte-codes in `Example1.class` file.
  - As a result, we will see *Hello there, World!* appear on the screen.



# Simple Java Program (cont.)

- What do we have in our simple Java program?
  - Identifiers – `Example1,args,...`
  - Reserved Words – `public, class, static, ...`
  - Literals – `"Hello there, World!"`
  - Operators -- `.`
  - Delimeters -- `{ } ( ) [ ] ;`
  - Comments -- `// end of class`
- When these parts are combined according to certain rules (the syntax of Java), a syntactically correct Java program is created.



# Identifiers

- We make up words for class names, method names, and variables.
- These words are called ***identifiers.***
- For example,
  - `Example1, main, System, out, println` are identifiers in our simple program.
  - We made up `Example1` and `main`(!); and others are defined in Java API (Application Programming Interface).



# Identifiers (cont.)

- An *identifier* can be composed of any combination of letters, digits, the under score character, and the dollar sign; but it cannot start with a digit.
- We can use both upper case letters and lower case letters in identifiers. But Java is case sensitive. Identifiers `Val`, `val` and `VAL` are all different variables.
- *Some Legal Identifiers*: `x val count_flag Test1 $amount val1 stockItem`
- *Some Illegal Identifiers*: `1val x# x-1 x+`

# Identifiers (cont.)

- Although we can choose any legal identifier to be used, but it is nice to follow certain style guidelines when make up an identifier.
  - Choose meaningful names (not too long, not too short, descriptive words)
  - First character of a class name should be an uppercase letter.
  - First character of a method name and a variable should be a lower case letter.



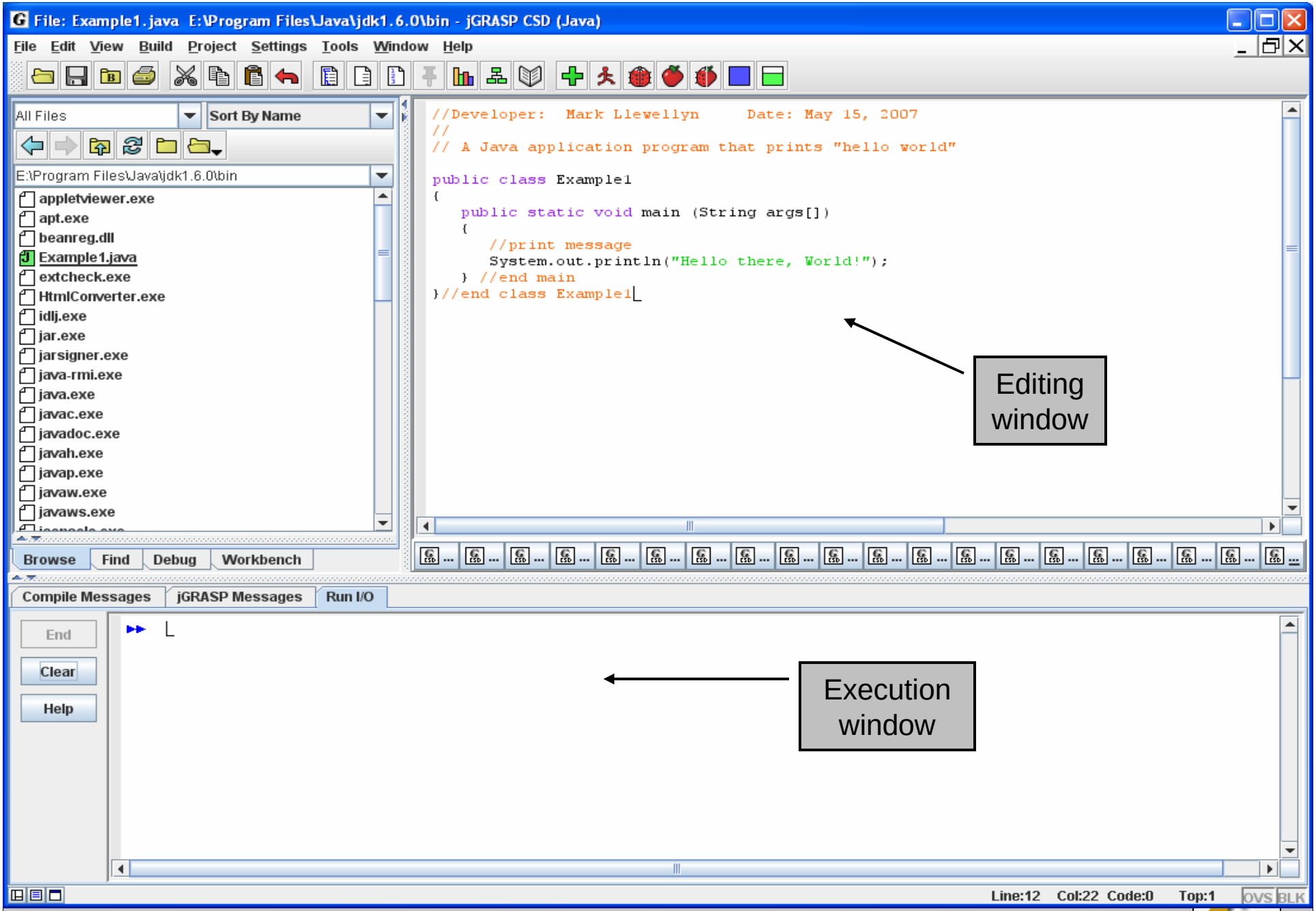
# Reserved Words

- ***Reserved words*** are identifiers that have a special meaning in a programming language.
- For example,
  - `public, void, class, static` are reserved words in our simple programs.
- In Java, all reserved words are lower case identifiers (Of course we can use just lower case letters for our own identifiers too)
- We cannot use the reserved words as our own identifiers (i.e. we cannot use them as variables, class names, and method names).

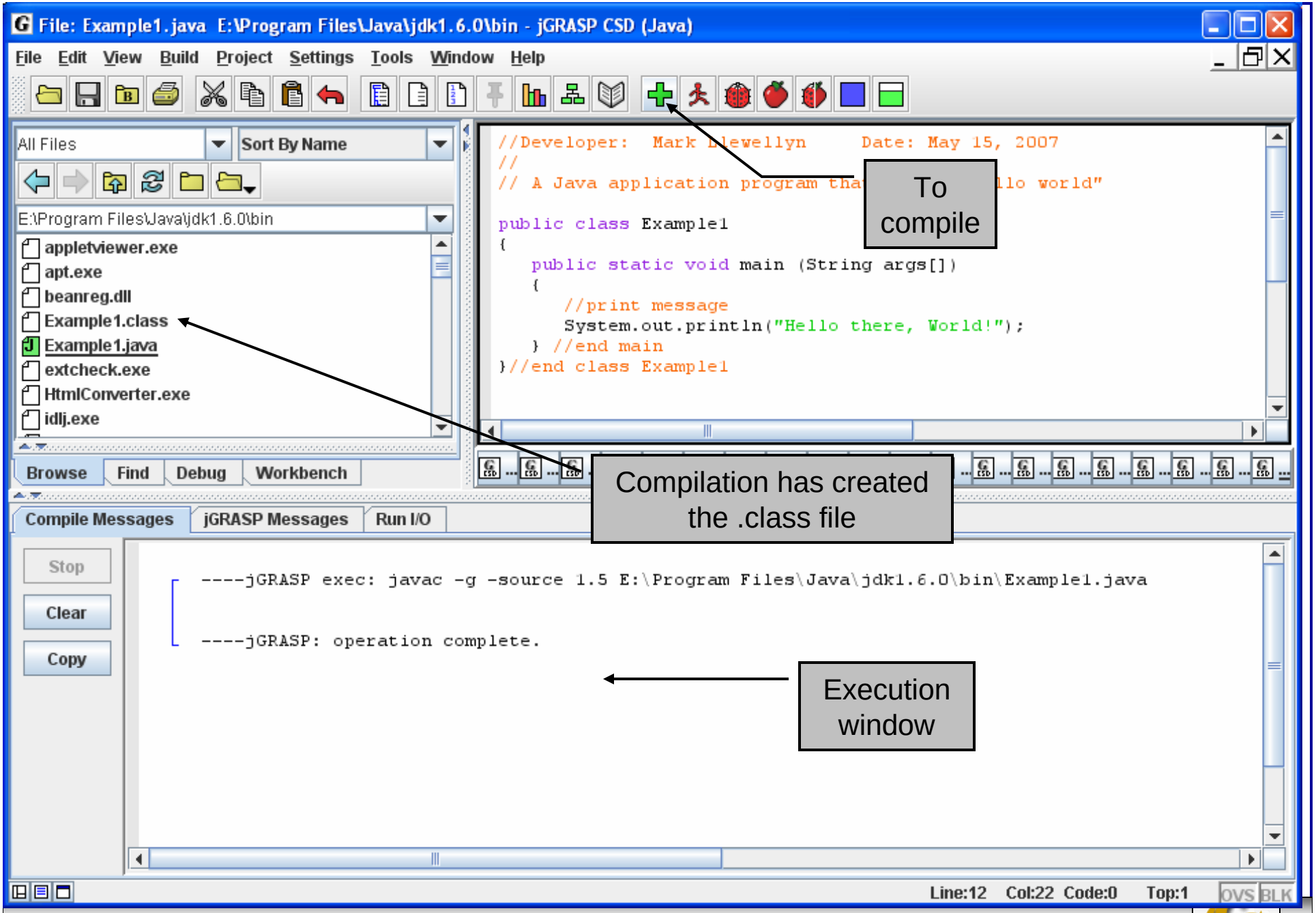


# Literals

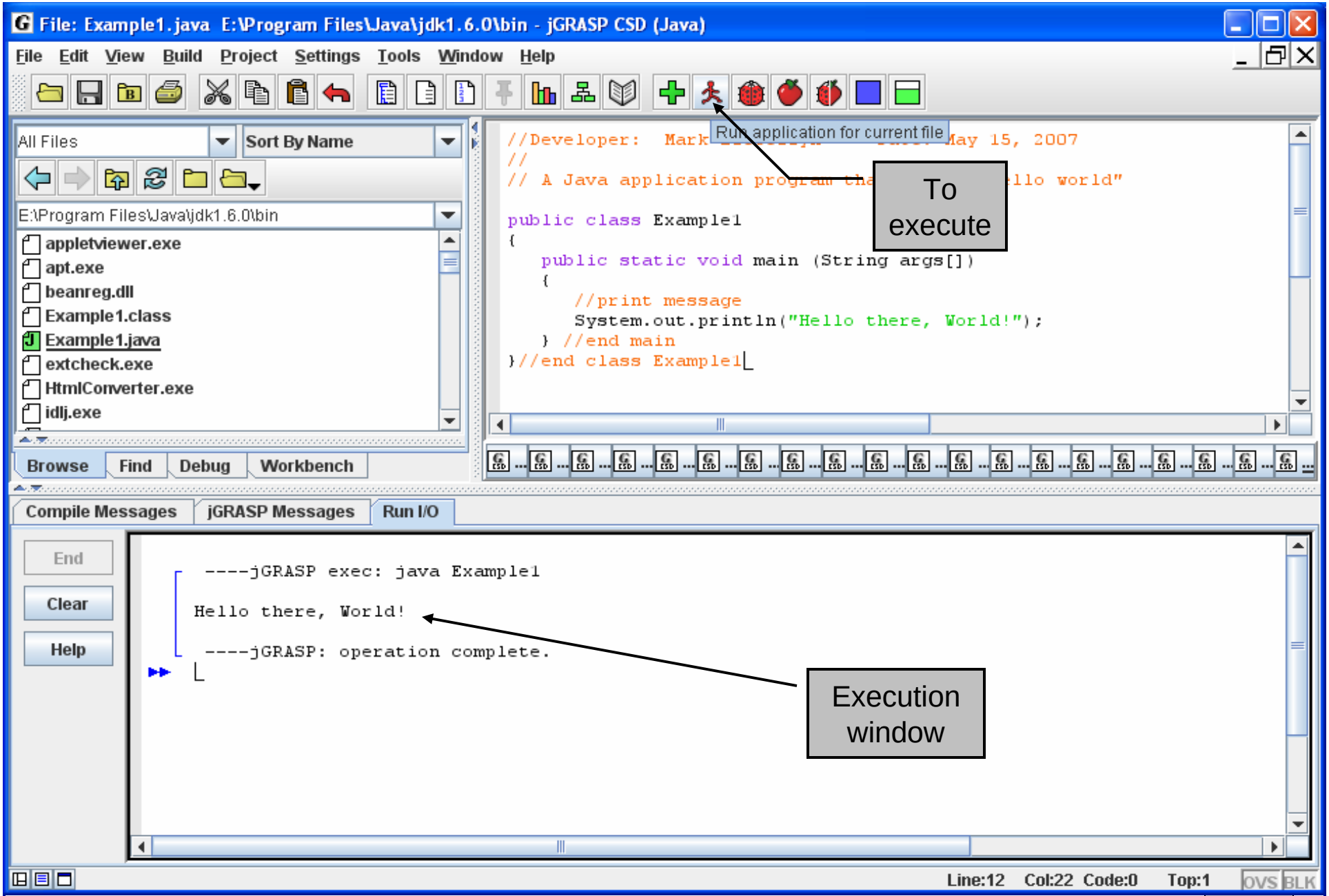
- ***Literals*** are explicit values used in a program.
- Certain data types can have literals.
  - String literal – `"Hello there,World!"`
  - integer literals -- `12  3  77`
  - double literals – `12.1  3.45`
  - character literals – `'a'  '1'`
  - Boolean literals -- `true  false`

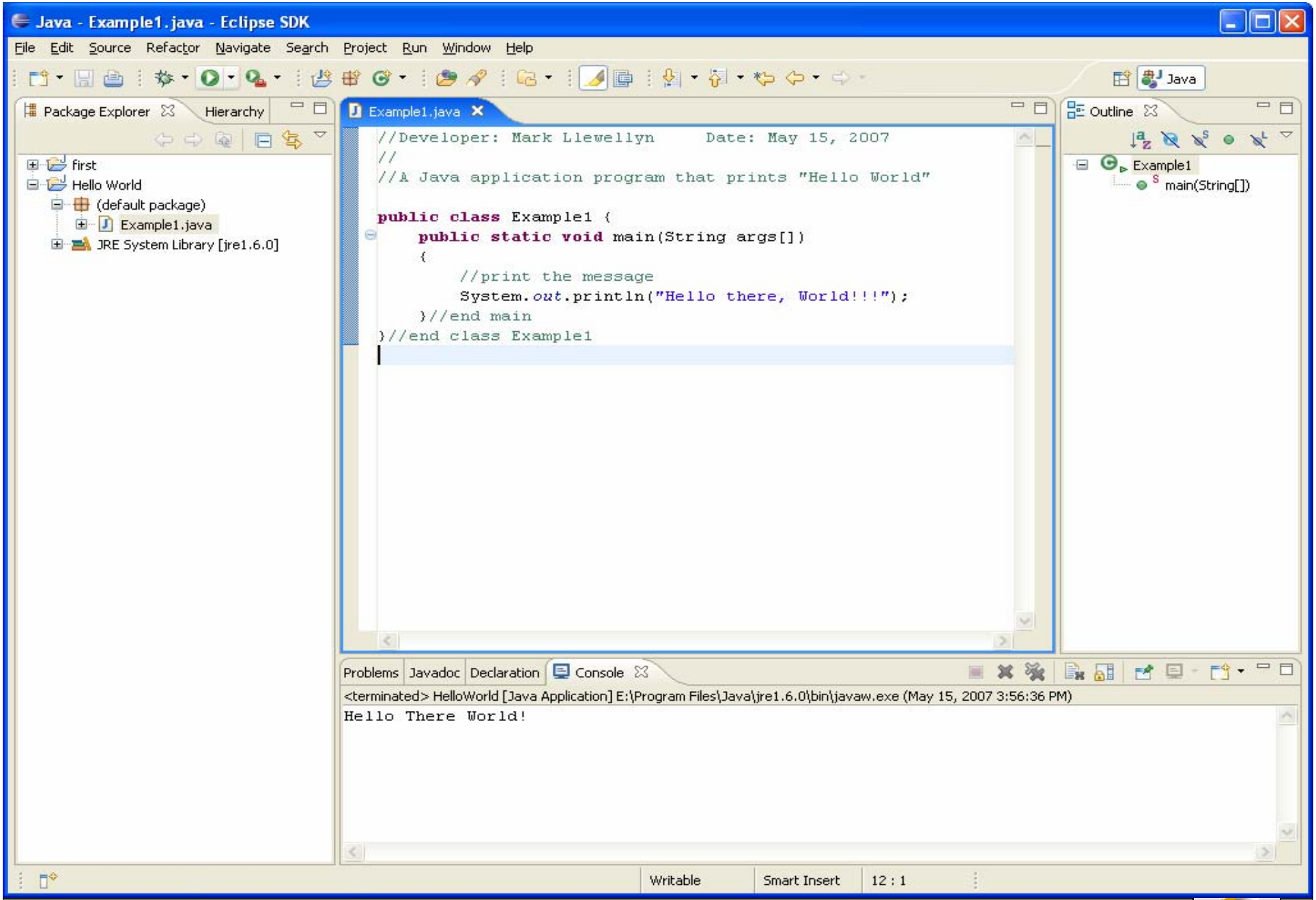

File: Example1.java E:\Program Files\Java\jdk1.6.0\bin - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
E:\Program Files\Java\jdk1.6.0\bin
appletviewer.exe
apt.exe
beanreg.dll
Example1.java
extcheck.exe
HtmlConverter.exe
idlj.exe
jar.exe
jarsigner.exe
java-rmi.exe
java.exe
javac.exe
javadoc.exe
javah.exe
javap.exe
javaw.exe
javaws.exe
Browse
Find
Debug
Workbench
//Developer: Mark Llewellyn Date: May 15, 2007
//
// A Java application program that prints "hello world"
public class Example1
{
public static void main (String args[])
{
//print message
System.out.println("Hello there, World!");
} //end main
}//end class Example1
Editing window
Compile Messages
jGRASP Messages
Run I/O
End
Clear
Help
Execution window
Line:12 Col:22 Code:0 Top:1
OVS
BLK




File: Example1.java E:\Program Files\Java\jdk1.6.0\bin - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
All Files
Sort By Name
E:\Program Files\Java\jdk1.6.0\bin
appletviewer.exe
apt.exe
beanreg.dll
Example1.class
Example1.java
extcheck.exe
HtmlConverter.exe
idlj.exe
Browse Find Debug Workbench
//Developer: Mark Llewellyn Date: May 15, 2007
// A Java application program tha llo world"
public class Example1
{
public static void main (String args[])
{
//print message
System.out.println("Hello there, World!");
} //end main
}//end class Example1
To compile
Compilation has created the .class file
Compile Messages
jGRASP Messages
Run I/O
Stop
Clear
Copy
----jGRASP exec: javac -g -source 1.5 E:\Program Files\Java\jdk1.6.0\bin\Example1.java
----jGRASP: operation complete.
Execution window
Line:12 Col:22 Code:0 Top:1
OVS BLK




File: Example1.java E:\Program Files\Java\jdk1.6.0\bin - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
Run application for current file
To execute
All Files
Sort By Name
E:\Program Files\Java\jdk1.6.0\bin
appletviewer.exe
apt.exe
beanreg.dll
Example1.class
Example1.java
extcheck.exe
HtmlConverter.exe
idlj.exe
Browse Find Debug Workbench
//Developer: Mark
May 15, 2007
// A Java application program
public class Example1
{
public static void main (String args[])
{
//print message
System.out.println("Hello there, World!");
} //end main
}//end class Example1
Compile Messages jGRASP Messages Run I/O
End
Clear
Help
----jGRASP exec: java Example1
Hello there, World!
----jGRASP: operation complete.
Execution window
Line:12 Col:22 Code:0 Top:1


Java - Example1.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
Hierarchy
first
Hello World
(default package)
Example1.java
JRE System Library [jre1.6.0]
Example1.java
//Developer: Mark Llewellyn Date: May 15, 2007
//
//A Java application program that prints "Hello World"
public class Example1 {
public static void main(String args[])
{
//print the message
System.out.println("Hello there, World!!!");
}//end main
}//end class Example1
Outline
Example1
main(String[])
Problems
Javadoc
Declaration
Console
<terminated> HelloWorld [Java Application] E:\Program Files\Java\jre1.6.0\bin\javaw.exe (May 15, 2007 3:56:36 PM)
Hello There World!
Writable
Smart Insert
12 : 1



# A Simple Applet Program

```
//Developer: Mark Llewellyn    Date: May 15, 2007
//
// A Java applet program that prints “Hello there, World!”

import java.awt.*;
import java.applet.Applet;

public class Example2Applet extends Applet
{
       public void paint (Graphics page) {
          page.drawString(“Hello there, World!”, 50,50);
       } // end of paint method

}  // end of class
```



# A Simple Applet Program (cont.)

- We import classes from packages `java.awt` and `java.applet`. (awt = Abstract Windows Toolkit)
  - From java.awt package, we use Graphics class.
  - From java.applet package, we use Applet class (and its methods).
- Our new class `Example2Applet` extends the already existing class Applet.
  - Our class will inherit all methods of Applet if they are not declared in our method



# A Simple Applet Program (cont.)

- We declared only a `paint` method
  - it is called after the initialization
  - it is also called automatically every time the applet needs to be repainted
- Event-driven programming
  - methods are automatically called responding to certain events
- `drawString` writes a string on the applet.
  - drawString(*string*, *x*, *y*)
  - top leftcorner of an applet is 0,0

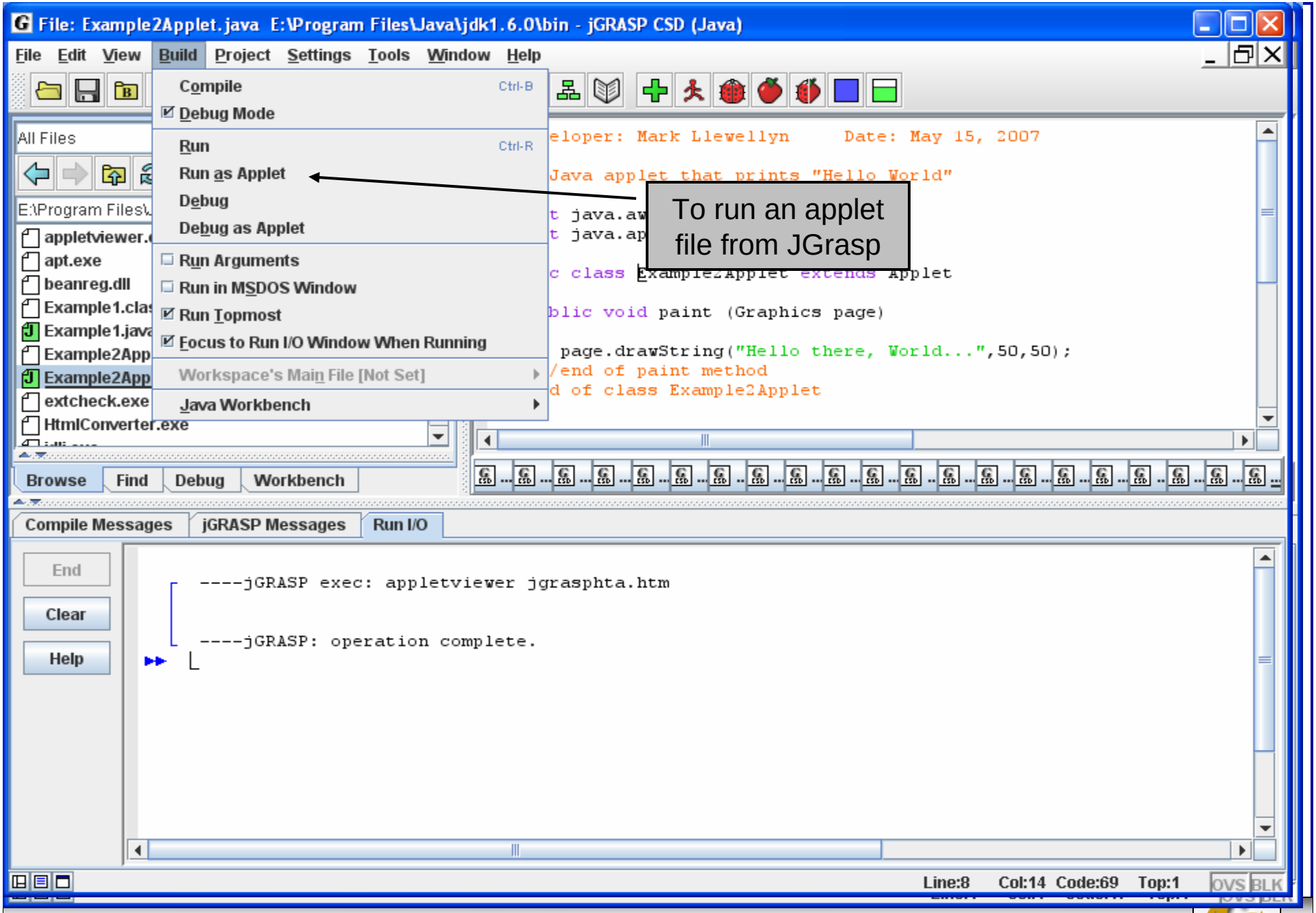


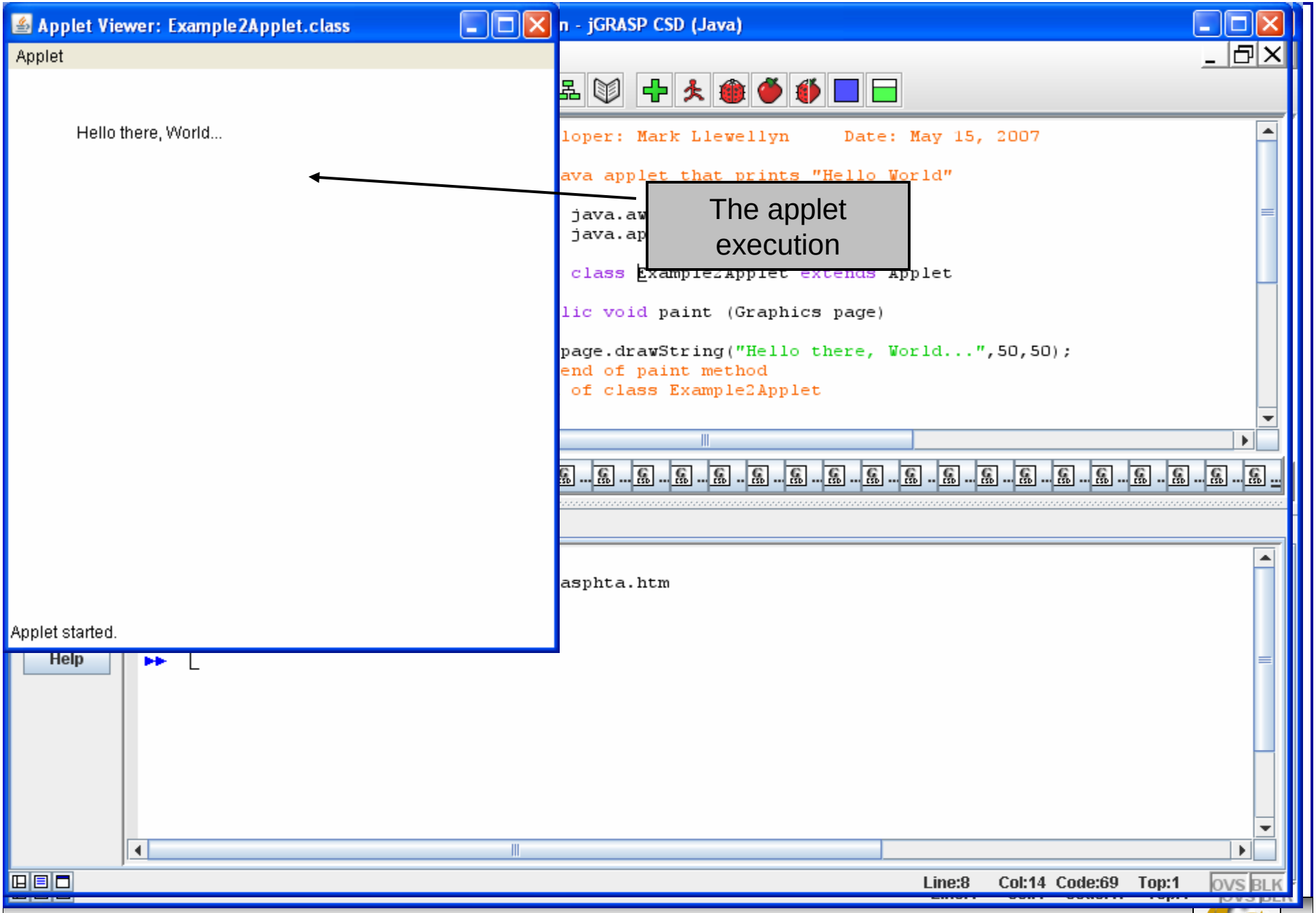
# A Simple Applet Program (cont.)

- To compile:
  - `javac Example2Applet.java`
- To run:
  - `appletviewer Example2Applet.html`
  - It will print "Hello there, World!" in the applet window.
  - The `Example2Applet.html` file should contain:

```
<html>
<applet code="Example2Applet.class" width=300 height=100>
</applet>
</html>
```




File: Example2Applet.java E:\Program Files\Java\jdk1.6.0\bin - jGRASP CSD (Java)
File Edit View Build Project Settings Tools Window Help
Compile Ctrl-B
Debug Mode
Run Ctrl-R
Run as Applet
Debug
Debug as Applet
Run Arguments
Run in MSDOS Window
Run Topmost
Focus to Run I/O Window When Running
Workspace's Main File [Not Set]
Java Workbench
To run an applet file from JGrasp
Browse Find Debug Workbench
Compile Messages jGRASP Messages Run I/O
End
Clear
Help
----jGRASP exec: appletviewer jgrasphta.htm
----jGRASP: operation complete.
Line:8 Col:14 Code:69 Top:1


Applet Viewer: Example2Applet.class
Applet
Hello there, World...
Applet started.
n - jGRASP CSD (Java)
loper: Mark Llewellyn Date: May 15, 2007
ava applet that prints "Hello World"
java.aw
java.ap
class Example2Applet extends Applet
lic void paint (Graphics page)
page.drawString("Hello there, World...",50,50);
end of paint method
of class Example2Applet
asphta.htm
Help
Line:8 Col:14 Code:69 Top:1
The applet execution

# A Simple Applet Program (cont.)

Output: