CSC1016S NOTES PRSALE003

CHAPTER 1:

Objects and Methods:

Java is an object-oriented programming (OOP) language
– Programming methodology that views a program as consisting of objects that interact with one another by means of actions (called methods)
– Objects of the same kind are said to have the same type or be in the same class

Other high-level languages have constructs called procedures, methods, functions, and/or subprograms
– These types of constructs are called methods in Java
– All programming constructs in Java, including methods, are part of a class

Java Application Programs:

Two common types of Java programs are
applications and applets

A Java application program or "regular" Java program
is a class with a method named main
– When a Java application program is run, the run-time
system automatically invokes the method named main – All Java application programs start with the main method

Applets:

A Java applet (little Java application) is a Java program that is meant to be run from a Web browser
– Can be run from a location on the Internet
– Can also be run with an applet viewer program for
debugging
– Applets always use a windowing interface

In contrast, application programs may use a windowing interface or console (i.e., text) I/O

System.out.println:

Java programs work by having things called objects perform actions
– System.out: an object used for sending output to the screen

The actions performed by an object are called methods
– println: the method or action that the System.out object performs

Invoking or calling a method: When an object performs an action using a method
– Also called sending a message to the object
– Method invocation syntax (in order): an object, a dot
(period), the method name, and a pair of parentheses
– Arguments: Zero or more pieces of information needed by the method that are placed inside the parentheses
System.out.println("This is an argument");
Variable declarations:

Variable declarations in Java are similar to those in other programming languages
– Simply give the type of the variable followed by its name and a semicolon
int answer;

Using = and +:

In Java, the equal sign (=) is used as the assignment operator
– The variable on the left side of the assignment operator is assigned the value of the expression on the right side of the assignment operator
answer = 2 + 2;

In Java, the plus sign (+) can be used to denote addition (as above) or concatenation
– Using +, two strings can be connected together System.out.println("2 plus 2 is " + answer);

Code: A program or a part of a program
Source code (or source program): A program written in a high-level language such as Java
– The input to the compiler program
Object code: The translated low-level program
– The output from the compiler program, e.g., Java byte- code
– In the case of Java byte-code, the input to the Java byte- code interpreter

Compiling a Java Program or Class:

Each class definition must be in a file whose name is the same as the class name followed by .java
– The class FirstProgram must be in a file named FirstProgram.java
Each class is compiled with the command javac followed by the name of the file in which the class resides
javac FirstProgram.java
– The result is a byte-code program whose filename is the same as the class name followed by .class
FirstProgram.class

Running a Java Program:

A Java program can be given the run command
(java) after all its classes have been compiled
– Only run the class that contains the main method (the system will automatically load and run the other classes, if any)
– The main method begins with the line: public static void main(String[ ] args)
– Follow the run command by the name of the class only (no .java or .class extension)
java FirstProgram

Syntax: The arrangement of words and punctuations that are legal in a language, the grammar rules of a language.

Semantics: The meaning of things written while following the syntax rules of a language.

Error Messages:

Bug: A mistake in a program
– The process of eliminating bugs is called debugging

Syntax error: A grammatical mistake in a program
– The compiler can detect these errors, and will output an error message saying what it thinks the error is, and where it thinks the error is

Run-time error: An error that is not detected until a program is run
– The compiler cannot detect these errors: an error message is not generated after compilation, but after execution

Logic error: A mistake in the underlying algorithm for a program
– The compiler cannot detect these errors, and no error message is generated after compilation or execution, but the program does not do what it is supposed to do

Identifiers:

Identifier: The name of a variable or other item (class, method, object, etc.) defined in a program
– A Java identifier must not start with a digit, and all the characters must be letters, digits, or the underscore symbol
– Java identifiers can theoretically be of any length
– Java is a case-sensitive language: Rate, rate, and RATE
are the names of three different variables.

Keywords and Reserved words: Identifiers that have a predefined meaning in Java
– Do not use them to name anything else
public class void static

Predefined identifiers: Identifiers that are defined in libraries required by the Java language standard
– Although they can be redefined, this could be confusing and dangerous if doing so would change their standard meaning
System String println

Naming Conventions:

Start the names of variables, classes, methods, and objects with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters
topSpeed bankRate1 timeOfArrival

Start the names of classes with an uppercase letter and, otherwise, adhere to the rules above

FirstProgram MyClass String

Variable Declarations:

Every variable in a Java program must be declared before it is used
– A variable declaration tells the compiler what kind of data (type) will be stored in the variable
– The type of the variable is followed by one or more variable names separated by commas, and terminated with a semicolon
– Variables are typically declared just before they are used or at the start of a block (indicated by an opening brace {)
– Basic types in Java are called primitive types int numberOfBeans;
double oneWeight, totalWeight;

Primitive Types:

TYPE NAME	KIND OF VALUE	MEMORY USED	SIZE RANGE
`boolean`	`true` or `false`	1 byte	not applicable
`char`	single character (Unicode)	2 bytes	all Unicode characters
`byte`	integer	1 byte	−128 to 127
`short`	integer	2 bytes	−32768 to 32767
`int`	integer	4 bytes	−2147483648 to 2147483647
`long`	integer	8 bytes	−9223372036854775808 to 9223372036854775807
`float`	floating-point number	4 bytes	$-3.40282347 \times 10^{+38}$ to $-1.40239846 \times 10^{-45}$
`double`	floating-point number	8 bytes	$\pm 1.76769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$

Shorthand Assignment Statements:

Example:	Equivalent To:
`count += 2;`	`count = count + 2;`
`sum -= discount;`	`sum = sum - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time / rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= count1 + count2;`	`amount = amount * (count1 + count2);`

Assignment Compatibility:

More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it
Boolean - char – byte – short – int – long – float - double
– Note that as your move down the list from left to right, the range of allowed values for the types becomes larger

An explicit type cast is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., double to int)

Note that in Java an int cannot be assigned to a variable of type boolean, nor can a boolean be assigned to a variable of type int

Constants:

Constant (or literal): An item in Java which has one specific value that cannot change
– Constants of an integer type may not be written with a decimal point (e.g., 10)
– Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., 367000.0 or 0.000589)
– Constant of a floating-point type can also be written in scientific (or floating-point) notation (e.g., 3.67e5 or 5.89e-4)

Note that the number before the e may contain a decimal point, but the number after the e may not

Constants of type char are expressed by placing a
single character in single quotes (e.g., 'Z')

Constants for strings of characters are enclosed by
double quotes (e.g., "Welcome to Java")

There are only two Boolean type constants, true and false
– Note that they must be spelled with all lowercase letters

Arithmetic Operators and Expressions:

As in most languages, expressions can be formed in Java using variables, constants, and arithmetic operators
– These operators are + (addition), - (subtraction), * (multiplication), / (division), and % (modulo, remainder)
– An expression can be used anyplace it is legal to use a value of the type produced by the expression

If an arithmetic operator is combined with int operands,
then the resulting type is int
• If an arithmetic operator is combined with one or two
double operands, then the resulting type is double
• If different types are combined in an expression, then the resulting type is the right-most type is found within the expression
– Exception: If the type produced should be byte or short (according to the rules above), then the type produced will actually be an int

Parentheses and Precedence Rules:

An expression can be fully parenthesized in order to specify exactly what subexpressions are combined with each operator

If some or all of the parentheses in an expression are omitted, Java will follow precedence rules to determine, in effect, where to place them
– However, it's best (and sometimes necessary) to include them

Highest Precedence

First: the unary operators: +, –, ++, ––, and !
Second: the binary arithmetic operators: *, /, and %
Third: the binary arithmetic operators: + and –

Lowest Precedence

When the order of two adjacent operations must be determined, the operation of higher precedence (and its apparent arguments) is grouped before the operation of lower precedence
base + rate * hours is evaluated as base + (rate * hours)

When two operations have equal precedence, the order of operations is determined by associativity rules

– Unary operators of equal precedence are grouped right-to- left
+-+rate is evaluated as +(-(+rate))
– Binary operators of equal precedence are grouped left-to- right
base + rate + hours is evaluated as (base + rate) + hours
– Exception: A string of assignment operators is grouped right-to-left
n1 = n2 = n3; is evaluated as n1 = (n2 = n3);

Round-Off Errors in Floating-Point Numbers:

Floating point numbers are only approximate quantities
– Mathematically, the floating-point number 1.0/3.0 is equal to 0.3333333 . . .
– A computer has a finite amount of storage space

It may store 1.0/3.0 as something like 0.3333333333, which is
slightly smaller than one-third
– Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy

Integer and Floating-Point Division

When one or both operands are a floating-point type, division results in a floating-point type
15.0/2 evaluates to 7.5
When both operands are integer types, division results in an integer type

– Any fractional part is discarded
– The number is not rounded
15/2 evaluates to 7
Be careful to make at least one of the operands a floating- point type if the fractional portion is needed

Increment and Decrement Operators:

When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
– If n is equal to 2, then 2*(++n) evaluates to 6

When either operator follows its variable, and is part of an expression, then the expression is evaluated using the original value of the variable, and only then is the variable value changed
– If n is equal to 2, then 2*(n++) evaluates to 4

The Class String:

There is no primitive type for strings in Java
The class String is a predefined class in Java that is used to store and process strings
Objects of type String are made up of strings of characters that are written within double quotes
– Any quoted string is a constant of type String "Live long and prosper."
A variable of type String can be given the value of a String object
String blessing = "Live long and prosper.";

Concatenation: Using the + operator on two strings in order to connect them to form one longer string
– If greeting is equal to "Hello ", and javaClass is equal to "class", then greeting + javaClass is equal to "Hello class"
Any number of strings can be concatenated together
When a string is combined with almost any other type of item, the result is a string
– "The answer is " + 42 evaluates to "The answer is 42"

Classes, Objects, and Methods:

A class is the name for a type whose values are objects

Objects are entities that store data and take actions
– Objects of the String class store data consisting of strings of characters

The actions that an object can take are called methods
– Methods can return a value of a single type and/or perform an action
– All objects within a class have the same methods, but each can have different data values

`int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns `6`.

`boolean equals(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

`boolean equalsIgnoreCase(Other_String)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "  Hmm  ";`
`pause.trim()` returns `"Hmm"`.

`char charAt(Position)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(Start)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

`String substring(Start, End)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(A_String)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns `-1` if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns `3`, and
`greeting.indexOf("Sally")` returns `-1`.

`int indexOf(A_String, Start)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns `6`.
The same value is returned if `1` is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns `0`.
`name.indexOf("Mary", 8)` returns `−1`.

`int lastIndexOf(A_String)`

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns `0`, and
`name.lastIndexOf("Mary")` returns `12`.

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns `0`, and
`entry.compareTo("above")` returns a positive number.

`int compareToIgnoreCase(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

Escape Sequences:

A backslash (\) immediately preceding a character (i.e., without any space) denotes an escape sequence or an escape character
– The character following the backslash does not have its usual meaning
– Although it is formed using two symbols, it is regarded as a single character

Display 1.6 Escape Sequences

```
\" Double quote.
\' Single quote.
\\ Backslash.
\n New line. Go to the beginning of the next line.
\r Carriage return. Go to the beginning of the current line.
\t Tab. White space up to the next tab stop.
```

String Processing:

A String object in Java is considered to be immutable, i.e., the characters it contains cannot be changed
There is another class in Java called StringBuffer that has methods for editing its string objects
However, it is possible to change the value of a String variable by using an assignment statement

```
String name = "Soprano";
name = "Anthony " + name;
```

Character Sets:

ASCII: A character set used by many programming languages that contains all the characters normally used on an English- language keyboard, plus a few special characters
– Each character is represented by a particular number

Unicode: A character set used by the Java language that includes all the ASCII characters plus many of the characters used in languages with a different alphabet from English

Naming Constants:

Instead of using "anonymous" numbers in a program, always declare them as named constants, and use their name instead

```
public static final int INCHES_PER_FOOT = 12;
public static final double RATE = 0.14;
```

– This prevents a value from being changed inadvertently
– It has the added advantage that when a value must be modified, it need only be changed in one place
– Note the naming convention for constants: Use all uppercase letters, and designate word boundaries with an underscore character

Comments:

A line comment begins with the symbols //, and causes the compiler to ignore the remainder of the line
– This type of comment is used for the code writer or for a programmer who modifies the code
A block comment begins with the symbol pair /*, and ends with the symbol pair */
– The compiler ignores anything in between
– This type of comment can span several lines
– This type of comment provides documentation for the users of the program

Program Documentation

Java comes with a program called javadoc that will automatically extract documentation from block comments in the classes you define
– As long as their opening has an extra asterisk (/**)

Ultimately, a well written program is self-documenting
– Its structure is made clear by the choice of identifier names and the indenting pattern
– When one structure is nested inside another, the inside structure is indented one more level

CHAPTER 2:

println Versus print:

Another method that can be invoked by the System.out object is print

The print method is like println, except that it does not end a line
– With println, the next output goes on a new line
– With print, the next output goes on the same line

Formatting Output with printf:

Starting with version 5.0, Java includes a method named printf that can be used to produce output in a specific format
The Java method printf is similar to the print method – Like print, printf does not advance the output to the next line

System.out.printf can have any number of arguments
– The first argument is always a format string that contains one or more format specifiers for the remaining arguments
– All the arguments except the first are values to be output to the screen

The format string "%6.2f" indicates the following:
– End any text to be output and start the format specifier (%)
– Display up to 6 right-justified characters, pad fewer than six characters

will output the line on the left with blank spaces (i.e., field width is 6)
– Display exactly 2 digits after the decimal point (.2)
– Display a floating point number, and end the format specifier (i.e., the conversion character is f)

Line breaks can be included in a format string using %n.

Display 2.1 Format Specifiers for `System.out.printf`

CONVERSION CHARACTER	TYPE OF OUTPUT	EXAMPLES
d	Decimal (ordinary) integer	%5d %d
f	Fixed-point (everyday notation) floating point	%6.2f %f
e	E-notation floating point	%8.3e %e
g	General floating point (Java decides whether to use E-notation or not)	%8.3g %g
s	String	%12s %s
c	Character	%2c %c

Formatting Money Amounts with printf:

A good format specifier for outputting an amount of money stored as a double type is %.2f
It says to include exactly two digits after the decimal point and to use the smallest field width
that the value will fit into:

```
double price = 19.99;
System.out.printf("The price is $%.2f each.")
```

produces the output:

```
The price is $19.99 each.
```

Legacy Code:

Code that is "old fashioned" but too expensive to replace is called legacy code

Sometimes legacy code is translated into a more modern language
The Java method printf is just like a C language function of the same name
This was done intentionally to make it easier to translate C code into Java

Money Formats:

Using the NumberFormat class enables a program to output amounts of money using the appropriate format
– The NumberFormat class must first be imported in order to use it import java.text.NumberFormat
– An object of NumberFormat must then be created using the getCurrencyInstance() method
– The format method takes a floating-point number as an argument and returns a String value representation of the number in the local currency

Specifying Locale:

Invoking the getCurrencyInstance() method without any arguments produces an object that will format numbers according to the default location
• In contrast, the location can be explicitly specified by providing a location from the Locale class as an argument to the getCurrencyInstance() method
– When doing so, the Locale class must first be imported import java.util.Locale;

Importing Packages and Classes:

Libraries in Java are called packages
– A package is a collection of classes that is stored in a manner that makes it easily accessible to any program
– In order to use a class that belongs to a package, the class must be brought into a program using an import statement
– Classes found in the package java.lang are imported automatically into every Java program

```
import java.text.NumberFormat;
// import theNumberFormat class only
import java.text.*;
//import all the classes in package java.text
```

The DecimalFormat Class:

Using the DecimalFormat class enables a program to format numbers in a variety of ways
– The DecimalFormat class must first be imported
– A DecimalFormat object is associated with a pattern when it is created using the new command
– The object can then be used with the method format to create strings that satisfy the format
– An object of the class DecimalFormat has a number of different methods that can be used to produce numeral strings in various formats

Console Input Using the Scanner Class:

In order to use the Scanner class, a program must include the following line near the start of the file:

```
import java.util.Scanner
```

This statement tells Java to
– Make the Scanner class available to the program
– Find the Scanner class in a library of classes (i.e.,Javapackage) named java.util

The following line creates an object of the class Scanner and names the object keyboard :

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

Although a name like keyboard is often used, a Scanner object can be given any name
– For example, in the following code the Scanner object is named scannerObject

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

Once a Scanner object has been created, a program can then use that object to perform keyboard input using methods of the Scanner class
The method nextInt reads one int value typed in at the keyboard and assigns it to a variable:

```
int numberOfPods = keyboard.nextInt();
```

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

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

Multiple inputs must be separated by whitespace and read by multiple invocations of the appropriate method

– White space is any string of characters, such as blankspaces, tabs, and line breaks that print out as white space

The method nextLine reads an entire line of keyboard input
The code,
String line = keyboard.nextLine();
reads in an entire line and places the string that is read into the variable line

The end of an input line is indicated by the escape sequence '\n'
– This is the character input when the Enter key is pressed
– On the screen it is indicated by the ending of one line and the beginning of the next line

When nextLine reads a line of text, it reads the '\n' character, so the next reading of input begins on the next line
– However, the '\n' does not become part of the string value returned (e.g., the string named by the variable line above does not end with the '\n' character)

The `Scanner` class can be used to obtain input from files as well as from the keyboard. However, here we are assuming it is being used only for input from the keyboard.

To set things up for keyboard input, you need the following at the beginning of the file with the keyboard input code:

```
import java.util.Scanner;
```

You also need the following before the first keyboard input statement:

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

The *Scannner_Object_Name* can then be used with the following methods to read and return various types of data typed on the keyboard.

Values to be read should be separated by whitespace characters, such as blanks and/or new lines. When reading values, these whitespace characters are skipped. (It is possible to change the separators from whitespace to something else, but whitespace is the default and is what we will use.)

Scannner_Object_Name`.nextInt()`

Returns the next value of type `int` that is typed on the keyboard.

Scannner_Object_Name.nextLong()

Returns the next value of type long that is typed on the keyboard.

Scannner_Object_Name.nextByte()

Returns the next value of type byte that is typed on the keyboard.

Scannner_Object_Name.nextShort()

Returns the next value of type short that is typed on the keyboard.

Scannner_Object_Name.nextDouble()

Returns the next value of type double that is typed on the keyboard.

Scannner_Object_Name.nextFloat()

Returns the next value of type float that is typed on the keyboard.

Scannner_Object_Name.next()

Returns the String value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

Scannner_Object_Name.nextBoolean()

Returns the next value of type boolean that is typed on the keyboard. The values of true and false are entered as the strings "true" and "false". Any combination of upper- and/or lowercase letters is allowed in spelling "true" and "false".

Scanner_Object_Name.nextLine()

Reads the rest of the current keyboard input line and returns the characters read as a value of type String. Note that the line terminator '\n' is read and discarded; it is not included in the string returned.

Scanner_Object_Name.useDelimiter(*New_Delimiter*);

Changes the delimiter for keyboard input with *Scanner_Object_Name*. The *New_Delimiter* is a value of type String. After this statement is executed, *New_Delimiter* is the only delimiter that separates words or numbers. See the subsection "Other Input Delimiters" for details.

Text Input:

Import the necessary classes in addition to Scanner

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

• Open the file inside a try/catchblock
– If an error occurs while trying to open the file then
execution jumps to the catch block
• UsenextInt(),nextLine(),etc.to read from the Scanner like reading from the console, except the input comes from the file

```
Scanner fileIn = null ; // initializes fileIn to empty
try
{
        // Attempt to open the file
        fileIn = new Scanner( new FileInputStream("PathToFile"));
}
catch (FileNotFoundException e)
{
        // If the file could not be found, this code is executed
        // and then the program exits
        System.out.println("File not found.");
        System.exit(0);
}
... Code continues here
```

CHAPTER 3:

Flow of Control:

• As in most programming languages, flow of control in Java
refers to its branching and looping mechanisms
• Java has several branching mechanisms: if-else, if, and
switch statements
• Java has three types of loop statements: the while, do-
while, and for statements
• Most branching and looping statements are controlled by Boolean expressions
– A Boolean expression evaluates to either true or false
– The primitive type Boolean may only take the values true or
false

Branching with an if-else Statement:

An if-else statement chooses between two alternative statements based on the value of a Boolean expression
if (Boolean_Expression)

Yes_Statement
else
No_Statement
– The Boolean_Expression must be closed in parentheses
– If the Boolean_Expression is true, then the Yes_Statement
– If the Boolean_Expression is false, then the No_Statementis executed

Compound Statements:

Each Yes_Statement and No_Statement branch of an if-else can be a made up of a single statement or many statements
• Compound Statement: A branch statement that is made up of a list of statements
– A compound statement must always be enclosed in a pair of braces ({ })
– A compound statement can be used anywhere that a single statement can be used

The switch Statement:

The switch statement is the only other kind of Java statement that implements multiway branching
– When a switch statement is evaluated, one of a number of different branches is executed
– The choice of which branch to execute is determined by a controlling expression enclosed in parentheses after the keyword switch
The controlling expression must evaluate to a char,int,short, or byte

Each branch statement in a switch statement starts with the reserved word case, followed by a constant called a case label, followed by a colon, and then a sequence of statements
– Each case label must be of the same type as the controlling expression – Case labels need not be listed in order or span a complete interval, but
each one may appear only once
– Each sequence of statements may be followed by a break statement
(break;)

There can also be a section labeled default:
– The default section is optional, and is usually last
– Even if the case labels cover all possible outcomes in a given switch statement, it is still a good practice to include a default section

• It can be used to output an error message, for example
When the controlling expression is evaluated, the code for the case label whose value matches the controlling expression is executed
– If no case label matches, then the only statements executed are those following the default label (if there is one)

The switch statement ends when it executes a break statement, or when the end of the switch statement is reached
– When the computer executes the statements after a case label, it continues until a break statement is reached
– If the break statement is omitted, then after executing the code for one case, the computer will go on to execute the code for the next case
– If the break statement is omitted inadvertently, the compiler will not issue an error message

```
switch (Controlling_Expression)
{
  case Case_Label_1:
           Statement_Sequence_1
           break;
  case Case_Label_2:
           Statement_Sequence_2
           break;
              :
  case Case_Label_n:
           Statement_Sequence_n
           break;
  default:
           Default_Statement Sequence
           break;
}
```

Boolean Expressions:

A Boolean expression is an expression that is either true or
• The simplest Boolean expressions compare the value of two expressions
time < limit
false
yourScore == myScore
– Note that Java uses two equal signs (==) to perform equality testing:
A single equal sign (=) is used only for assignment

– A Boolean expression does not need to been closed in parentheses, unless it is used in an if-else statement

MATH NOTATION	NAME	JAVA NOTATION	JAVA EXAMPLES
=	Equal to	==	`x + 7 == 2*y` `answer == 'y'`
≠	Not equal to	!=	`score != 0` `answer != 'y'`
>	Greater than	>	`time > limit`
≥	Greater than or equal to	>=	`age >= 21`
<	Less than	<	`pressure < max`
≤	Less than or equal to	<=	`time <=limit`

When two Boolean expressions are combined using the "and" (&&) operator, the entire expression is true provided both expressions are true

When two Boolean expressions are combined using the "or" (||) operator, the entire expression is true as long as one of the expressions is true
– The expression is false only if both expressions are false

Any Boolean expression can be negated using the ! operator
– Otherwise the expression is false
– Place the expression in parentheses and place the ! operator in front of it
Unlike mathematical notation, strings of inequalities must be joined by &&

Short circuit evaluation: Java can take a shortcut when the evaluation of the first part of a Boolean expression produces a result that evaluation of the second part cannot change.

Highest Precedence

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation., [ ], ( )	Left to right
++ (postfix, as in x++), -- (postfix)	Right to left
The unary operators: +, -, ++ (prefix, as in ++x), -- (prefix), and !	Right to left
Type casts (Type)	Right to left
The binary operators *, /, %	Left to right
The binary operators +, -	Left to right
The binary operators <, >, <=, >=	Left to right
The binary operators ==, ! =	Left to right
The binary operator &	Left to right
The binary operator |	Left to right
The binary operator &&	Left to right
The binary operator ||	Left to right
The ternary operator (conditional operator) ?:	Right to left
The assignment operators =, *=, /=, %=, +=, -=, & =, |=	Right to left

Lowest Precedence

Loops:

Java has three types of loop statements: the while, the do-while, and the for statements
– The code that is repeated in a loop is called the body of the loop
– Each repetition of the loop body is called an iteration of the loop

while statement:

A while statement is used to repeat a portion of code (i.e., the loop body) based on the evaluation of a Boolean expression
– The Boolean expression is checked before the loop body is executed.
When false, the loop body is not executed at all

– Before the execution of each following iteration of the loop body, the Boolean expression is checked again
If true, the loop body is executed again
If false, the loop statement ends
– The loop body can consist of a single statement, or multiple statements enclosed in a pair of braces ({ })

```
while (Boolean_Expression)
    Statement
```

Or

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2

    .
    .
    .
    Statement_Last
}
```

```
do
    Statement
while (Boolean_Expression);
```

Or

```
do
{
    Statement_1
    Statement_2

    Statement_Last
}  while (Boolean_Expression);
```

Given the following structure for a `do-while` loop:

```
do
 {
    Statements;
 } while (Boolean condition);
```

The equivalent while loop is:

```
Statements;
while (Boolean condition)
{
    Statements;
}
```

```
while (Boolean condition)
{
    Statements;
}
```

The equivalent `do-while` loop is:

```
if (Boolean condition)
{
    do
    {
        Statements;
    } while (Boolean condition);
}
```

The for Statement:

The for statement is most commonly used to step through
It begins with the keyword for, followed by three expressions in parentheses that describe what to do with one or more controlling variables
– The first expression tells how the control variable or variables are initialized or declared and initialized before the first iteration
– The second expression determines when the loop should end, based on the evaluation of a Boolean expression before each iteration

– The third expression tells how the control variable or variables are updated

```
for (Initializing; Boolean_Expression; Update)
  Body
```

after each iteration of the loop body

The break and continue Statements:

The break statement consists of the keyword break followed by a semicolon
– When executed, the break statement ends the nearest enclosing switch or loop statement
The continue statement consists of the keyword continue followed by a semicolon
– When executed, the continue statement ends the current loop body iteration of the nearest enclosing loop statement
– Note that in a for loop, the continue statement transfers control to the update expression
When loop statements are nested, remember that any break or continue statement applies to the innermost, containing loop statement

The exit Statement:

A break statement will end a loop or switch statement, but will not end the program
The exit statement will immediately end the program as soon as it is invoked:
  System.exit(0);
The exit statement takes one integer argument
– By tradition, a zero argument is used to indicate a normal ending of the program

Generating Random Numbers:

The Random class can be used to generate pseudo-random numbers
– Not truly random, but uniform distribution based on a mathematical function and good enough in most cases
Add the following import import java.util.Random;
Create an object of type Random
    Random rnd = new Random();

To generate random numbers use the nextInt() method to get a random number from 0 to n-1
int i = rnd.nextInt(10); // Random number from 0 to 9

Use the nextDouble() method to get a random number from 0 to 1 (always less than 1)
double d = rnd.nextDouble(); // d is >=0 and < 1

CHAPTER 4:

Classes are the most important language feature that
make object-oriented programming (OOP) possible
Programming in Java consists of defining a number of classes
– Every program is a class
– All helping software consists of classes
– All programmer-defined types are classes

A Class Is a Type:

A class is a special kind of programmer-defined type, and variables can be declared of a class type
A value of a class type is called an object or an instance of the class
– If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing
A class determines the types of data that an object can contain, as well as the actions it can perform

Primitive Type Values vs. Class Type Values:

A primitive type value is a single piece of data
A class type value or object can have multiple pieces of data, as well as actions called methods
– All objects of a class have the same methods
– All objects of a class have the same pieces of data (i.e.,
name, type, and number)
– For a given object, each piece of data can hold a different value

The Contents of a Class Definition:

A class definition specifies the data items and methods that all of its objects will have

- These data items and methods are sometimes called members of the object
- Data items are called fields or instance variables
- Instance variable declarations and method definitions can be placed in any order within the class definition

The new Operator:

An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

The new operator must then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

These can be combined as follows:

```
ClassName classVar = new ClassName();
```

Instance Variables and Methods:

Method definitions are divided into two parts: a heading and a method body:

```
public void myMethod()
```

Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

Invoking a method is equivalent to executing the method body

There are two kinds of methods:

– Methods that compute and return a value
– Methods that perform an action

This type of method does not return a value, and is called a void method

Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

A method that returns a value must specify the type of that value in its heading:
public typeReturned methodName(paramList)
A void method uses the keyword void in its heading to show that it does not return a
value : public void methodName(paramList)

The methods equals and toString:

Java expects certain methods, such as equals and toString, to be in all, or almost all, classes
The purpose of equals, a boolean valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
– Note: You cannot use == to compare objects
public boolean equals(ClassName objectName)
The purpose of the toString method is to return a String value that represents the data in the object
public String toString()

Information hiding is the practice of separating how to use a class from the details of its implementation
– Abstraction is another term used to express the concept of discarding details in order to avoid information overload

Encapsulation means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
– Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
– In Java, hiding details is done by marking them private

The API or application programming interface for a class is a description of how to use the class
– A programmer need only read the
API in order to use a well designed class

An ADT or abstract data type is a data type that is written using good information-hiding techniques.

public and private Modifiers:

The modifier public means that there are no restrictions on
where an instance variable or method can be used
The modifier private means that an instance variable or method cannot be accessed by name outside of the class
It is considered good programming practice to make all instance variables private
Most methods are public, and thus provide controlled access to the object
Usually, methods are private only if used as helping methods for other methods in the class

Accessor and Mutator Methods:

Accessor methods allow the programmer to obtain the value of an object's instance variables
– The data can be accessed but not changed
– The name of an accessor method typically starts with the word get

Mutator methods allow the programmer to change the value of an object's instance variables in a controlled manner
– Incoming data is typically tested and/or filtered
– The name of a mutator method typically starts with the word set

Preconditions and Postconditions:

The precondition of a method states what is assumed to be true when the method is called
• The postcondition of a method states what will be true after the method is executed, as long as the precondition holds
• It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment

Overloading:

Overloading is when two or more methods in the
same class have the same method name
To be valid, any two definitions of the method name must have different signatures
– A signature consists of the name of a method together

with its parameter list
– Differing signatures must have different numbers and/or types of parameters

Overloading and Automatic Type Conversion:

If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
The interaction of overloading and automatic type conversion can have unintended results
In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
– Ambiguous method invocations will produce an error in Java

Constructors:

A constructor is a special kind of method that is designed to initialize the instance variables for an object:
  public ClassName(anyParameters){code}
– A constructor must have the same name as the class – A constructor has no type returned, not even void – Constructors are typically overloaded

A constructor is called when an object of the class is created using new
  ClassName objectName = new ClassName(anyArgs);
– The name of the constructor and its parenthesized list of arguments (if any) must follow the new operator
– This is the only valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method

If a constructor is invoked again (using new), the first object is discarded and an entirely new object is created
– If you need to change the values of instance variables of the object, use mutator methods instead

If you do not include any constructors in your class, Java will automatically create a default or no-argument constructor that takes no arguments, performs no initializations, but allows the object to be created
If you include even one constructor in your class, Java will not provide this default constructor

If you include any constructors in your class, be sure to provide your own no-argument constructor as well

The StringTokenizer Class:

The StringTokenizer class is used to recover the words or tokens in a multi-word String
– You can use whitespace characters to separate each token, or you can specify the characters you wish to use as separators
– In order to use the StringTokenizer class, be sure to include the following at the start of the file:

```
import java.util.StringTokenizer;
```

The class `StringTokenizer` is in the `java.util` package.

`public StringTokenizer(String theString)`

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

`public StringTokenizer(String theString, String delimiters)`

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

`public boolean hasMoreTokens()`

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

`public String nextToken()`

Returns the next token from this tokenizer's string. (Throws `NoSuchElementException` if there are no more tokens to return.)[5]

`public String nextToken(String delimiters)`

First changes the delimiter characters to those in the string `delimiters`. Then returns the next token from this tokenizer's string. After the invocation is completed, the delimiter characters are those in the string `delimiters`.
(Throws `NoSuchElementException` if there are no more tokens to return. Throws `NullPointerException` if `delimiters` is `null`.)[5]

`public int countTokens()`

Returns the number of tokens remaining to be returned by `nextToken`.

CHAPTER 5:

A static method is one that can be used without a calling object
A static method still belongs to a class, and its definition is given inside the class definition
When a static method is defined, the keyword static is placed in the method header
public static returnedType myMethod(parameters) {...}
Static methods are invoked using the class name in place of a calling object
  returnedValue = MyClass.myMethod(arguments);

A static method cannot refer to an instance variable of the class, and it cannot invoke a nonstatic method of the class
– A static method has no this, so it cannot use an instance variable or method that has an implicit or explicit this for a calling object
– A static method can invoke another static method, however

Although the main method is often by itself in a class separate from the other classes of a program, it can also be contained within a regular class definition
– In this way the class in which it is contained can be used to create objects in other classes, or it can be run as a program
– A main method so included in a regular class definition is especially useful when it contains diagnostic code for the class

Static Variables:

A static variable is a variable that belongs to the class as a whole, and not just to one object
– There is only one copy of a static variable per class, unlike instance variables where each object has its own copy
All objects of the class can read and change a static variable
Although a static method cannot access an instance variable,
a static method can access a static variable
A static variable is declared like an instance variable, with the addition of the modifier static
    private static int myStaticVariable;

Wrapper Classes:

Wrapper classes provide a class type corresponding to each of the primitive types

– This makes it possible to have class types that behave somewhat like primitive types
– The wrapper classes for the primitive types byte, short, long, float, double, and char are (in order) Byte, Short, Long, Float, Double, and Character

Boxing: the process of going from a value of a primitive type to an object of its wrapper class
– To convert a primitive value to an "equivalent" class type value, create an object of the corresponding wrapper class using the primitive value as an argument
– The new object will contain an instance variable that stores a copy of the primitive value
– Unlike most other classes, a wrapper class does not have a no-argument constructor

```
Integer integerObject = new Integer(42);
```

Unboxing: the process of going from an object of a wrapper class to the corresponding value of a primitive type
– The methods for converting an object from the wrapper classes Byte, Short, Integer, Long, Float, Double, and Character to their corresponding primitive type are (in order) byteValue, shortValue, intValue, longValue, floatValue, doubleValue, and charValue
– None of these methods take an argument int i = integerObject.intValue();

Instead of creating a wrapper class object using the new operation (as shown before), it can be done as an automatic type cast:

```
Integer integerObject = 42;
```

Instead of having to invoke the appropriate method (such as intValue, doubleValue, charValue, etc.) in order to convert from an object of a wrapper class to a value of its associated primitive type, the primitive value can be recovered automatically

```
int i = integerObject;
```

The class `Character` is in the `java.lang` package, so it requires no `import` statement.

```
public static char toUpperCase(char argument)
```

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

```
public static char toLowerCase(char argument)
```

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

```
public static boolean isUpperCase(char argument)
```

Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLE

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

```
public static boolean isLowerCase(char argument)
```

Returns `true` if its `argument` is a lowercase letter; otherwise returns `false`.

EXAMPLE

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Character.isLowerCase('%')` both return `false`.

```
public static boolean isWhitespace(char argument)
```

Returns `true` if its `argument` is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the line break character (`'\n'`).

EXAMPLE

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

`public static boolean isLetter(char argument)`

Returns `true` if its `argument` is a letter; otherwise returns `false`.

EXAMPLE

`Character.isLetter('A')` returns `true`. `Character.isLetter('%')` and `Character.isLetter('5')` both return `false`.

`public static boolean isDigit(char argument)`

Returns `true` if its `argument` is a digit; otherwise returns `false`.

EXAMPLE

`Character.isDigit('5')` returns `true`. `Character.isDigit('A')` and `Character.isDigit('%')` both return `false`.

`public static boolean isLetterOrDigit(char argument)`

Returns `true` if its `argument` is a letter or a digit; otherwise returns `false`.

EXAMPLE

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return `true`. `Character.isLetterOrDigit('&')` returns `false`.

Variables and Memory:

A computer has two forms of memory
• Secondary memory is used to hold files for "permanent" storage
• Main memory is used by a computer when it is running a program
– Values stored in a program's variables are kept in main memory

Main memory consists of a long list of numbered locations called bytes
– Each byte contains eight bits: eight 0 or 1 digits
• The number that identifies a byte is called its address
– A data item can be stored in one (or more) of these bytes
– The address of the byte is used to find the data item when needed

Values of most data types require more than one byte of storage
– Several adjacent bytes are then used to hold the data item – The entire chunk of memory that holds the data is called
its memory location
– The address of the first byte of this memory location is
used as the address for the data item
• A computer's main memory can be thought of as a
long list of memory locations of varying sizes

References:

Every variable is implemented as a location in computer memory
• When the variable is a primitive type, the value of the variable is stored in the memory location assigned to the variable
– Each primitive type always requires the same amount of memory to store its values

When the variable is a class type, only the memory address (or reference) where its object is located is stored in the memory location assigned to the variable
– The object named by the variable is stored in some other location in memory
– Like primitives, the value of a class variable is a fixed size
– Unlike primitives, the value of a class variable is a memory address or reference
– The object, whose address is stored in the variable, can be of any size

Two reference variables can contain the same reference, and therefore name the same object
– The assignment operator sets the reference (memory address) of one class type variable equal to that of another
– Any change to the object named by one of theses variables will produce a change to the object named by the other variable, since they are the same object

```
variable2 = variable1;
```

Class Parameters:

All parameters in Java are call-by-value parameters
– A parameter is a local variable that is set equal to the value of its argument
– Therefore, any change to the value of the parameter cannot change the value of its argument
• Class type parameters appear to behave differently from primitive type parameters
– They appear to behave in a way similar to parameters in languages that have the call-by-reference parameter passing mechanism

The value plugged in to a class type parameter is a reference (memory address)
– Therefore, the parameter becomes another name for the argument

– Any change made to the object named by the parameter (i.e., changes made to the values of its instance variables) will be made to the object named by the argument, because they are the same object
– Note that, because it still is a call-by-value parameter, any change made to the class type parameter itself (i.e., its address) will not change its argument (the reference or memory address)

A method cannot change the value of a variable of a primitive type that is an argument to the method

In contrast, a method can change the values of the instance variables of a class type that is an argument to the method

The new Operator and Anonymous Objects:

• The new operator invokes a constructor which initializes an object, and returns a reference to the location in memory of the object created
– This reference can be assigned to a variable of the object's class type
• Sometimes the object created is used as an argument to a method, and never used again
– In this case, the object need not be assigned to a variable, i.e., given a name
• An object whose reference is not assigned to a variable is called an anonymous object

A statement that is always true for every object of the class is called a class invariant
– A class invariant can help to define a class in a consistent and organized way
Copy Constructor:

A copy constructor is a constructor with a single argument of the same type as the class
• The copy constructor should create an object that is a separate, independent object, but with the instance variables set so that it is an exact copy of the argument object

Mutable and Immutable Classes:

A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an immutable class
– Objects of such a class are called immutable objects

– It is perfectly safe to return a reference to an immutable
object because the object cannot be changed in any way – The String class is an immutable class

A class that contains public mutator methods or other public methods that can change the data in its objects is called a mutable class, and its objects are called mutable objects
– Never write a method that returns a mutable object
– Instead, use a copy constructor to return a reference to a
completely independent copy of the mutable object

Deep Copy Versus Shallow Copy:

A deep copy of an object is a copy that, with one exception, has no references in common with the original
– Exception: References to immutable objects are allowed to be shared
Any copy that is not a deep copy is called a shallow copy
– This type of copy can cause dangerous privacy leaks in a program

Packages and Import Statements:

Java uses packages to form libraries of classes
A package is a group of classes that have been placed in a directory or folder, and that can be used in any program that includes an import statement that names the package
– The import statement must be located at the beginning of the program file:
Only blank lines, comments, and package statements may precede it
– The program can be in a different directory from the package

Import Statements:

We have already used import statements to include some predefined packages in Java, such as Scanner from the java.util package

```
import java.util.Scanner;
```

It is possible to make all the classes in a package available instead of just one class:

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

– Note that there is no additional overhead for importing the entire package

The package Statement:

To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each class file:
package package_name;
– Only the .class files must be in the directory or
folder, the .java files are optional
– Only blank lines and comments may precede the
package statement
– If there are both import and package statements, the package statement must precede any import statements

The Package java.lang:

The package java.lang contains the classes that are fundamental to Java programming
– It is imported automatically, so no import statement is needed
– Classes made available by java.lang include Math, String, and the wrapper classes

Package Names and Directories:

A package name is the path name for the directory or subdirectories that contain the package classes
Java needs two things to find the directory for a package: the name of the package and the value of the CLASSPATH variable
– The CLASSPATH variable is set equal to the list of directories (including the current directory, ".") in which Java will look for packages on a particular computer
– Java searches this list of directories in order, and uses the first directory on the list in which the package is found
– The CLASSPATH environment variable is similar to the PATH variable, and is set in the same way for a given operating system

CHAPTER 7:

Inheritance is one of the main techniques of object- oriented programming (OOP)
Using this technique, a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
– The specialized classes are said to inherit the methods and instance variables of the general class

Inheritance is the process by which a new class is created from another class
– The new class is called a derived class
– The original class is called the base class
• A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
• Inheritance is especially advantageous because it allows code to be reused, without having to copy it into the definitions of the derived classes

Overriding a Method Definition:

Although a derived class inherits methods from the base class, it can change or override an inherited method if necessary
– In order to override a method definition, a new definition of the method is simply placed in the class definition, just like any other method that is added to the derived class

Changing the Return Type of an Overridden Method:

Ordinarily, the type returned may not be changed when overriding a method
However, if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
This is known as a covariant return type

Do not confuse overriding a method in a derived class with overloading a method name
– When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
– When a method in a derived class has a different signature from the method in the base class, that is overloading

– Note that when the derived class overloads the original method, it still inherits the original method from the base class as well

If the modifier final is placed before the definition of a method, then that method may not be redefined in a derived class
If the modifier final is placed before the definition of a class, then that class may not be used as a base class to derive other classes

The super Constructor:

A derived class uses a constructor from the base class to initialize all the data inherited from the base class
– In order to invoke a constructor from the base class, it uses a special syntax:

```
public derivedClass(int p1, int p2, double p3)
{
  super(p1, p2);
  instanceVariable = p3;
}
```

– In the above example, super(p1, p2); is a call to the base class constructor

A call to the base class constructor can never use the
name of the base class, but uses the keyword super instead
• A call to super must always be the first action taken in a constructor definition
• An instance variable cannot be used as an argument to super

If a derived class constructor does not include an invocation of super, then the no-argument constructor of the base class will automatically be invoked
– This can result in an error if the base class has not defined a no-argument constructor
Since the inherited instance variables should be initialized, and the base class constructor is designed to do that, then an explicit call to super should always be used

Protected and Package Access:

• If a method or instance variable is modified by protected (rather than public or private), then it can be accessed by name
– Inside its own class definition
– Inside any class derived from it

– In the definition of any class in the same package
• The protected modifier provides very weak protection compared to the private modifier
– It allows direct access to any programmer who defines a suitable derived class
– Therefore, instance variables should normally not be marked protected

When considering package access, do not forget the default package
– All classes in the current directory (not belonging to some other package) belong to an unnamed package called the default package
• If a class in the current directory is not in any other package, then it is in the default package
– If an instance variable or method has package access, it can be accessed by name in the definition of any other class in the default package

CHAPTER 8:

There are three main programming mechanisms that constitute object-oriented programming (OOP)
– Encapsulation
– Inheritance
– Polymorphism
• Polymorphism is the ability to associate many meanings to one method name
– It does this through a special mechanism known as late binding or dynamic binding

Inheritance allows a base class to be defined, and other classes derived from it
– Code for the base class can then be used for its own objects, as well as objects of any derived classes
• Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

Upcasting is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)

Downcasting is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)

Abstract Class:

In order to postpone the definition of a method, Java allows an abstract method to be declared
– An abstract method has a heading, but no method body – The body of the method is defined in the derived classes

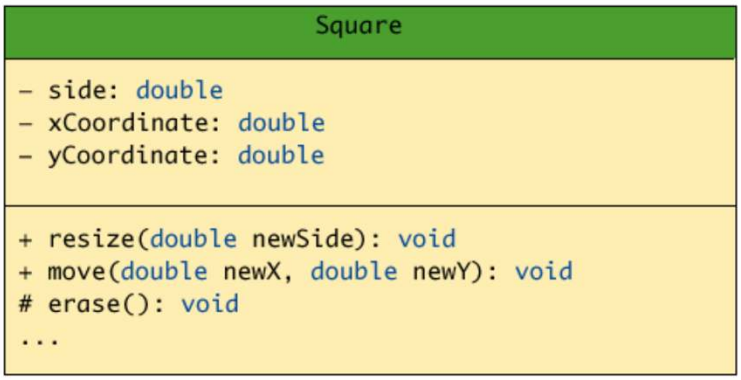
An abstract method is like a place holder for a method that will be fully defined in a descendent class
• It has a complete method heading, to which has been added the modifier abstract
• It cannot be private
• It has no method body, and ends with a semicolon in place of its body
public abstract double getPay();
A class that has no abstract methods is called a concrete class.

CHAPTER 12:

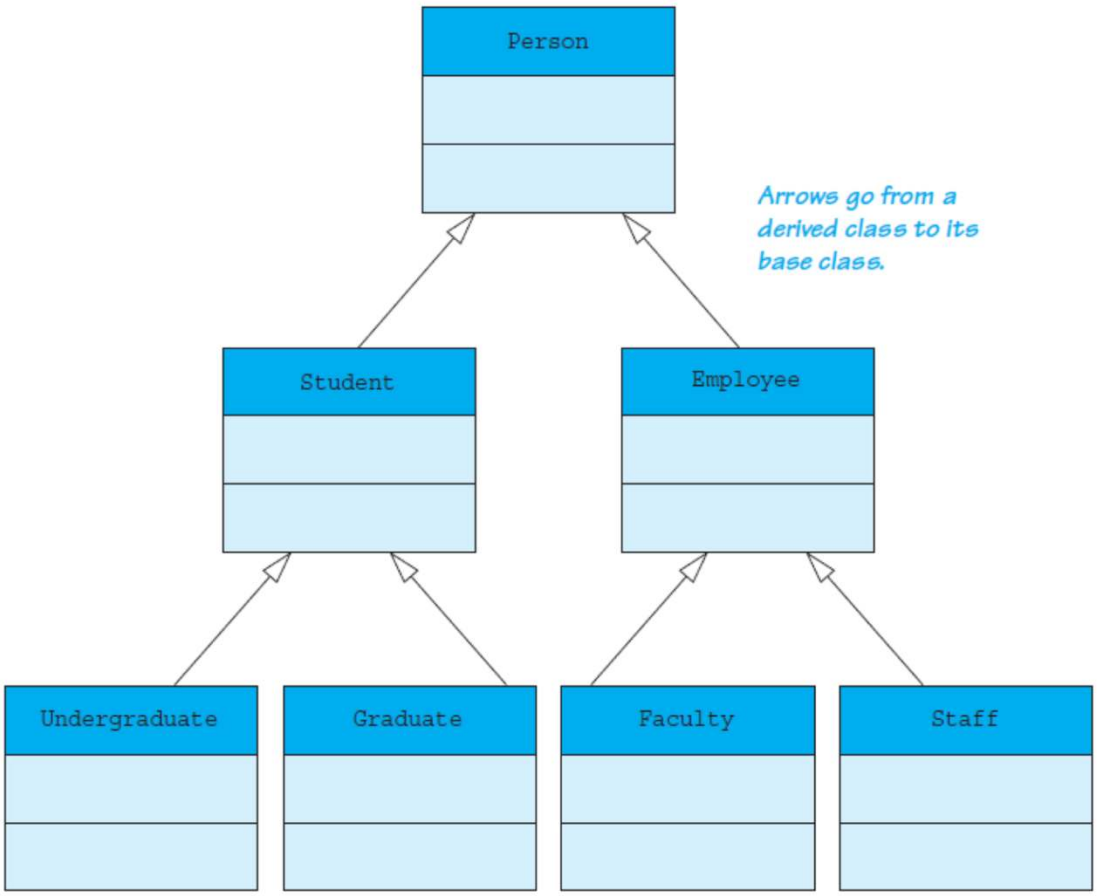
Unified Modeling Language(UML)

A class diagram is divided up into three sections
– The top section contains the class name
– The middle section contains the data specification for the class
– The bottom section contains the actions or methods of the class

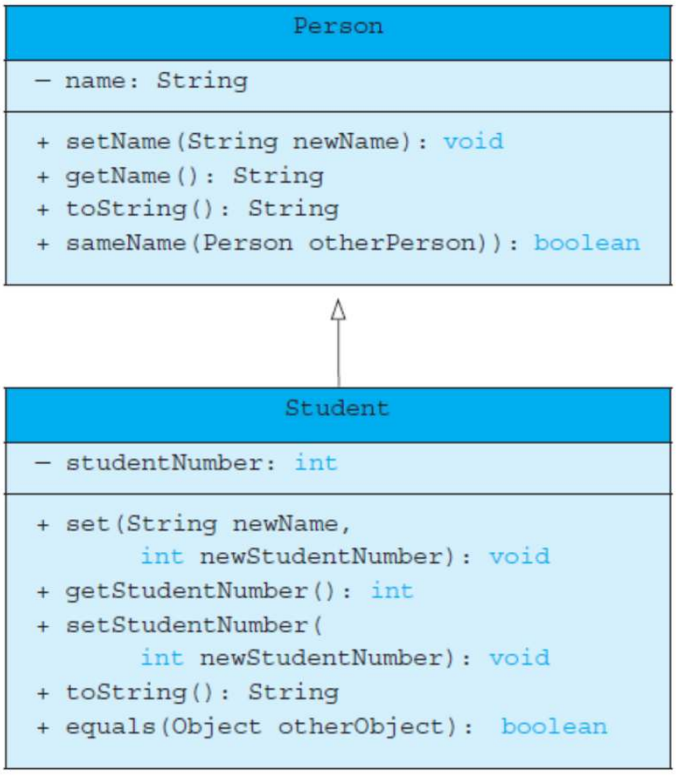
The data specification for each piece of data in a UML diagram consists of its name, followed by a colon, followed by its type
Each name is preceded by a character that specifies its access type:
– A minus sign (-) indicates private access
– A plus sign (+) indicates public access
– A sharp (#) indicates protected access
– A tilde (~) indicates package access



Display 12.2 A Class Hierarchy in UML Notation



Display 12.3 Some Details of a UML Class Hierarchy



The ArrayList Class:

ArrayList is a class in the standard Java libraries
– Unlike arrays, which have a fixed length once they have been created, an ArrayList is an object that can grow and shrink while your program is running
In general, an ArrayList serves the same purpose as an array, except that an ArrayList can change length while the program is running
The class ArrayList is implemented using an array as a private instance variable
– When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

Why not always use an ArrayList instead of an array?
1. An ArrayList is less efficient than an array
2. It does not have the convenient square bracket notation
3. The base type of an ArrayList must be a class type (or other reference type): it cannot be a primitive type

– This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives

In order to make use of the ArrayList class, it must first be imported from the package
java.util
ArrayList<BaseType> aList = new ArrayList<BaseType>();
– Specifying an initial capacity does not limit the size to which an ArrayList can eventually grow
Note that the base type of an ArrayList is specified as a type parameter

Arrays:

The new keyword creates an array object and stores it in memory and then returns the address/reference of the newly created object in the variable on the left of the "=".
An array is a fixed-size data structure that stores a collection of elements of the same type.
Arrays are objects
Use new keyword to dynamically allocate memory • Has one instance variable length.

Creating arrays

• Base Type declares the type of data to store
• Size is the number of elements that will be stored
• This allows java to reserve the amount of memory needed
• All elements of the array have to be of the same type

```
//Option 1
BaseType[] arrayName; arrayName = new BaseType[size];
//Option 2
BaseType[] arrayName = new BaseType[size];
```

Array size
• The size of an array can be set by using a variable allowing the memory to be dynamically allocated at runtime
• Example use case would be to read the size in from user input (Keyboard)
• The length variable is read-only and returns the size of the array

Arrays and loops

• A simple and elegant alternative

• The below for-each line should be read as "for each element in arrayVariable do the following:"

```
for(baseType element : arrayVariable)
{
   //Do something with e
}
```

Privacy Leaks with Array Instance Variables

• Solved by returning a deep copy

```
private double[] grades = new double[5];
public double[] getGradesArray()
{
double[] temp = new double[grades.length]; //Create copy
for(int i=0; i<grades.length; i++)
{
      temp[i] = grades[i];
   }
return temp; //Now we can't access original grades array
}
```

Privacy Leaks with Array Instance Variables

• If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

```
public ClassType[] getArray()
{
ClassType[] temp = new ClassType[count];
    for(int i = 0; i < count; i++)
    {
       temp[i] = new ClassType(someArray[i]);
    }
    return temp;
 }
```

Arrays and the use of = and ==

• Since arrays are objects, the use of = will only copy the memory address stored in the variable
• Does not copy the values of each indexed variable
• Using = will make two variables point to the same array in memory
• Similarly == will only return true if the variables share the same memory address
• Solution is to use Arrays.equals() method for 1D arrays and Arrays.deepEquals() for multidimensional arrays

```
double[][] grades = new double[100][5]; //2D
grades[23].length; //Will return 5
```

Ragged Arrays

• Typically 2D arrays are created with the same number of entries for each row. A ragged array is when different rows have different numbers of columns.
• Essentially, each row in a two-dimensional array need not have the same number of elements.

```
double[][] grades = new double[100][]; grades[0] = new double[10];
grades[1] = new double[14];
grades[2] = new double[50];
```

Enumerated Types

• An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it.
• Because they are constants, the names of an enum type's fields are in uppercase letters.

• In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days- of-the-week enum type as:

```
public enum Day
{
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY
};
```

Arrays – Data Types

- Primitive types
- int,byte,short,long,double,float,boolean,char
- Class Types
- String,Scanner,Arrays,etc. Including your own classes

Arrays – Storing Primitive data

- Declare array
- Int[] arrayName;
- Initialise array by setting its size
- arrayName = new int[3];
- Store values in array
- arrayName[0] = 5;
- arrayName[1] = 2;
- arrayName[2] = 9;

Arrays – Storing Objects
- Declare array
- Student[] arrayName;
- Initialise array by setting its size
- arrayName = new Student[3];
- Store values in array
- arrayName[0] = new Student();

Linked Data Structures

- A linked data structure consists of capsules of data, known as nodes, which are connected together via links
- These links can be thought of as pointers or variables that store the memory address of the next node/capsule.

Linked List

- A linked list is the simplest kind of linked data structure
- It consists of a single chain of nodes

• Each node contains some data and a link to the next node (This link allows nodes to be traversed)
• The first node is called the Head Node
• The last node also acts as an “end” marker

• The last node’s pointer is null allowing us to know when we’ve reached the end of the list.
• All other node’s pointer is set to the address of the next node in the list

Linked Lists vs. Arrays

• Arrays reserve a fixed size of memory before they can be used (Fixed size), whereas linked data structures allocate memory as required during runtime (Dynamic size).
• Linked lists can grow as the number of items increase, unlike arrays. Arrays need to make a new larger array and then copy all data across.
• One disadvantage of a linked list is that it does not offer direct access like an array
• Need to iterate over nodes (starting at the head node and following the links) until the desired item is found
• Linked lists also can be computationally more expensive since each node needs to store a reference to the next node

The Link in a linked List
• The link is simply an instance variable of the same type as the node. This variable stores a reference (memory address) of where in memory the next node is located

• Node class
    • Item to store
    • Link to next node
• Linked List class to manage the nodes
    • Insert method – adds new nodes to the list
    • Delete – deletes a node from the linked list
    • Size – returns the size of the linked list
    • Contains – returns true if an element is in the linked list

Simple Linked List Class

• The head node and allows us to access entire list

• A variable of type Node is used to store the reference to the head node.
• This variable is not the head node, nor is it a node
• It simply contains a reference to the head node

Traversing a Linked List

• A linked list class does not contain all the nodes in the linked list directly, instead as previously noted, it only stores a reference to the head node.
• Because each node has a link variable holding the reference to the next node, we can reach all of the nodes by following these links, starting from the head node.

• We set a temporary variable (position) = head and update it to = link value as we traverse the list.
• How do we know when we've reached the end of the list?
• We can continue traversing until the link variable is null. This signifies the end of the list.
• We use == to test for null

Know if the list is empty

• So the head variable points to the first node in the linked list
• If list is empty, this head variable is set to null
• To test if null, we use ==, not the equals() method!
• Why is this?
• Because the head variable will contain an address to some memory if it points to a node. Else the head variable will contain null.

Adding a node

• A new node is added to the start of the list, making the new node become the first on the list (head).
• The head variable gives the location of the first node on the list
• So the newly created node's link field is set equal to the address of the head node
• And the head is set equal to the new node

Deleting the Head node

• The deleteHeadNode()method removes the first node from the list and sets the head variable to point to the old 2nd node in the list. How is this done?
• The head variable is set equal to the link variable of the first node (The one to be deleted)

Deleting any node

• Conceptually, a node is deleted by setting the previous node to point the node that follows the current node you want to delete
• In technical terms, this is done by setting the Link variable in the previous node equal to the link variable in the node that is to be deleted.
• What happens to the node that is deleted?
• Since nothing points to it anymore, the Automatic Garbage Collection will automatically recycle the memory used by this old node.

Automatic Garbage Collection
• Is a method of automatically reclaiming memory that is occupied by objects which are no longer being used by your program.
• Example: What happens to the memory once you “delete” the node?
• “Delete” since we aren’t explicitly deleting the object from memory, but instead, we are removing any pointers that point to it (The node). And since nothing points to it in memory, it cannot be located anymore, and hence can be considered unused. This unused object/memory is then detected by the garbage collector and then freed up.

Make the Node an Inner Class

• So far the Node class has been an external class from the Linked List class.
• We can make the linked list self-contained by making the node an inner class
• This inner node class should be private unless used elsewhere
• An inner class reduces the need to have accessor and mutator methods
• Since the instance variables are private, they can be accessed directly by methods from the outer class without causing any privacy leaks

A Generic Linked List

• A linked list can be created whose Node class has a type parameter T for the type of data stored in the node

• Therefore, it can hold objects of any class type, including types that contain multiple instance variable
• The type of the actual object is plugged in for the type parameter T
• For the most part, this class can have the same methods, coded in basically the same way, as the previous linked list example
• The only difference is that a type parameter is used instead of an actual type for the data in the node

Pitfall: Using Node instead of Node<T>

• Note: This pitfall is explained by example – any names can be substituted for the node Node and its parameter <T>
• When defining the LinkedList3<T> class, the type for a node is Node<T>, not Node
• If the <T> is omitted, this is an error for which the compiler may or may not issue an error message (depending on the details of the code), and even if it does, the error message may be quite strange
• Look for a missing <T> when a program that uses nodes with type parameters gets a strange error message or doesn't run correctly

A Generic Linked List: the equals Method

• Like other classes, a linked list class should normally have an equals method
• The equals method can be defined in a number of reasonable ways
• Different definitions may be appropriate for different situations
• Two such possibilities are the following:
1. They contain the same data entries (possibly in different orders)
2. They contain the same data entries in the same order
• Of course, the type plugged in for T must also have redefined the equals method

Recap: Linked Lists
• Linked data structure consisting of a single chain of nodes connected by links
• Advantages:
• Linked lists are a dynamic data structure - can grow and shrink, allocating and deallocating memory while the program is running.
• Insertion and deletion node operations are easily implemented in a linked list.
• No need to define an initial size for a linked list.
• Items can be added or removed at any point in the list.

• Disadvantages:
• They use more memory than arrays because of the storage used by their pointers.
• Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access
• Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list

Iterators

• A collection of objects, such as the nodes of a linked list, must often be traversed in order to perform some action on each object
• An iterator is any object that enables a list to be traversed in this way
• A linked list class may be created that has an iterator inner class
• If iterator variables are to be used outside the linked list class, then the iterator class would be made public
• The linked list class would have an iterator method that returns an iterator for its calling object
• Given a linked list named list, this can be done as follows:
LinkedList2.List2Iterator i = list.iterator();

The basic methods used by an iterator are as follows:
• restart: Resets the iterator to the beginning of the list
• hasNext: Determines if there is another data item on the list
• next: Produces the next data item on the list

Adding and Deleting Nodes

• An iterator is normally used to add or delete a node in a linked list
• Given iterator variables position and previous, the following two lines of code will delete the node at location position:

```
previous.link = position.link;
position = position.link;
```

• Note: previous points to the node before position

Variations on a Linked List
• An ordinary linked list allows movement in one direction only
• However, a doubly linked list has one link that references the next node, and one that references
the previous node

• The node class for a doubly linked list can begin as follows:

```
private class TwoWayNode
{
private String item;
private TwoWayNode previous;
 private TwoWayNode next; ...
```

• In addition, the constructors and methods in the doubly linked list class would be modified to accommodate the extra link

Simple Copy Constructors and clone Methods

• There is a simple way to define copy constructors and the clone method for data structures such as linked lists
• Unfortunately, this approach produces only shallow copies
• The private helping method copyOf is used by both the copy constructor and the clone method
• The copy constructor uses copyOf to create a copy of the list of nodes
• The clone method first invokes its superclass clone method, and then uses copyOf to create a clone of the list of nodes

Exceptions

• A generic data structure is likely to have methods that throw exceptions
• Situations such as a null argument to the copy constructor may be handled differently in different situations
• If this happens, it is best to throw a NullPointerException, and let the programmer who is using the linked list handle the exception, rather than take some arbitrary action
• A NullPointerException is an unchecked exception: it need not be caught or declared in a throws clause

A Linked List with a Deep Copy clone Method

• Some of the details of the clone method in the previous linked list class may be puzzling, since the following code would also return a deep copy:

```
  public LinkedList<T> clone()
  {
        return new LInkedList<T>(this);
  }
```

• However, because the class implements PubliclyCloneable which, in turn, extends Cloneable, it must implement the Cloneable interface as specified in the Java documentation

Pitfall: The clone Method Is Protected in Object

• It would have been preferable to clone the data belonging to the list being copied in the copyOf method as follows:
nodeReference = new Node((T)(position.data).clone(), null);
• However, this is not allowed, and this code will not compile
• The error message generated will state that clone is protected in Object
• Although the type used is T, not Object, any class can be plugged in for T
• When the class is compiled, all that Java knows is that T is a descendent class of Object

Tip: Use a Type Parameter Bound for a Better clone

•One solution to this problem is to place a bound on the type parameter T so that it must satisfy a suitable interfaceAlthough there is no standard interface that does this, it is easy to define one
For example, a PubliclyCloneable interface could be defined
Bounded Type Parameters

• There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.
• To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.
• Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

• Why can’t we use the following?
public class LinkedList3<T extends Cloneable> ...{...}
• Because clone() is protected in Object and we need to enforce that it is overridden as public in our class T

```
public interface PubliclyCloneable extends Cloneable
 {
   public Object clone();
 }
```

• So every class that implements this interface has to implement a public clone method! So we use:
public class LinkedList3<T extends PubliclyCloneable> ...{...}

Cloneable Interface in Java

• A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.
• Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
• By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.
• Note that this interface does not contain the clone method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface.

The Stack Data Structure

• A stack data structure is not necessarily a linked data structure, but can be implemented as one
• A stack is a data structure that removes items in the reverse order of which they were inserted (LIFO: Last In First Out)
• A linked list that inserts and deletes only at the head of the list is a stack
• The following class doesn't show it (for simplicity), but it would be appropriate to add methods such as peek(), clone(), equals() and also convert it into a generic class.

The Queue Data Structure

• A queue is a data structure that handles data in a first-in/first-out fashion (FIFO) like a line at a bank
• Customers add themselves to the end of the line and are served from the front of the line
• A queue can be implemented with a linked list
• However, a queue needs a pointer at both the head and tail (the end) of the linked list
• Nodes are removed from the front F (head end), and are added to the back B (tail end)

• Why? Because its more efficient given a one-directional link

Big-O Terminology

- Linear running time:
    - O(N)—directly proportional to input size N
- Quadratic running time:
    - O(N2)
- Logarithmic running time:
    - O(log N)
        - Typically "log base 2"
        - Very fast algorithms!

Efficiency of Linked Lists

- Find method for linked list
    - May have to search entire list
    - On average would expect to search half of the list, or n/2
    - In big-O notation, this is O(n)
- Adding to a linked list
    - When adding to the start we only reassign some references
    - Constant time or O(1)

Comparing Linked List to Array

Array
- Indexing: O(1)
- Add To Start: O(1)
- Add Any Other Position: O(1)
- Memory: Pre-Allocated

Linked List
- Indexing: O(n)
- Add To Start: O(1)
- Add Any Other Position: O(n)
- Memory: Dynamically Allocated

Software Testing

• Software Testing is the process of checking the functionality of an application to ensure it runs as per requirements.
• Black Box vs. White Box

Unit Testing

• Unit testing is the process in which the individual classes or methods of a program are tested in isolation, to ensure that it functions as required.
• Manual Testing or Automated Testing
• Automated:
• Fast
• Less human resources
• More reliable

What is JUnit?

• Junit is a unit testing framework for Java
• It promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented
• Some features include:
    • JUnit is an open source framework
    • Provides annotations to identify test methods
    • Provides assertions for testing expected results
    • Provides test runners for running tests
    • JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

What is a Unit Test Case?

• A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected
• unit test case is characterized by:
• A known input (precondition) and an expected output (post-condition)
• There must be at least two unit test cases for each requirement
• one positive test and one negative test

Basic Annotations

1. @BeforeClass – Run once before any of the test methods in the class. (Static method)

2. @AfterClass – Run once after all the tests in the class have been run. (Static Method)
3. @Before – Run each time before @Test
4. @After – Run each time after @Test
5. @Test – This is the test method to run
6. @Ignore - Used to ignore the test case

JUnit Assertions

• assertEquals ([String message], expected, actual)
• Checks that two primitives/objects are equal.

• assertTrue ([String message], boolean condition)
• Checks that a condition is true.

• assertFalse ([String message], boolean condition)
• Checks that a condition is false.

• assertNotNull ([String message], Object object)
• Checks that an object isn't null.

• assertNull ([String message], Object object)
• Checks that an object is null.

• assertSame ([String message], Object object1, Object object2)
• The assertSame() method tests if two object references point to the same object.

• assertNotSame ([String message], Object object1, Object object2)
• The assertNotSame() method tests if two object references do not point to the same object.

• assertArrayEquals ([String message], expectedArray, resultArray);
• The assertArrayEquals() method will test whether two arrays are equal to each other.

Swing:

• A GUI (graphical user interface) is a windowing system that interacts with the user
• The Java AWT (Abstract Window Toolkit) package is the original Java package for creating GUIs
• The Swing package is an improved version of the AWT
• However, it does not completely replace the AWT
• Some AWT classes are replaced by Swing classes, but other AWT classes are needed when using Swing
• Swing GUIs are designed using a form of object-oriented programming known as event-driven programming

Events:

• Event-driven programming is a programming style that uses a signal- and-response approach to programming
• An event is an object that acts as a signal to another object know as a listener
• The sending of an event is called firing the event
• The object that fires the event is often a GUI component, such as a button that has been clicked

Listeners:

• A listener object performs some action in response to the event
• A given component may have any number of listeners
• Each listener may respond to a different kind of event, or multiple listeners might may respond to the same events

Exception Objects:

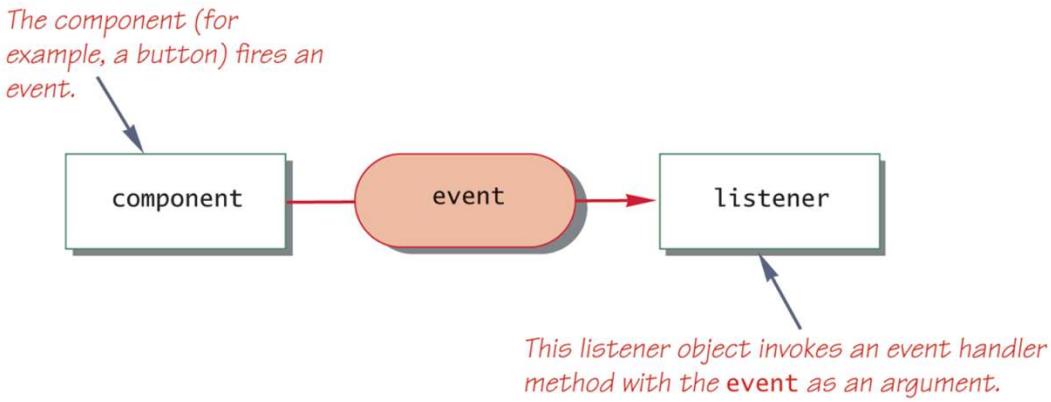
• An exception object is an event
• The throwing of an exception is an example of firing an event
• The listener for an exception object is the catch block that catches the event

Event Handlers:

• A listener object has methods that specify what will happen when events of various kinds are received by it
• These methods are called event handlers
• The programmer using the listener object will define or redefine

these event-handler methods

Display 17.1 **Event Firing and an Event Listener**



Event-Driven Programming:

• In event-driven programming, objects are created that can fire events, and listener objects are created that can react to the events
• The program itself no longer determines the order in which things can happen
• Instead, the events determine the order

• In an event-driven program, the next thing that happens depends on the event that occurs
• In particular, methods are defined that will never be explicitly invoked in any program
• Instead, methods are invoked automatically when an event signals that the method needs to be called

A Simple Window:

• A simple window can consist of an object of the JFrame class
• A JFrame object includes a border and the usual three buttons for minimizing, changing the
size of, and closing the window
• The JFrame class is found in the javax.swing package
JFrame firstWindow = new JFrame();
• A JFrame can have components added to it, such as buttons, menus, and text labels
• These components can be programmed for action firstWindow.add(endButton);

- It can be made visible using the setVisible method firstWindow.setVisible(true);

Display 17.2 A First Swing Demonstration Program

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;

3  public class FirstSwingDemo
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public static void main(String[] args)
8      {
9          JFrame firstWindow = new JFrame();
10         firstWindow.setSize(WIDTH, HEIGHT);
```

This program is not typical of the style we will use in Swing programs.

(continued)

Display 17.2 A First Swing Demonstration Program

```
11         firstWindow.setDefaultCloseOperation(
12                                 JFrame.DO_NOTHING_ON_CLOSE);

13         JButton endButton = new JButton("Click to end program.");
14         EndingListener buttonEar = new EndingListener();
15         endButton.addActionListener(buttonEar);
16         firstWindow.add(endButton);

17         firstWindow.setVisible(true);
18     }
19 }
```

This is the file `FirstSwingDemo.java`.

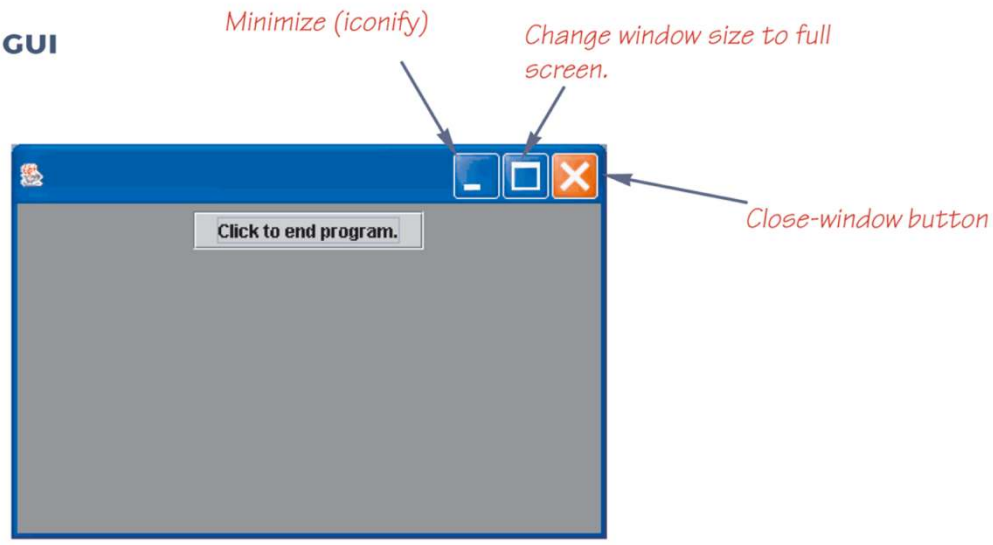
Display 17.2 A First Swing Demonstration Program

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;    This is the file EndingListener.java.

3  public class EndingListener implements ActionListener
4  {
5      public void actionPerformed(ActionEvent e)
6      {
7          System.exit(0);
8      }
9  }
```

(continued)

Display 17.2 A First Swing Demonstration Program

RESULTING GUI



Display 17.3 Some Methods in the Class `JFrame`

The class `JFrame` is in the `javax.swing` package.

`public JFrame()`

Constructor that creates an object of the class `JFrame`.

`public JFrame(String title)`

Constructor that creates an object of the class `JFrame` with the title given as the argument.

(continued)

`public void setDefaultCloseOperation(int operation)`

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:
`JFrame.DO_NOTHING_ON_CLOSE`: Do nothing. The `JFrame` does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 19.)
`JFrame.HIDE_ON_CLOSE`: Hide the frame after invoking any registered `WindowListener` objects.
`JFrame.DISPOSE_ON_CLOSE`: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the program, you use the next constant as an argument to `setDefaultCloseOperation`.
`JFrame.EXIT_ON_CLOSE`: Exit the application using the `System exit` method. (Do not use this for frames in applets. Applets are discussed in Chapter 18.)
If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.
Throws an `IllegalArgumentException` if the argument is not one of the values listed above.[2]
Throws a `SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

`public void setSize(int width, int height)`

Sets the size of the calling frame so that it has the `width` and `height` specified. Pixels are the units of length used.

`public void setTitle(String title)`

Sets the title for this frame to the argument string.

`public void add(Component componentAdded)`

Adds a component to the `JFrame`.

`public void setLayout(LayoutManager manager)`

Sets the layout manager. Layout managers are discussed later in this chapter.

`public void setJMenuBar(JMenuBar menubar)`

Sets the menubar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

`public void dispose()`

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method `dispose` is discussed in Chapter 19.)

Pixels and the Relationship between Resolution and Size:

• A pixel is the smallest unit of space on a screen
• Both the size and position of Swing objects are measured in pixels
• The more pixels on a screen, the greater the screen resolution
• A high-resolution screen of fixed size has many pixels
• Therefore, each one is very small
• A low-resolution screen of fixed size has fewer pixels
• Therefore, each one is much larger
• Therefore, a two-pixel figure on a low-resolution screen will look larger than a two-pixel figure on a high-resolution screen

Pitfall: Forgetting to Program the Close-Window Button:

• The following lines from the FirstSwingDemo program ensure that when the user clicks the close-window button, nothing happens

```
firstWindow.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
```

• If this were not set, the default action would be JFrame.HIDE_ON_CLOSE
• This would make the window invisible and inaccessible, but would not end the program
• Therefore, given this scenario, there would be no way to click the "Click to end program" button
• Note that the close-window and other two accompanying buttons are part of the JFrame object, and not separate buttons

Buttons:

• A button object is created from the class JButton and can be added to a JFrame
• The argument to the JButton constructor is the string that appears on the button when it is displayed

```
JButton endButton = new JButton("Click to end program.");
firstWindow.add(endButton);
```

Action Listeners and Action Events:

Clicking a button fires an event
The event object is "sent" to another object called a listener
• This means that a method in the listener object is invoked automatically
• Furthermore, it is invoked with the event object as its argument

In order to set up this relationship, a GUI program must do two things
1. It must specify, for each button, what objects are its listeners, i.e., it must register the listeners
2. It must define the methods that will be invoked automatically when the event is sent to the listener

```
EndingListener buttonEar = new EndingListener());
endButton.addActionListener(buttonEar);
```

• Above, a listener object named buttonEar is created and registered as a listener for the button named endButton
• Note that a button fires events known as action events, which are handled by listeners known as action listeners
• An action listener is an object whose class implements the ActionListener interface
• The ActionListener interface has one method heading that must be implemented

```
  public void actionPerformed(ActionEvent e)
public void actionPerformed(ActionEvent e)
{
    System.exit(0);
  }
```

• The EndingListener class defines its actionPerformed method as above
• When the user clicks the endButton, an action event is sent to the action listener for that button
• The EndingListener object buttonEar is the action listener for endButton
• The action listener buttonEar receives the action event as the parameter e to its actionPerformed method, which is automatically invoked
• Note that e must be received, even if it is not used

When the actionPerformed method is implemented in an action listener, its header must be the one specified in the ActionListener interface
• It is already determined, and may not be changed
• Not even a throws clause may be added

```
public void actionPerformed(ActionEvent e)
```

• The only thing that can be changed is the name of the parameter, since it is just a placeholder

• Whether it is called e or something else does not matter, as long as it is used consistently within the body of the method

Tip: Ending a Swing Program:

• GUI programs are often based on a kind of infinite loop
• The windowing system normally stays on the screen until the user indicates that it should go
away
• If the user never asks the windowing system to go away, it will never go away
• In order to end a GUI program, System.exit must be used when the user asks to end the program
• It must be explicitly invoked, or included in some library code that is executed
• Otherwise, a Swing program will not end after it has executed all the code in the program

The Normal Way to Define a **`JFrame`** (Part 1 of 4)

Display 17.4 The Normal Way to Define a `JFrame`

```
1  import javax.swing.JFrame;
2  import javax.swing.JButton;


3  public class FirstWindow extends JFrame
4  {
5      public static final int WIDTH = 300;
6      public static final int HEIGHT = 200;

7      public FirstWindow()
8      {
9          super();
10         setSize(WIDTH, HEIGHT);

11         setTitle("First Window Class");
```

(continued)

Display 17.4 The Normal Way to Define a JFrame

```
12          setDefaultCloseOperation(
13                                  JFrame.DO_NOTHING_ON_CLOSE);

14          JButton endButton = new JButton("Click to end program.");
15          endButton.addActionListener(new EndingListener());
16          add(endButton);
17      }
18  }
```

The class `EndingListener` *is defined in Display 17.2.*

This is the file `FirstWindow.java`.

(continued)

Display 17.4 The Normal Way to Define a JFrame

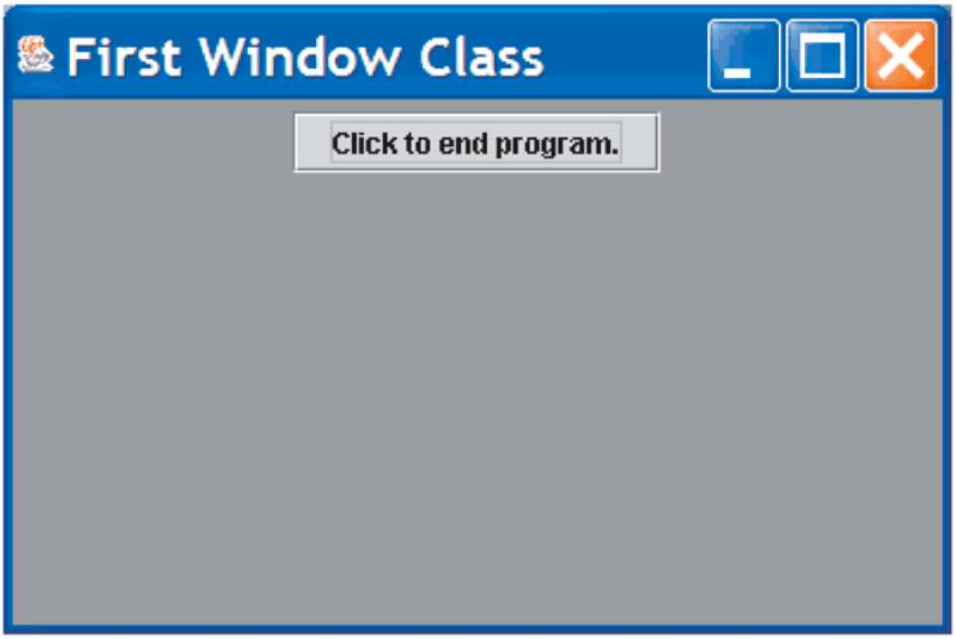
This is the file `DemoWindow.java`.

```
1   public class DemoWindow
2   {
3       public static void main(String[] args)
4       {
5           FirstWindow w = new FirstWindow();
6           w.setVisible(true);
7       }
8   }
```

(continued)

Display 17.4 The Normal Way to Define a JFrame

RESULTING GUI



Labels:

- A label is an object of the class JLabel
- Text can be added to a JFrame using a label
- The text for the label is given as an argument when the JLabel is created
- The label can then be added to a JFrame

```
JLabel greeting = new JLabel("Hello");
add(greeting);
```

Color:

- In Java, a color is an object of the class Color
- The class Color is found in the java.awt package
- There are constants in the Color class that represent a number of basic colors
- A JFrame can not be colored directly
- Instead, a program must color something called the content pane of the JFrame

• Since the content pane is the "inside" of a JFrame, coloring the content pane has the effect of coloring the inside of the JFrame
• Therefore, the background color of a JFrame can be set using the following code: getContentPane().setBackground(Color);

Display 17.5 The Color Constants

```
Color.BLACK
Color.BLUE
Color.CYAN
Color.DARK_GRAY
Color.GRAY
Color.GREEN
Color.LIGHT_GRAY
Color.MAGENTA
Color.ORANGE
Color.PINK
Color.RED
Color.WHITE
Color.YELLOW
```

The class `Color` is in the `java.awt` package.

Display 17.6 A JFrame with Color

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.Color;

4  public class ColoredWindow extends JFrame
5  {
6      public static final int WIDTH = 300;
7      public static final int HEIGHT = 200;

8      public ColoredWindow(Color theColor)
9      {
10         super("No Charge for Color");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

Display 17.6 A JFrame with Color

```
13         getContentPane().setBackground(theColor);

14         JLabel aLabel = new JLabel("Close-window button works.");
15         add(aLabel);
16     }

17     public ColoredWindow()
18     {
19         this(Color.PINK);
20     }
21 }
```

This is an invocation of the other constructor.

This is the file `ColoredWindow.java`.

(continued)

Display 17.6 A JFrame with Color

This is the file `ColoredWindow.java`.

```
1  import java.awt.Color;

2  public class DemoColoredWindow
3  {
4      public static void main(String[] args)
5      {
6          ColoredWindow w1 = new ColoredWindow();
7          w1.setVisible(true);

8          ColoredWindow w2 = new ColoredWindow(Color.YELLOW);
9          w2.setVisible(true);
10     }
11 }
```

(continued)

Containers and Layout Managers:

• Multiple components can be added to the content pane of a JFrame using the add method
• However, the add method does not specify how these components are to be arranged
• To describe how multiple components are to be arranged, a layout manager is used
• There are a number of layout manager classes such as BorderLayout, FlowLayout, and GridLayout
• If a layout manager is not specified, a default layout manager is used

• A BorderLayout manager places the components that are added to a JFrame object into five regions
• These regions are: BorderLayout.NORTH,
BorderLayout.SOUTH,
BorderLayout.EAST,
BorderLayout.WEST,
BorderLayout.Center
• A BorderLayout manager is added to a JFrame using the setLayout method
• For example:
setLayout(new BorderLayout());

Display 17.7 The BorderLayout Manager

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.BorderLayout;

4  public class BorderLayoutJFrame extends JFrame
5  {
6      public static final int WIDTH = 500;
7      public static final int HEIGHT = 400;

8      public BorderLayoutJFrame()
9      {
10         super("BorderLayout Demonstration");
11         setSize(WIDTH, HEIGHT);
12         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

Display 17.7 The BorderLayout Manager

```
13         setLayout(new BorderLayout());

14         JLabel label1 = new JLabel("First label");
15         add(label1, BorderLayout.NORTH);

16         JLabel label2 = new JLabel("Second label");
17         add(label2, BorderLayout.SOUTH);

18         JLabel label3 = new JLabel("Third label");
19         add(label3, BorderLayout.CENTER);
20     }
21 }
```

This is the file `BorderLayoutJFrame.java`.

(continued)

Display 17.7 The BorderLayout Manager

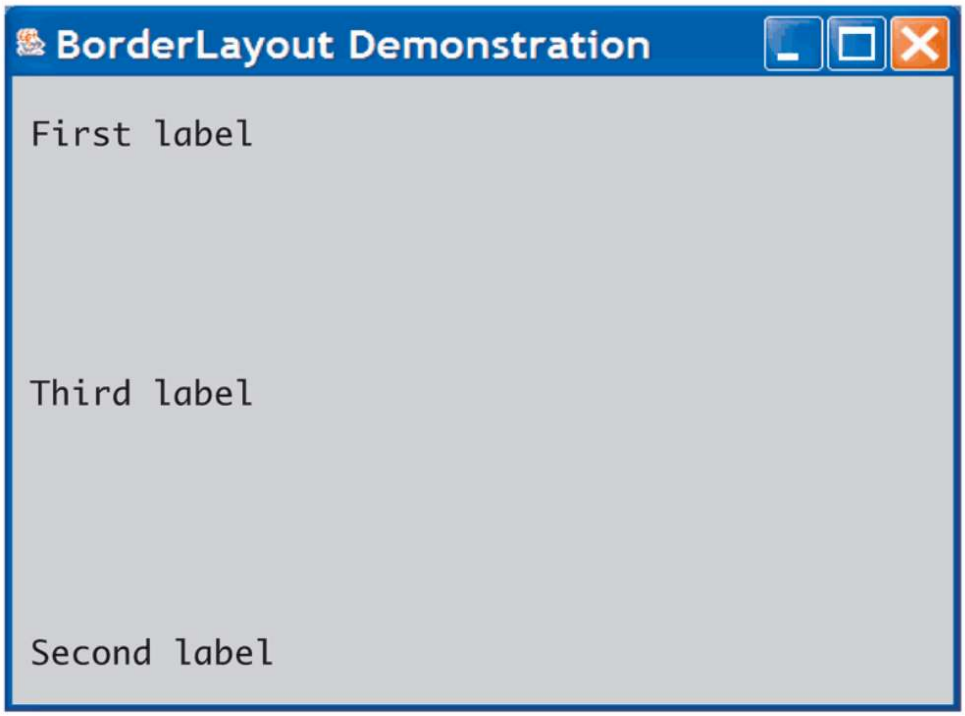
This is the file `BorderLayoutDemo.java`.

```
1  public class BorderLayoutDemo
2  {
3      public static void main(String[] args)
4      {
5          BorderLayoutJFrame gui = new BorderLayoutJFrame();
6          gui.setVisible(true);
7      }
8  }
```

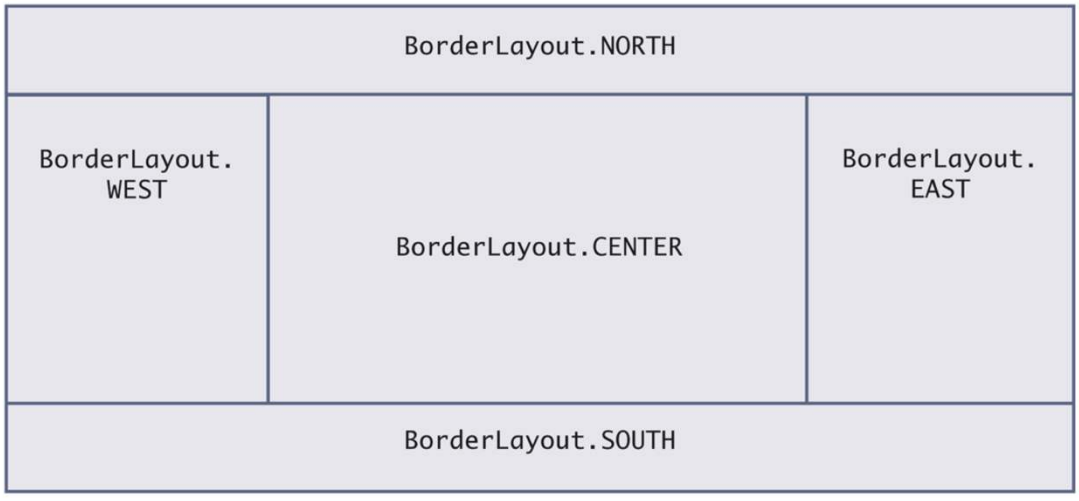
(continued)

Display 17.7 The BorderLayout Manager

RESULTING GUI



Display 17.8 BorderLayout Regions



Flow Layout Managers:

• The FlowLayout manager is the simplest layout manager setLayout(new FlowLayout());
• It arranges components one after the other, going from left to right

• Components are arranged in the order in which they are added
• Since a location is not specified, the add method has only one argument when using the FlowLayoutManager
add.(label1);

Grid Layout Managers:

• A GridLayout manager arranges components in a two-dimensional grid with some number of rows and columns
setLayout(new GridLayout(rows, columns));
• Each entry is the same size
• The two numbers given as arguments specify the number of rows and columns
• Each component is stretched so that it completely fills its grid position

Display 17.9 The GridLayout Manager

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  import java.awt.GridLayout;

4  public class GridLayoutJFrame extends JFrame
5  {
6      public static final int WIDTH = 500;
7      public static final int HEIGHT = 400;

8      public static void main(String[] args)
9      {
10         GridLayoutJFrame gui = new GridLayoutJFrame(2, 3);
11         gui.setVisible(true);
12     }
```

Display 17.9 The GridLayout Manager

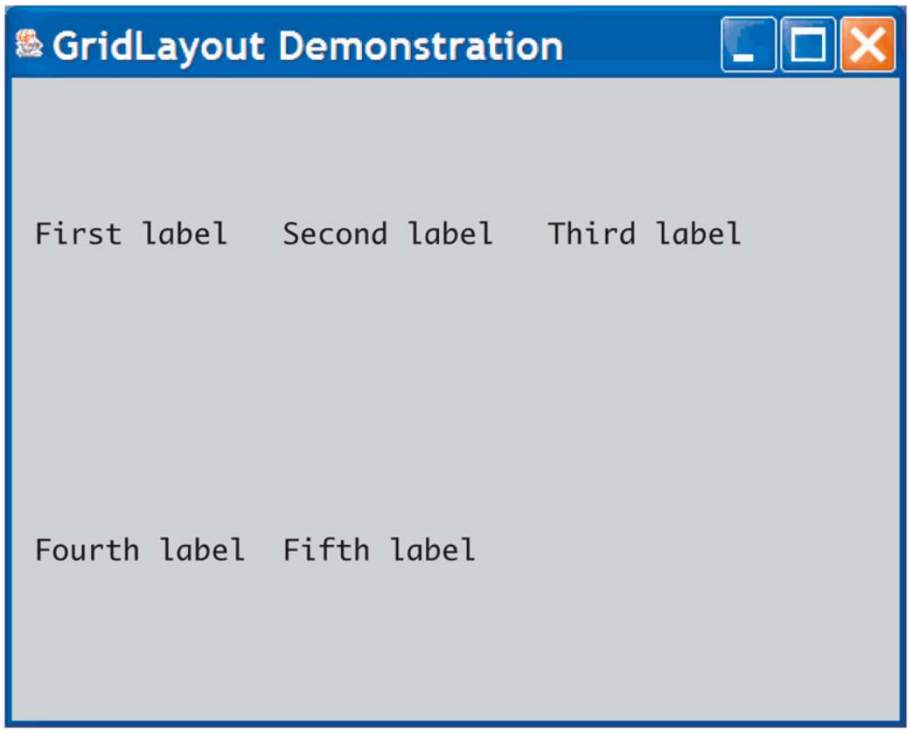
```
13     public GridLayoutJFrame(int rows, int columns )
14     {
15         super();
16         setSize(WIDTH, HEIGHT);
17         setTitle("GridLayout Demonstration");
18         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19         setLayout(new GridLayout(rows, columns ));

20         JLabel label1 = new JLabel("First label");
21         add(label1);
```

Display 17.9 **The GridLayout Manager**

```
22          JLabel label2 = new JLabel("Second label");
23          add(label2);

24          JLabel label3 = new JLabel("Third label");
25          add(label3);

26          JLabel label4 = new JLabel("Fourth label");
27          add(label4);

28          JLabel label5 = new JLabel("Fifth label");
29          add(label5);
30      }
31  }
```

Display 17.9 **The GridLayout Manager**

RESULTING GUI



Display 17.10 **Some Layout Managers**

LAYOUT MANAGER	DESCRIPTION
These layout manager classes are in the `java.awt` package.	
`FlowLayout`	Displays components from left to right in the order in which they are added to the container.
`BorderLayout`	Displays the components in five areas: north, south, east, west, and center. You specify the area a component goes into in a second argument of the add method.
`GridLayout`	Lays out components in a grid, with each component stretched to fill its box in the grid.

Panels:

• A GUI is often organized in a hierarchical fashion, with containers called panels inside other containers
• A panel is an object of the JPanel class that serves as a simple container
• It is used to group smaller objects into a larger component (the panel)
• One of the main functions of a JPanel object is to subdivide a JFrame or other container

• Both a JFrame and each panel in a JFrame can use different layout managers
• Additional panels can be added to each panel, and each panel can have its own layout
manager
• This enables almost any kind of overall layout to be used in a GUI
setLayout(new BorderLayout());
JPanel somePanel = new JPanel(); somePanel.setLayout(new FlowLayout());
• Note in the following example that panel and button objects are given color using the setBackground method without invoking getContentPane
• The getContentPane method is only used when adding color to a JFrame

Display 17.11 Using Panels

```
14      private JPanel redPanel;
15      private JPanel whitePanel;
16      private JPanel bluePanel;

17      public static void main(String[] args)
18      {
19          PanelDemo gui = new PanelDemo();
20          gui.setVisible(true);
21      }

22      public PanelDemo()
23      {
24          super("Panel Demonstration");
25          setSize(WIDTH, HEIGHT);
26          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          setLayout(new BorderLayout());
```

We made these instance variables because we want to refer to them in both the constructor and the method `actionPerformed`.

(continued)

Display 17.11 Using Panels

```
28          JPanel biggerPanel = new JPanel();
29          biggerPanel.setLayout(new GridLayout(1, 3));

30          redPanel = new JPanel();
31          redPanel.setBackground(Color.LIGHT_GRAY);
32          biggerPanel.add(redPanel);

33          whitePanel = new JPanel();
34          whitePanel.setBackground(Color.LIGHT_GRAY);
35          biggerPanel.add(whitePanel);
```

(continued)

Display 17.11 Using Panels

```
36          bluePanel = new JPanel();
37          bluePanel.setBackground(Color.LIGHT_GRAY);
38          biggerPanel.add(bluePanel);

39          add(biggerPanel, BorderLayout.CENTER);

40          JPanel buttonPanel = new JPanel();
41          buttonPanel.setBackground(Color.LIGHT_GRAY);
42          buttonPanel.setLayout(new FlowLayout());

43          JButton redButton = new JButton("Red");
44          redButton.setBackground(Color.RED);
45          redButton.addActionListener(this);
46          buttonPanel.add(redButton);
```

An object of the class `PanelDemo` *is the action listener for the buttons in that object.*

(continued)

Display 17.11 Using Panels

```
47          JButton whiteButton = new JButton("White");
48          whiteButton.setBackground(Color.WHITE);
49          whiteButton.addActionListener(this);
50          buttonPanel.add(whiteButton);

51          JButton blueButton = new JButton("Blue");
52          blueButton.setBackground(Color.BLUE);
53          blueButton.addActionListener(this);
54          buttonPanel.add(blueButton);

55          add(buttonPanel, BorderLayout.SOUTH);
56      }
```

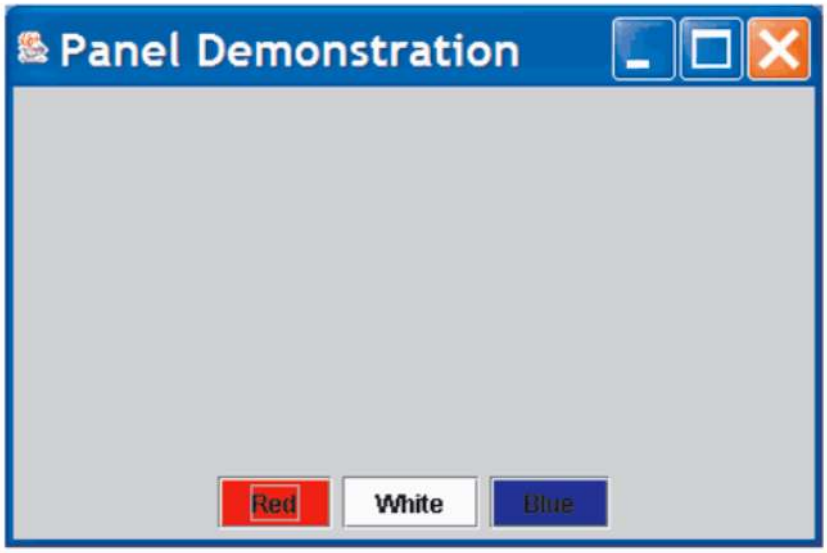
Display 17.11 Using Panels

```
57      public void actionPerformed(ActionEvent e)
58      {
59          String buttonString = e.getActionCommand();

60          if (buttonString.equals("Red"))
61               redPanel.setBackground(Color.RED);
62          else if (buttonString.equals("White"))
63               whitePanel.setBackground(Color.WHITE);
64          else if (buttonString.equals("Blue"))
65               bluePanel.setBackground(Color.BLUE);
66          else
67              System.out.println("Unexpected error.");
68      }
69  }
```

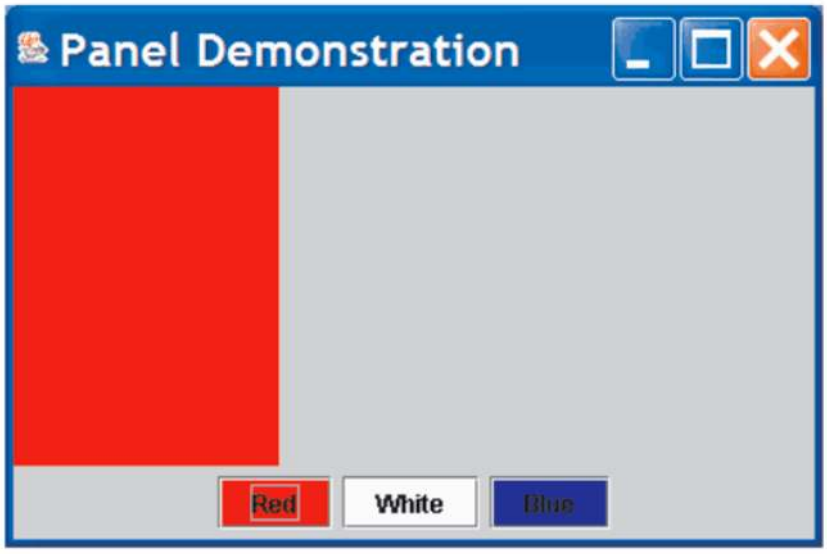
(continued)

Display 17.11 Using Panels

RESULTING GUI (When first run)

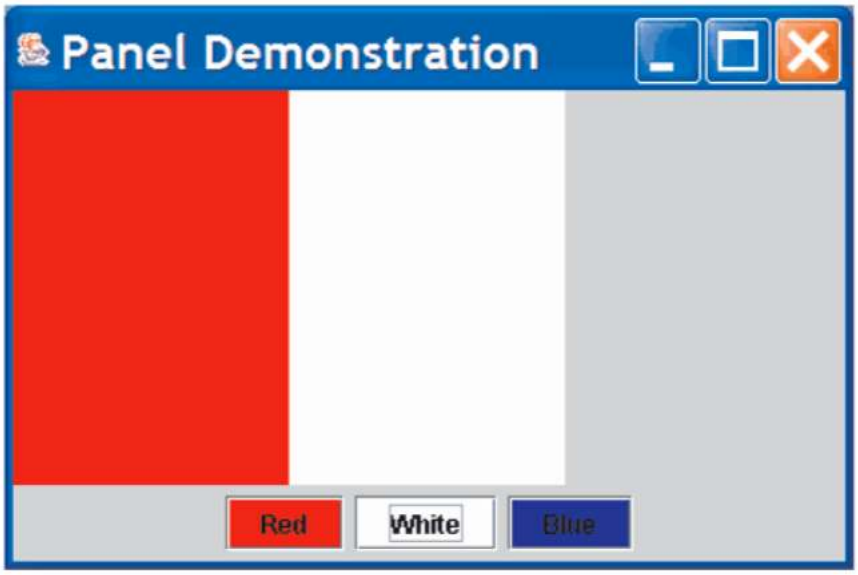


RESULTING GUI (After clicking Red button)

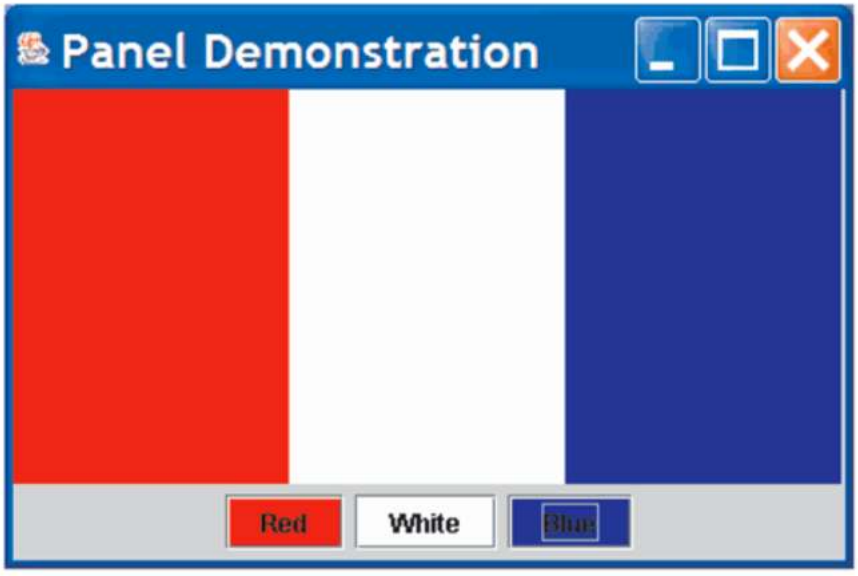


Display 17.11 Using Panels

RESULTING GUI (After clicking White button)



RESULTING GUI (After clicking Blue button)



The Container Class:

• Any class that is a descendent class of the class Container is considered to be a container class
• The Container class is found in the java.awt package, not in the Swing library

• Any object that belongs to a class derived from the Container class (or its descendents) can have components added to it
• The classes JFrame and JPanel are descendent classes of the class Container
• Therefore they and any of their descendents can serve as a container

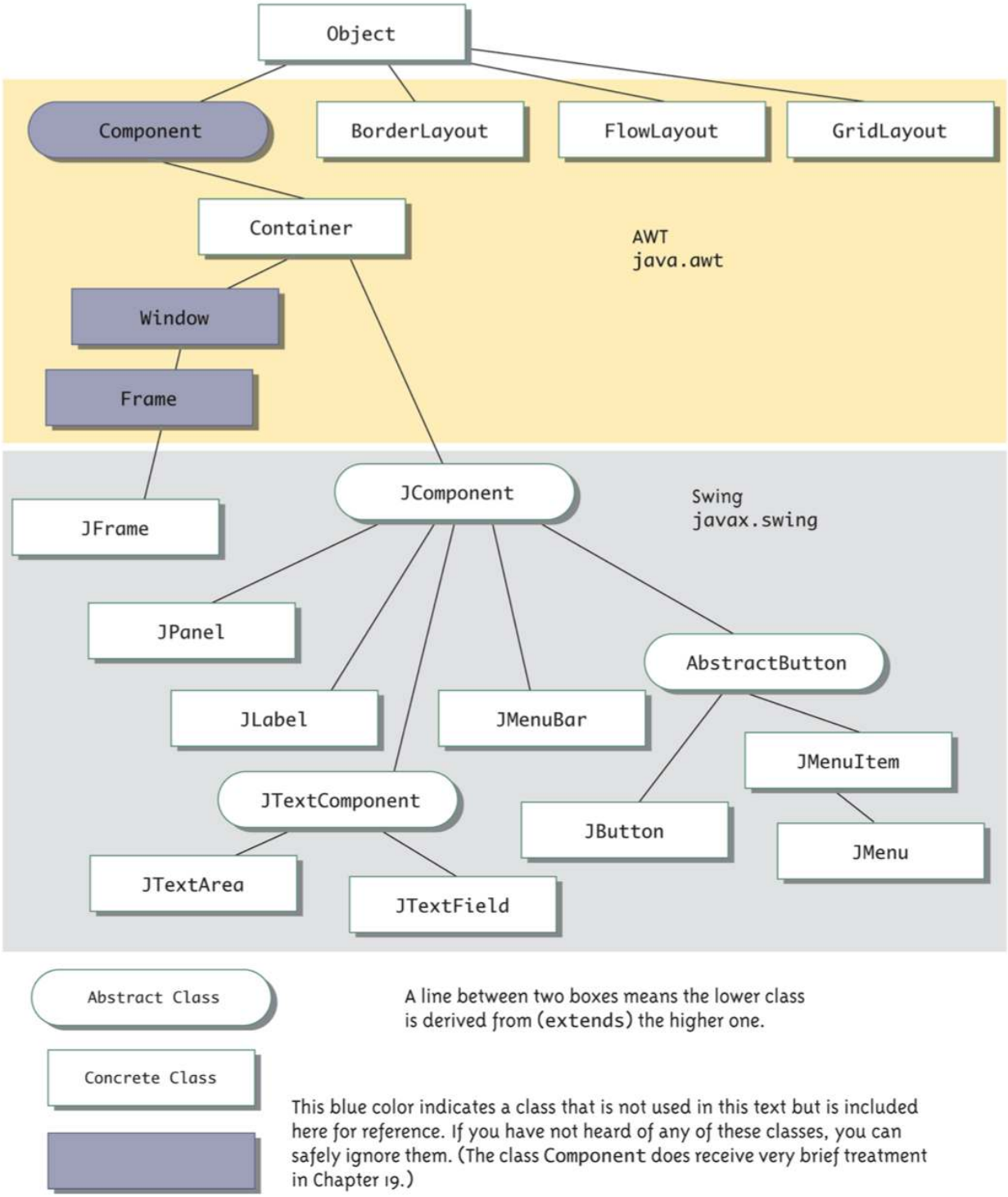
The JComponent Class:

• Any descendent class of the class JComponent is called a component class
• Any JComponent object or component can be added to any container class object
• Because it is derived from the class Container, a JComponent can also be added to another JComponent

Objects in a Typical GUI:

Almost every GUI built using Swing container classes will be made up of three kinds of objects:
1. The container itself, probably a panel or window-like object
2. The components added to the container such as labels, buttons, and panels
3. A layout manager to position the components inside the container

Display 17.12 **Hierarchy of Swing and AWT Classes**



Tip: Code a GUI's Look and Actions Separately:

The task of designing a Swing GUI can be divided into two main subtasks:
1. Designing and coding the appearance of the GUI on the screen
2. Designing and coding the actions performed in response to user actions

In particular, it is useful to implement the actionPerformed method as a stub, until the GUI looks the way it should
public void actionPerformed(ActionEvent e) {}
This philosophy is at the heart of the technique used by the Model-View-Controller pattern

Menu Bars, Menus, and Menu Items:

• A menu is an object of the class JMenu
• A choice on a menu is called a menu item, and is an object of the class JMenuItem
• A menu can contain any number of menu items
• A menu item is identified by the string that labels it, and is displayed in the order to which it was added to the menu
• The add method is used to add a menu item to a menu in the same way that a component is added to a container object

The following creates a new menu, and then adds a menu item to it
JMenu diner = new JMenu("Daily Specials");
JMenuItem lunch = new JMenuItem("Lunch Specials");
lunch.addActionListener(this);
diner.add(lunch);
• Note that the this parameter has been registered as an action listener for the menu item

Nested Menus:

• The class JMenu is a descendent of the JMenuItem class
• Every JMenu can be a menu item in another menu
• Therefore, menus can be nested
• Menus can be added to other menus in the same way as menu items

Menu Bars and JFrame:

• A menu bar is a container for menus, typically placed near the top of a windowing interface
• The add method is used to add a menu to a menu bar in the same way that menu items are added to a menu
JMenuBar bar = new JMenuBar();

bar.add(diner);

• The menu bar can be added to a JFrame in two different ways
1. Using the setJMenuBar method setJMenuBar(bar);
2. Using the add method – which can be used to add a menu bar to a JFrame or any other container

Display 17.14 A GUI with a Menu

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import java.awt.GridLayout;
4  import java.awt.Color;
5  import javax.swing.JMenu;
6  import javax.swing.JMenuItem;
7  import javax.swing.JMenuBar;
8  import java.awt.event.ActionListener;
9  import java.awt.event.ActionEvent;
```

Display 17.14 A GUI with a Menu

```
10 public class MenuDemo extends JFrame implements ActionListener
11 {
12     public static final int WIDTH = 300;
13     public static final int HEIGHT = 200;

14     private JPanel redPanel;
15     private JPanel whitePanel;
16     private JPanel bluePanel;

17     public static void main(String[] args)
18     {
19         MenuDemo gui = new MenuDemo();
20         gui.setVisible(true);
21     }
```

Display 17.14 A GUI with a Menu

```
22      public MenuDemo()
23      {
24          super("Menu Demonstration");
25          setSize(WIDTH, HEIGHT);
26          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27          setLayout(new GridLayout(1, 3));

28          redPanel = new JPanel();
29          redPanel.setBackground(Color.LIGHT_GRAY);
30          add(redPanel);

31          whitePanel = new JPanel();
32          whitePanel.setBackground(Color.LIGHT_GRAY);
33          add(whitePanel);
```

Display 17.14 A GUI with a Menu

```
34          bluePanel = new JPanel();
35          bluePanel.setBackground(Color.LIGHT_GRAY);
36          add(bluePanel);

37          JMenu colorMenu = new JMenu("Add Colors");

38          JMenuItem redChoice = new JMenuItem("Red");
39          redChoice.addActionListener(this);
40          colorMenu.add(redChoice);

41          JMenuItem whiteChoice = new JMenuItem("White");
42          whiteChoice.addActionListener(this);
43          colorMenu.add(whiteChoice);
```

Display 17.14 A GUI with a Menu

```
44          JMenuItem blueChoice = new JMenuItem("Blue");
45          blueChoice.addActionListener(this);
46          colorMenu.add(blueChoice);

47          JMenuBar bar = new JMenuBar();
48          bar.add(colorMenu);
49          setJMenuBar(bar);
50      }
```

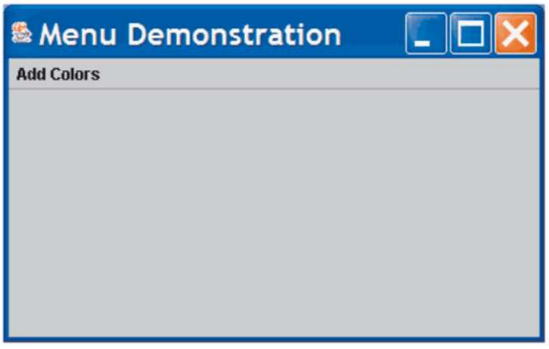
The definition of `actionPerformed` *is identical to the definition given in Display 17.11 for a similar GUI using buttons instead of menu items.*

Display 17.14 A GUI with a Menu

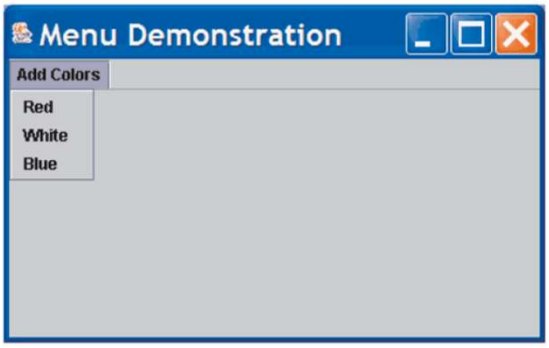
```
51      public void actionPerformed(ActionEvent e)
52      {
53          String buttonString = e.getActionCommand();

54          if (buttonString.equals("Red"))
55               redPanel.setBackground(Color.RED);
56          else if (buttonString.equals("White"))
57              whitePanel.setBackground(Color.WHITE);
58          else if (buttonString.equals("Blue"))
59              bluePanel.setBackground(Color.BLUE);
60          else
61              System.out.println("Unexpected error.");
62      }
63  }
```

Display 17.14 A GUI with a Menu

RESULTING GUI

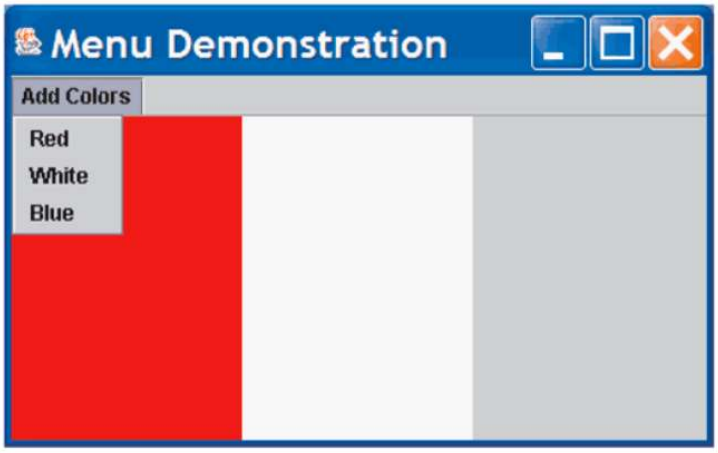


RESULTING GUI (after clicking Add Colors in the menu bar)



Display 17.14 A GUI with a Menu

RESULTING GUI (after choosing Red and White on the menu)



RESULTING GUI (after choosing all the colors on the menu)

The AbstractButton and Dimension Classes:

• The classes JButton and JMenuItem are derived classes of the abstract class named AbstractButton
• All of their basic properties and methods are inherited from the class AbstractButton
• Objects of the Dimension class are used with buttons, menu items, and other objects to specify a size
• The Dimension class is in the package java.awt Dimension(int width, int height)

• Note: width and height parameters are in pixels

The setActionCommand Method:

• When a user clicks a button or menu item, an event is fired that normally goes to one or more action listeners
• The action event becomes an argument to an actionPerformed method
• This action event includes a String instance variable called the action command for the
button or menu item
• The default value for this string is the string written on the button or the menu item
• This string can be retrieved with the getActionCommand method
e.getActionCommand()

The setActionCommand Method:

• The setActionCommand method can be used to change the action command for a component
• This is especially useful when two or more buttons or menu items have the same default action command strings

```
JButton nextButton = new JButton("Next");
nextButton.setActionCommand("Next Button");
JMenuItem choose = new JMenuItem("Next");
choose.setActionCommand("Next Menu Item");
```

Listeners as Inner Classes:

• Often, instead of having one action listener object deal with all the action events in a GUI, a separate ActionListener class is created for each button or menu item
• Each button or menu item has its own unique action listener
• There is then no need for a multiway if-else statement
• When this approach is used, each class is usually made a private inner class

Display 17.15 Some Methods in the Class `AbstractButton`

The abstract class `AbstractButton` is in the `javax.swing` package.
All of these methods are inherited by both of the classes `JButton` and `JMenuItem`.

```
public void setBackground(Color theColor)
```

Sets the background color of this component.

```
public void addActionListener(ActionListener listener)
```

Adds an `ActionListener`.

```
public void removeActionListener(ActionListener listener)
```

Removes an `ActionListener`.

```
public void setActionCommand(String actionCommand)
```

Sets the action command.

Display 17.15 Some Methods in the Class `AbstractButton`

```
public String getActionCommand()
```

Returns the action command for this component.

```
public void setText(String text)
```

Makes text the only text on this component.

```
public String getText()
```

Returns the text written on the component, such as the text on a button or the string for a menu item.

```
public void setPreferredSize(Dimension preferredSize)
```

Sets the preferred size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to use the preferred size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setPreferredSize(
                        new Dimension(int width, int height))
```

Display 17.15 Some Methods in the Class `AbstractButton`

```
public void setMaximumSize(Dimension maximumSize)
```

Sets the maximum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this maximum size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMaximumSize(
                        new Dimension(int width, int height))
```

```
public void setMinimumSize(Dimension minimumSize)
```

Sets the minimum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this minimum size.
Although we do not discuss the `Dimension` class, the following special case is intuitively clear and will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMinimumSize(
                       new Dimension(int width, int height))
```

Display 17.16 Listeners as Inner Classes

<Import statements are the same as in Display 17.14.>

```
1  public class InnerListenersDemo extends JFrame
2  {
3      public static final int WIDTH = 300;
4      public static final int HEIGHT = 200;

5      private JPanel redPanel;
6      private JPanel whitePanel;
7      private JPanel bluePanel;
```

```
private class RedListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        redPanel.setBackground(Color.RED);
    }
} //End of RedListener inner class
```

```
private class WhiteListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        whitePanel.setBackground(Color.WHITE);
    }
} //End of WhiteListener inner class
```

Display 17.16 **Listeners as Inner Classes**

```
22      private class BlueListener implements ActionListener
23      {
24          public void actionPerformed(ActionEvent e)
25          {
26              bluePanel.setBackground(Color.BLUE);
27          }
28      } //End of BlueListener inner class

29      public static void main(String[] args)
30      {
31          InnerListenersDemo gui = new InnerListenersDemo();
32          gui.setVisible(true);
33      }
```

Display 17.16 Listeners as Inner Classes

The resulting GUI is the same as in Display 17.14.

```
34      public InnerListenersDemo()
35      {
36          super("Menu Demonstration");
37          setSize(WIDTH, HEIGHT);
38          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
39          setLayout(new GridLayout(1, 3));

40          redPanel = new JPanel();
41          redPanel.setBackground(Color.LIGHT_GRAY);
42          add(redPanel);

43          whitePanel = new JPanel();
44          whitePanel.setBackground(Color.LIGHT_GRAY);
45          add(whitePanel);
```

(continued)

Display 17.16 Listeners as Inner Classes

```
46          bluePanel = new JPanel();
47          bluePanel.setBackground(Color.LIGHT_GRAY);
48          add(bluePanel);

49          JMenu colorMenu = new JMenu("Add Colors");

50          JMenuItem redChoice = new JMenuItem("Red");
51          redChoice.addActionListener(new RedListener());
52          colorMenu.add(redChoice);
```

(continued)

Display 17.16 Listeners as Inner Classes

```
53          JMenuItem whiteChoice = new JMenuItem("White");
54          whiteChoice.addActionListener(new WhiteListener());
55          colorMenu.add(whiteChoice);

56          JMenuItem blueChoice = new JMenuItem("Blue");
57          blueChoice.addActionListener(new BlueListener());
58          colorMenu.add(blueChoice);

59          JMenuBar bar = new JMenuBar();
60          bar.add(colorMenu);
61          setJMenuBar(bar);
62      }

63  }
```

Text Fields:

• A text field is an object of the class JTextField
• It is displayed as a field that allows the user to enter a single line of text

```
private JTextField name;
name = new JTextField(NUMBER_OF_CHAR);
```

• In the text field above, at least NUMBER_OF_CHAR characters can be visible

• There is also a constructor with one additional String parameter for displaying an initial String in the text field

```
JTextField name = new JTextField("Enter name here.", 30);
```

• A Swing GUI can read the text in a text field using the getText method

```
String inputString = name.getText();
```

• The method setText can be used to display a new text string in a text field

```
name.setText("This is some output");
```

Display 17.17 A Text Field

```
1   import javax.swing.JFrame;
2   import javax.swing.JTextField;
3   import javax.swing.JPanel;
4   import javax.swing.JLabel;
5   import javax.swing.JButton;
6   import java.awt.GridLayout;
7   import java.awt.BorderLayout;
8   import java.awt.FlowLayout;
9   import java.awt.Color;
10  import java.awt.event.ActionListener;
11  import java.awt.event.ActionEvent;
```

Display 17.17 A Text Field

```
12  public class TextFieldDemo extends JFrame
13                              implements ActionListener
14  {
15      public static final int WIDTH = 400;
16      public static final int HEIGHT = 200;
17      public static final int NUMBER_OF_CHAR = 30;

18      private JTextField name;


19      public static void main(String[] args)
20      {
21          TextFieldDemo gui = new TextFieldDemo();
22          gui.setVisible(true);
23      }
```

Display 17.17 A Text Field

```
24      public TextFieldDemo()
25      {
26          super("Text Field Demo");
27          setSize(WIDTH, HEIGHT);
28          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29          setLayout(new GridLayout(2, 1));

30          JPanel namePanel = new JPanel();
31          namePanel.setLayout(new BorderLayout());
32          namePanel.setBackground(Color.WHITE);

33          name = new JTextField(NUMBER_OF_CHAR);
```

Display 17.17 A Text Field

```
34          namePanel.add(name, BorderLayout.SOUTH);
35          JLabel nameLabel = new JLabel("Enter your name here:");
36          namePanel.add(nameLabel, BorderLayout.CENTER);

37          add(namePanel);

38          JPanel buttonPanel = new JPanel();
39          buttonPanel.setLayout(new FlowLayout());
40          buttonPanel.setBackground(Color.PINK);
41          JButton actionButton = new JButton("Click me");
42          actionButton.addActionListener(this);
43          buttonPanel.add(actionButton);

44          JButton clearButton = new JButton("Clear");
45          clearButton.addActionListener(this);
46          buttonPanel.add(clearButton);
```

(continued)

Display 17.17 A Text Field

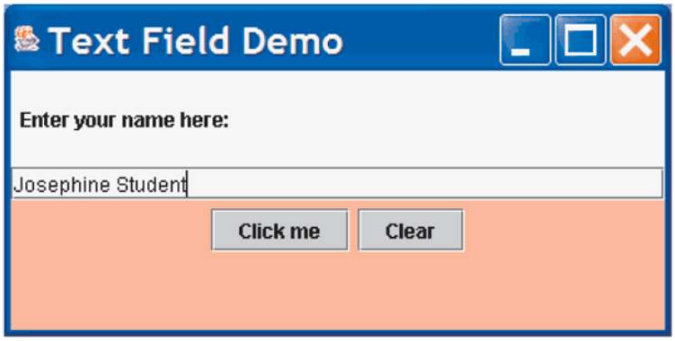
```
47          add(buttonPanel);
48      }

49      public void actionPerformed(ActionEvent e)
50      {
51          String actionCommand = e.getActionCommand();

52          if (actionCommand.equals("Click me"))
53              name.setText("Hello " + name.getText());
54          else if (actionCommand.equals("Clear"))
55              name.setText("");
56          else
57              name.setText("Unexpected error.");
58      }
59  }
```

This sets the text field equal to the empty string, which makes it blank.

(continued)

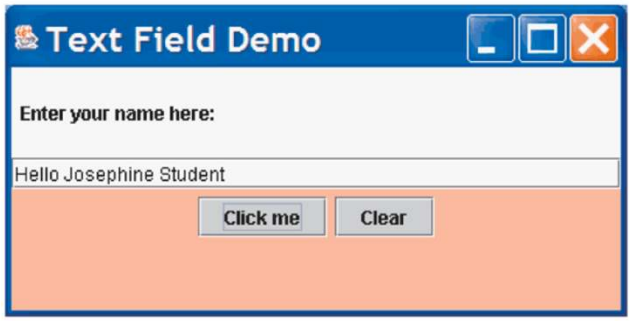
Display 17.17 A Text Field

RESULTING GUI (When program is started and a name entered)



Display 17.17 A Text Field

RESULTING GUI (After clicking the "Click me" button)



Text Areas:

• A text area is an object of the class JTextArea
• It is the same as a text field, except that it allows multiple lines
• Two parameters to the JTextArea constructor specify the minimum number of lines, and the minimum number of characters per line that are guaranteed to be visible

```
JTextArea theText = new JTextArea(5,20);
```

• Another constructor has one addition String parameter for the string initially displayed in
the text area

```
JTextArea theText = new JTextArea("Enter\ntext here." 5, 20);
```

• The line-wrapping policy for a JTextArea can be set using the method setLineWrap
• The method takes one boolean type argument
• If the argument is true, then any additional characters at the end of a line will appear on the following line of the text area
• If the argument is false, the extra characters will remain on the same line and not be visible

```
theText.setLineWrap(true);
```

Text Fields and Text Areas:

• A JTextField or JTextArea can be set so that it can not be changed by the user

```
theText.setEditable(false);
```

• This will set theText so that it can only be edited by the GUI program, not the user
• To reverse this, use true instead (this is the default)

```
theText.setEditable(true);
```

Numbers of Characters Per Line:

• The number of characters per line for a JTextField or JTextArea object is the number of em spaces
• An em space is the space needed to hold one uppercase letter M
• The letter M is the widest letter in the alphabet
• A line specified to hold 20 M 's will almost always be able to hold more than 20 characters

Tip: Inputting and Outputting Numbers:

• When attempting to input numbers from any Swing GUI, input text must be converted to numbers
• If the user enters the number 42 in a JTextField, the program receives the string "42" and must convert it to the integer 42
• The same thing is true when attempting to output a number
• In order to output the number 42, it must first be converted to the string "42"

The Class JTextComponent:

• Both JTextField and JTextArea are derived classes of the abstract class JTextComponent
• Most of their methods are inherited from JTextComponent and have the same meanings
• Except for some minor redefinitions to account for having just one line or multiple lines

Display 17.18 Some Methods in the Class `JTextComponent`

All these methods are inherited by the classes `JTextField` and `JTextArea`.
The abstract class `JTextComponent` is in the package `javax.swing.text`. The classes `JTextField` and `JTextArea` are in the package `javax.swing`.

```
public String getText()
```

Returns the text that is displayed by this text component.

```
public boolean isEditable()
```

Returns `true` if the user can write in this text component. Returns `false` if the user is not allowed to write in this text component.

Display 17.18 Some Methods in the Class `JTextComponent`

```
public void setBackground(Color theColor)
```

Sets the background color of this text component.

```
public void setEditable(boolean argument)
```

If `argument` is `true`, then the user is allowed to write in the text component. If `argument` is `false`, then the user is not allowed to write in the text component.

```
public void setText(String text)
```

Sets the text that is displayed by this text component to be the specified `text`.

Display 17.19 A Simple Calculator

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextField;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import java.awt.BorderLayout;
7  import java.awt.FlowLayout;
8  import java.awt.Color;
9  import java.awt.event.ActionListener;
10 import java.awt.event.ActionEvent;
```

Display 17.19 A Simple Calculator

```
11 /**
12  A simplified calculator.
13  The only operations are addition and subtraction.
14 */
15 public class Calculator extends JFrame
16                          implements ActionListener
17 {
18     public static final int WIDTH = 400;
19     public static final int HEIGHT = 200;
20     public static final int NUMBER_OF_DIGITS = 30;

21     private JTextField ioField;
22     private double result = 0.0;

23     public static void main(String[] args)
24     {
25         Calculator aCalculator = new Calculator();
26         aCalculator.setVisible(true);
27     }
```

Display 17.19 A Simple Calculator

```
28      public Calculator()
29      {
30          setTitle("Simplified Calculator");
31          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
32          setSize(WIDTH, HEIGHT);
33          setLayout(new BorderLayout());

34          JPanel textPanel = new JPanel();
35          textPanel.setLayout(new FlowLayout());
36          ioField =
37               new JTextField("Enter numbers here.", NUMBER_OF_DIGITS);
38          ioField.setBackground(Color.WHITE);
39          textPanel.add(ioField);
40          add(textPanel, BorderLayout.NORTH);
```

(continued)

Display 17.19 A Simple Calculator

```
41          JPanel buttonPanel = new JPanel();
42          buttonPanel.setBackground(Color.BLUE);
43          buttonPanel.setLayout(new FlowLayout());

44          JButton addButton = new JButton("+");
45          addButton.addActionListener(this);
46          buttonPanel.add(addButton);
47          JButton subtractButton = new JButton("-");
48          subtractButton.addActionListener(this);
49          buttonPanel.add(subtractButton);
50          JButton resetButton = new JButton("Reset");
51          resetButton.addActionListener(this);
52          buttonPanel.add(resetButton);

53          add(buttonPanel, BorderLayout.CENTER);
54      }
```

Display 17.19 A Simple Calculator

```
55      public void actionPerformed(ActionEvent e)
56      {
57          try
58          {
59              assumingCorrectNumberFormats(e);
60          }
61          catch (NumberFormatException e2)
62          {
63              ioField.setText("Error: Reenter Number.");
64          }
65      }
```

A `NumberFormatException` *does not need to be declared or caught in a* `catch` *block.*

Display 17.19 A Simple Calculator

```
66      //Throws NumberFormatException.
67      public void assumingCorrectNumberFormats(ActionEvent e)
68      {
69          String actionCommand = e.getActionCommand();

70          if (actionCommand.equals("+"))
71          {
72              result = result + stringToDouble(ioField.getText());
73              ioField.setText(Double.toString(result));
74          }
75          else if (actionCommand.equals("-"))
76          {
77              result = result - stringToDouble(ioField.getText());
78              ioField.setText(Double.toString(result));
```

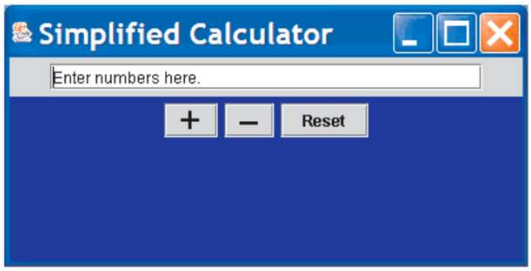
Display 17.19 A Simple Calculator

```
79          }
80          else if (actionCommand.equals("Reset"))
81          {
82              result = 0.0;
83              ioField.setText("0.0");
84          }
85          else
86              ioField.setText("Unexpected error.");
87      }
```

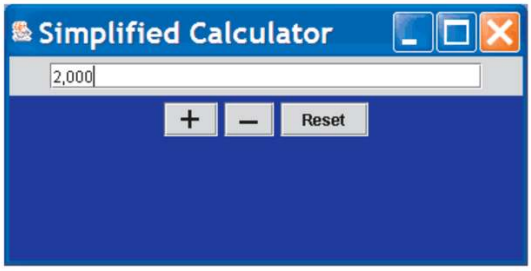
```
88      //Throws NumberFormatException.
89      private static double stringToDouble(String stringObject)
90      {
91          return Double.parseDouble(stringObject.trim());
92      }

93  }
```

Display 17.19 **A Simple Calculator**

RESULTING GUI (When started)

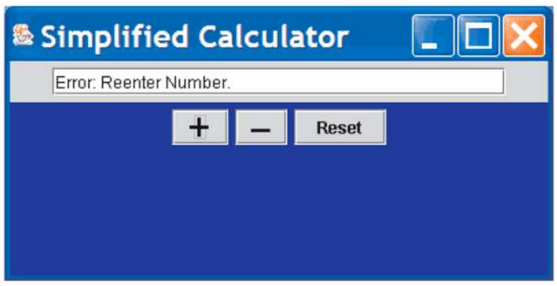


RESULTING GUI (After entering 2,000)

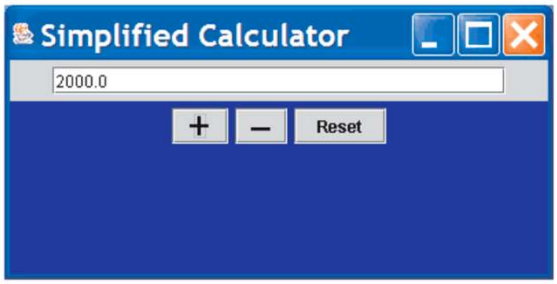


Display 17.19 **A Simple Calculator**

RESULTING GUI (After clicking +)

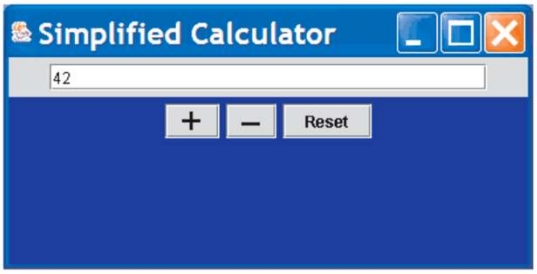


RESULTING GUI (After entering 2000 and clicking +)

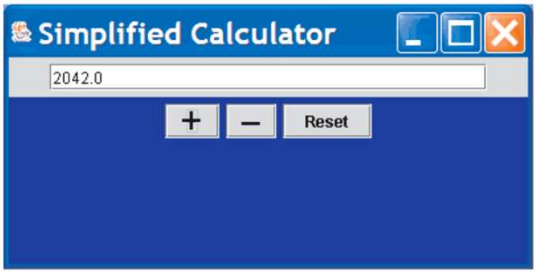


Display 17.19 **A Simple Calculator**

RESULTING GUI (After entering 42)



RESULTING GUI (After clicking +)



Window Listeners:

• Clicking the close-window button on a JFrame fires a window event • Window events are objects of the class WindowEvent

• The setWindowListener method can register a window listener for a window event
• A window listener can be programmed to respond to this type of event
• A window listener is any class that satisfies the WindowListener interface

• A class that implements the WindowListener interface must have definitions for all seven method headers in this interface
• Should a method not be needed, it is defined with an empty body
public void windowDeiconified(WindowEvent e) {}

Methods in the `WindowListener` Interface

The `WindowListener` interface and the `WindowEvent` class are in the package `java.awt.event`.

```
public void windowOpened(WindowEvent e)
```

Invoked when a window has been opened.

```
public void windowClosing(WindowEvent e)
```

Invoked when a window is in the process of being closed. Clicking the close-window button causes an invocation of this method.

```
public void windowClosed(WindowEvent e)
```

Invoked when a window has been closed.

Methods in the `WindowListener` Interface

```
public void windowIconified(WindowEvent e)
```

Invoked when a window is iconified. When you click the minimize button in a `JFrame`, it is iconified.

```
public void windowDeiconified(WindowEvent e)
```

Invoked when a window is deiconified. When you activate a minimized window, it is deiconified.

```
public void windowActivated(WindowEvent e)
```

Invoked when a window is activated. When you click in a window, it becomes the activated window. Other actions can also activate a window.

```
public void windowDeactivated(WindowEvent e)
```

Invoked when a window is deactivated. When a window is activated, all other windows are deactivated. Other actions can also deactivate a window.

A Window Listener

```
1   import javax.swing.JFrame;
2   import javax.swing.JPanel;
3   import java.awt.BorderLayout;
4   import java.awt.FlowLayout;
5   import java.awt.Color;
6   import javax.swing.JLabel;
7   import javax.swing.JButton;
8   import java.awt.event.ActionListener;
9   import java.awt.event.ActionEvent;
10  import java.awt.event.WindowListener;
11  import java.awt.event.WindowEvent;
```

A Window Listener

```
12  public class WindowListenerDemo extends JFrame
13  {
14      public static final int WIDTH = 300; //for main window
15      public static final int HEIGHT = 200; //for main window
16      public static final int SMALL_WIDTH = 200; //for confirm window
17      public static final int SMALL_HEIGHT = 100;//for confirm window

18      private class CheckOnExit implements WindowListener
19      {
20          public void windowOpened(WindowEvent e)
21          {}

22          public void windowClosing(WindowEvent e)
23          {
24              ConfirmWindow checkers = new ConfirmWindow();
25              checkers.setVisible(true);
26          }
```

This `WindowListener` *class is an inner class.*

(continued)

A Window Listener

```
27          public void windowClosed(WindowEvent e)
28          {}

29          public void windowIconified(WindowEvent e)
30          {}

31           public void windowDeiconified(WindowEvent e)
32          {}

33          public void windowActivated(WindowEvent e)
34          {}
```

A window listener must define all the method headings in the `WindowListener` *interface, even if some are trivial implementations.*

(continued)

A Window Listener

```
35          public void windowDeactivated(WindowEvent e)
36          {}
37      } //End of inner class CheckOnExit

38      private class ConfirmWindow extends JFrame implements ActionListener
39      {
40          public ConfirmWindow()
41          {
42              setSize(SMALL_WIDTH, SMALL_HEIGHT);
43              getContentPane().setBackground(Color.YELLOW);
44              setLayout(new BorderLayout());

45              JLabel confirmLabel = new JLabel(
46                              "Are you sure you want to exit?");
47              add(confirmLabel, BorderLayout.CENTER);
```

Another inner class.

(continued)

A Window Listener

```
48              JPanel buttonPanel = new JPanel();
49              buttonPanel.setBackground(Color.ORANGE);
50              buttonPanel.setLayout(new FlowLayout());

51              JButton exitButton = new JButton("Yes");
52              exitButton.addActionListener(this);
53              buttonPanel.add(exitButton);

54              JButton cancelButton = new JButton("No");
55              cancelButton.addActionListener(this);
56              buttonPanel.add(cancelButton);

57              add(buttonPanel, BorderLayout.SOUTH);
58          }
```

(continued)

A Window Listener

```
59          public void actionPerformed(ActionEvent e)
60          {
61              String actionCommand = e.getActionCommand();

62              if (actionCommand.equals("Yes"))
63                  System.exit(0);
64              else if (actionCommand.equals("No"))
65                  dispose();//Destroys only the ConfirmWindow.
66              else
67                  System.out.println("Unexpected Error in Confirm Window.");
68          }
69      } //End of inner class ConfirmWindow
```

(continued)

A Window Listener

```
70
71      public static void main(String[] args)
72      {
73          WindowListenerDemo demoWindow = new WindowListenerDemo();
74          demoWindow.setVisible(true);
75      }
76
77      public WindowListenerDemo()
78      {
79          setSize(WIDTH, HEIGHT);
80          setTitle("Window Listener Demonstration");
81
82          setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
83          addWindowListener(new CheckOnExit());
84
85          getContentPane().setBackground(Color.LIGHT_GRAY);
86          JLabel aLabel = new JLabel("I like to be sure you are sincere.");
87          add(aLabel);
88      }
89  }
```

Even if you have a window listener, you normally must still invoke `setDefaultCloseOperation`.

(continued)

A Window Listener Inner Class:

- An inner class often serves as a window listener for a JFrame
- The following example uses a window listener inner class name CheckOnExit
  addWindowListener(new CheckOnExit());
- When the close-window button of the main window is clicked, it fires a window event
- This is received by the anonymous window listener object
- This causes the windowClosing method to be invoked

- The method windowClosing creates and displays a ConfirmWindow class object
- It contains the message "Are you sure you want to exit?" as well as "Yes" and "No" buttons
- If the user clicks "Yes," the action event fired is received by the actionPerformed method
- It ends the program with a call to System.exit
- If the user clicks "No," the actionPerformed method invokes the dispose method
- This makes the calling object go away (i.e., the small window of the ConfirmWindow class), but does not affect the main window

The dispose Method:

• The dispose method of the JFrame class is used to eliminate the invoking JFrame without ending the program
• The resources consumed by this JFrame and its components are returned for reuse
• Unless all the elements are eliminated (i.e., in a one window program), this does not end the program
• dispose is often used in a program with multiple windows to eliminate one window without ending the program

The WindowAdapter Class:

• When a class does not give true implementations to most of the method headings in the WindowListener interface, it may be better to make it a derived class of the WindowAdapter class
• Only the method headings used need be defined
• The other method headings inherit trivial implementation from WindowAdapter, so there is no need for empty method bodies
• This can only be done when the JFrame does not need to be derived from any other class

Using `WindowAdapter`

This requires the following import statements:

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

```
1   private class CheckOnExit extends WindowAdapter
2   {
3       public void windowClosing(WindowEvent e)
4       {
5           ConfirmWindow checkers = new ConfirmWindow();
6           checkers.setVisible(true);
7       }
8   } //End of inner class CheckOnExit
```

Icons:

• JLabels, JButtons, and JMenuItems can have icons
• An icon is just a small picture (usually)
• It is not required to be small
• An icon is an object of the ImageIcon class
• It is based on a digital picture file such as .gif, .jpg, or .tiff
• Labels, buttons, and menu items may display a string, an icon, a string and an icon, or nothing
• The class ImageIcon is used to convert a picture file to a Swing icon
```
ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
```
• The picture file must be in the same directory as the class in which this code appears, unless a complete or relative path name is given
• Note that the name of the picture file is given as a string

• An icon can be added to a label using the setIcon method as follows:
```
JLabel dukeLabel = new JLabel("Mood check");
dukeLabel.setIcon(dukeIcon);
```
• Instead, an icon can be given as an argument to the JLabel constructor:
```
JLabel dukeLabel = new JLabel(dukeIcon);
```
• Text can be added to the label as well using the setText method:
```
dukeLabel.setText("Mood check");
```

Button or menu items can be created with just an icon by giving the ImageIcon object as an argument to the JButton or JMenuItem constructor
```
ImageIcon happyIcon = new
                ImageIcon("smiley.gif");
```
JButton smileButton = new JButton(happyIcon); JMenuItem happyChoice = new JMenuItem(happyIcon);
• A button or menu item created without text should use the setActionCommand method
to explicitly set the action command, since there is no string

Using Icons

```
1  import javax.swing.JFrame;
2  import javax.swing.JPanel;
3  import javax.swing.JTextField;
4  import javax.swing.ImageIcon;
5  import java.awt.BorderLayout;
6  import java.awt.FlowLayout;
7  import java.awt.Color;
8  import javax.swing.JLabel;
9  import javax.swing.JButton;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;


12 public class IconDemo extends JFrame implements ActionListener
13 {
14     public static final int WIDTH = 500;
15     public static final int HEIGHT = 200;
16     public static final int TEXT_FIELD_SIZE = 30;

17     private JTextField message;
```

(continued)

Using Icons

```
18     public static void main(String[] args)
19     {
20         IconDemo iconGui = new IconDemo();
21         iconGui.setVisible(true);
22     }


23     public IconDemo()
24     {
25         super("Icon Demonstration");
26         setSize(WIDTH, HEIGHT);
27         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

28         setBackground(Color.WHITE);
29         setLayout(new BorderLayout());
```

(continued)

Using Icons

```
30          JLabel dukeLabel = new JLabel("Mood check");
31          ImageIcon dukeIcon = new ImageIcon("duke_waving.gif");
32          dukeLabel.setIcon(dukeIcon);
33          add(dukeLabel, BorderLayout.NORTH);

34          JPanel buttonPanel = new JPanel();
35          buttonPanel.setLayout(new FlowLayout());
36          JButton happyButton = new JButton("Happy");
37          ImageIcon happyIcon = new ImageIcon("smiley.gif");
38          happyButton.setIcon(happyIcon);
39          happyButton.addActionListener(this);
40          buttonPanel.add(happyButton);
41          JButton sadButton = new JButton("Sad");
42          ImageIcon sadIcon = new ImageIcon("sad.gif");
43          sadButton.setIcon(sadIcon);
```

(continued)

Using Icons

```
44          sadButton.addActionListener(this);
45          buttonPanel.add(sadButton);
46          add(buttonPanel, BorderLayout.SOUTH);

47          message = new JTextField(TEXT_FIELD_SIZE);
48          add(message, BorderLayout.CENTER);
49      }


50      public void actionPerformed(ActionEvent e)
51      {
52          String actionCommand = e.getActionCommand();
```

(continued)

Using Icons

```
44          sadButton.addActionListener(this);
45          buttonPanel.add(sadButton);
46          add(buttonPanel, BorderLayout.SOUTH);

47          message = new JTextField(TEXT_FIELD_SIZE);
48          add(message, BorderLayout.CENTER);
49      }


50      public void actionPerformed(ActionEvent e)
51      {
52         String actionCommand = e.getActionCommand();
```

(continued)

```
53          if (actionCommand.equals("Happy"))
54              message.setText(
55                      "Smile and the world smiles with you!");
56          else if (actionCommand.equals("Sad"))
57              message.setText(
58                     "Cheer up. It can't be that bad.");
59          else
60               message.setText("Unexpected Error.");
61      }
62  }
```

```
public JButton()
public JMenuItem()
public JLabel()
```

Creates a button, menu item, or label with no text or icon on it. (Typically, you will later use `setText` and/or `setIcon` with the button, menu item, or label.)

```
public JButton(String text)
public JMenuItem(String text)
public JLabel(String text)
```

Creates a button, menu item, or label with the `text` on it.

```
public JButton(ImageIcon picture)
public JMenuItem(ImageIcon picture)
public JLabel(ImageIcon picture)
```

Creates a button, menu item, or label with the icon `picture` on it and no text.

Some Methods in the Classes `JButton`, `JMenuItem`, and `JLabel`

```
public JButton(String text, ImageIcon picture)
public JMenuItem(String text, ImageIcon picture)
public JLabel(
       String text, ImageIcon picture, int horizontalAlignment)
```

Creates a button, menu item, or label with both the `text` and the icon `picture` on it. `horizontalAlignment` is one of the constants `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.RIGHT`, `SwingConstants.LEADING`, or `SwingConstants.TRAILING`.
The interface `SwingConstants` is in the `javax.swing` package.

```
public void setText(String text)
```

Makes `text` the only text on the button, menu item, or label.

Some Methods in the Classes `JButton`, `JMenuItem`, and `JLabel`

```
public void setIcon(ImageIcon picture)
```

Makes `picture` the only icon on the button, menu item, or label.

```
public void setMargin(Insets margin)
```

`JButton` and `JMenuItem` have the method `setMargin`, but `JLabel` does not.
The method `setMargin` sets the size of the margin around the text and icon in the button or menu item. The following special case will work for most simple situations. The `int` values give the number of pixels from the edge of the button or menu item to the text and/or icon.

```
public void setMargin(new Insets(
                int top, int left, int bottom, int right))
```

The class `Insets` is in the `java.awt` package. (We will not be discussing any other uses for the class `Insets`.)

Some Methods in the Classes `JButton`, `JMenuItem`, and `JLabel`

```
public void setVerticalTextPosition(int textPosition)
```

Sets the vertical position of the text relative to the icon. The `textPosition` should be one of the constants `SwingConstants.TOP`, `SwingConstants.CENTER` (the default position), or `SwingConstants.BOTTOM`.
The interface `SwingConstants` is in the `javax.swing` package.

```
public void setHorizontalTextPosition(int textPosition)
```

Sets the horizontal position of the text relative to the icon. The `textPosition` should be one of the constants `SwingConstants.RIGHT`, `SwingConstants.LEFT`, `SwingConstants.CENTER`, `SwingConstants.LEADING`, or `SwingConstants.TRAILING`.
The interface `SwingConstants` is in the `javax.swing` package.

The Inserts Class:

- Objects of the class Inserts are used to specify the size of the margin in a button or menu item
- The arguments given when an Inserts class object is created are in pixels

• The Inserts class is in the package java.awt
public Inserts(int top, int left, int bottom, int right)

Scroll Bars:

• When a text area is created, the number of lines that are visible and the number of characters per line are specified as follows:
  JTextArea memoDisplay = new JTextArea(15, 30);
• However, it would often be better not to have to set a firm limit on the number of lines or the number of characters per line
• This can be done by using scroll bars with the text area

• When using scroll bars, the text is viewed through a view port that shows only part of the text at a time
• A different part of the text may be viewed by using the scroll bars placed along the side and bottom of the view port
• Scroll bars can be added to text areas using the JScrollPane class
• The JScrollPane class is in the javax.swing package
• An object of the class JScrollPane is like a view port with scroll bars
• When a JScrollPane is created, the text area to be viewed is given as an argument
  JScrollPane scrolledText = new JScrollPane(memoDisplay);
• The JScrollPane can then be added to a container, such as a JPanel or JFrame
  textPanel.add(scrolledText);

The scroll bar policies can be set as follows:
scrolledText.setHorizontalScrollBarPolicy(
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
scrolledText.setVerticalScrollBarPolicy(
JscrollPane.VERTICAL_SCROLLBAR_ALWAYS);
• If invocations of these methods are omitted, then the scroll bars will be visible only when needed
• If all the text fits in the view port, then no scroll bars will be visible
• If enough text is added, the scroll bars will appear automatically

Some Methods in the Class `JScrollPane`

The `JScrollPane` class is in the `javax.swing` package.

```
public JScrollPane(Component objectToBeScrolled)
```

Creates a new `JScrollPane` for the `objectToBeScrolled`. Note that the `objectToBeScrolled` need not be a `JTextArea`, although that is the only type of argument considered in this book.

```
public void setHorizontalScrollBarPolicy(int policy)
```

Sets the policy for showing the horizontal scroll bar. The `policy` should be one of

```
JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.
(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

Some Methods in the Class `JScrollPane`

```
public void setVerticalScrollBarPolicy(int policy)
```

Sets the policy for showing the vertical scroll bar. The `policy` should be one of

```
JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED
```

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in the text. The meanings of the other policy constants are obvious from their names.
(As indicated, these constants are defined in the class `JScrollPane`. You should not need to even be aware of the fact that they have `int` values. Think of them as policies, not as `int` values.)

A Text Area with Scroll Bars

```
1  import javax.swing.JFrame;
2  import javax.swing.JTextArea;
3  import javax.swing.JPanel;
4  import javax.swing.JLabel;
5  import javax.swing.JButton;
6  import javax.swing.JScrollPane;
7  import java.awt.BorderLayout;
8  import java.awt.FlowLayout;
9  import java.awt.Color;
10 import java.awt.event.ActionListener;
11 import java.awt.event.ActionEvent;
```

(continued)

A Text Area with Scroll Bars

```
12  public class ScrollBarDemo extends JFrame
13                              implements ActionListener
14  {
15      public static final int WIDTH = 600;
16      public static final int HEIGHT = 400;
17      public static final int LINES = 15;
18      public static final int CHAR_PER_LINE = 30;


19      private JTextArea memoDisplay;
20      private String memo1;
21      private String memo2;
```

(continued)

A Text Area with Scroll Bars

```
22      public static void main(String[] args)
23      {
24          ScrollBarDemo gui = new ScrollBarDemo();
25          gui.setVisible(true);
26      }


27      public ScrollBarDemo()
28      {
29          super("Scroll Bars Demo");
30          setSize(WIDTH, HEIGHT);
31          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

(continued)

A Text Area with Scroll Bars

```
32          JPanel buttonPanel = new JPanel();
33          buttonPanel.setBackground(Color.LIGHT_GRAY);
34          buttonPanel.setLayout(new FlowLayout());
35          JButton memo1Button = new JButton("Save Memo 1");
36          memo1Button.addActionListener(this);
37          buttonPanel.add(memo1Button);

38          JButton memo2Button = new JButton("Save Memo 2");
39          memo2Button.addActionListener(this);
40          buttonPanel.add(memo2Button);

41          JButton clearButton = new JButton("Clear");
42          clearButton.addActionListener(this);
43          buttonPanel.add(clearButton);
```

(continued)

A Text Area with Scroll Bars

```
44          JButton get1Button = new JButton("Get Memo 1");
45          get1Button.addActionListener(this);
46          buttonPanel.add(get1Button);

47          JButton get2Button = new JButton("Get Memo 2");
48          get2Button.addActionListener(this);
49          buttonPanel.add(get2Button);

50          add(buttonPanel, BorderLayout.SOUTH);

51          JPanel textPanel = new JPanel();
52          textPanel.setBackground(Color.BLUE);
```

(continued)

A Text Area with Scroll Bars

```
53          memoDisplay = new JTextArea(LINES, CHAR_PER_LINE);
54          memoDisplay.setBackground(Color.WHITE);

55          JScrollPane scrolledText = new JScrollPane(memoDisplay);
56          scrolledText.setHorizontalScrollBarPolicy(
57                      JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
58          scrolledText.setVerticalScrollBarPolicy(
59                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

60          textPanel.add(scrolledText);

61          add(textPanel, BorderLayout.CENTER);
62      }
```

(continued)

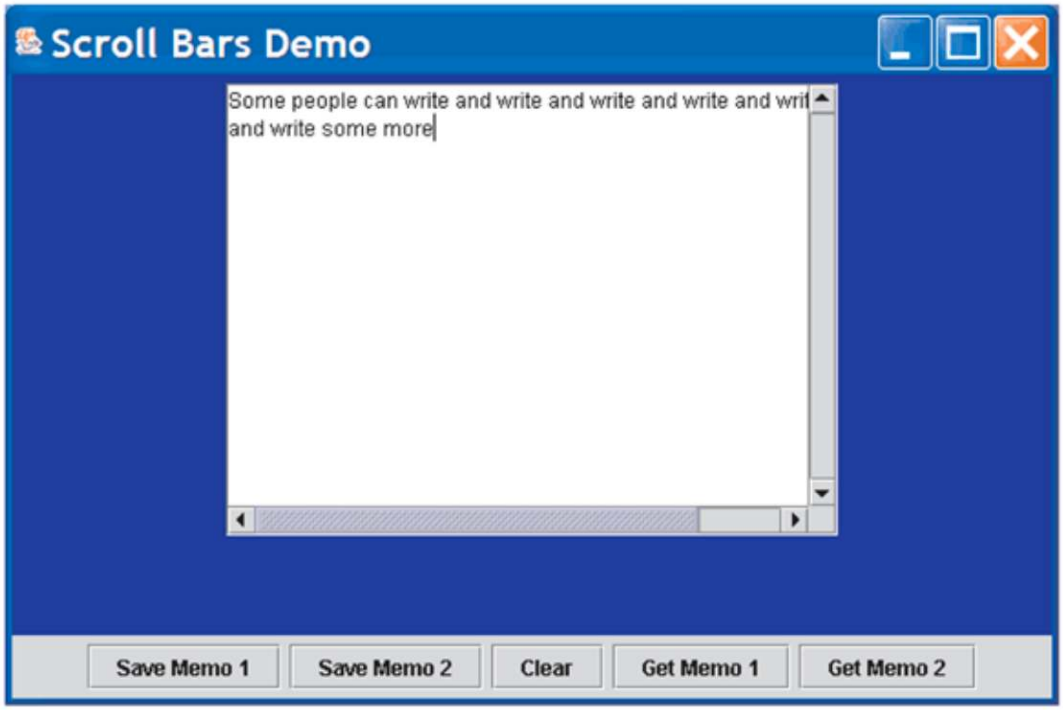
A Text Area with Scroll Bars

```
63      public void actionPerformed(ActionEvent e)
64      {
65          String actionCommand = e.getActionCommand();


66          if (actionCommand.equals("Save Memo 1"))
67              memo1 = memoDisplay.getText();
68          else if (actionCommand.equals("Save Memo 2"))
69              memo2 = memoDisplay.getText();
70          else if (actionCommand.equals("Clear"))
71              memoDisplay.setText("");
72          else if (actionCommand.equals("Get Memo 1"))
73              memoDisplay.setText(memo1);
74          else if (actionCommand.equals("Get Memo 2"))
75              memoDisplay.setText(memo2);
76          else
77              memoDisplay.setText("Error in memo interface");
78      }
79  }
```

(continued)

A Text Area with Scroll Bars

RESULTING GUI

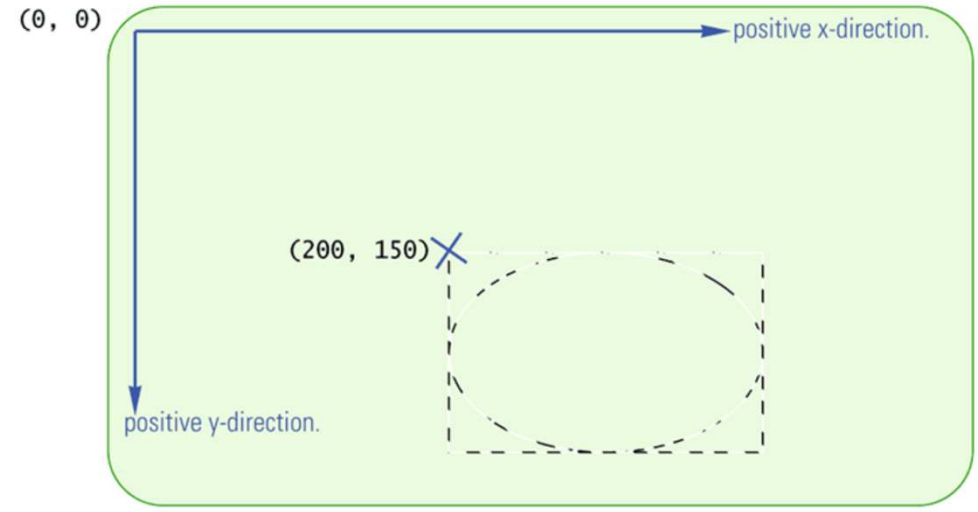


Coordinate System for Graphics Objects:

• When drawing objects on the screen, Java uses a coordinate system where the origin point (0,0) is at the upper-left corner of the screen area used for drawing
• The x-coordinate (horizontal) is positive and increasing to the right
• The y- coordinate(vertical) is positive and increasing down
• All coordinates are normally positive
• Units and sizes are in pixels
• The area used for drawing is typically a JFrame or JPanel

• The point (x,y) is located x pixels in from the left edge of the screen, and down y pixels from the top of the screen
• When placing a rectangle on the screen, the location of its upper-left corner is specified
• When placing a figure other than a rectangle on the screen, Java encloses the figure in an imaginary rectangle, called a bounding box, and positions the upper-left corner of this rectangle

Screen Coordinate System



Social Issues and Professional Practice:

The philosophical study of this is called axiology - the values that underlie your professional life.

Policy vacuum - The speed of innovations outpaces the slowness of devising policies and laws how to deal with the new technologies, leaving a 'vacuum' where events and practices occur. These events and practices, while (unintentionally?) not illegal, may still be unethical or immoral.

Critical reasoning is a branch of informal logic with which one can assess and analyze the arguments that occur in 'every day' natural language discourse.

Social Context:

Computers and the Internet, perhaps more than any other technologies, have transformed society over the past 75 years,
with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication;
and enabled fundamental breakthroughs in almost every branch of science and engineering.

Social Context provides the foundation for all other Social Issues and Professional Practice knowledge units, especially Professional Ethics.

Effects of the ICT Revolution:

Disruptive effects on:
Buying and selling
Communication
Access to knowledge
Socializing and falling in love
Crime
Classroom

Effects:

Globalization & Economics
Privacy and surveillance
How we think?

Social implications of computing in a networked world:

Not everything is positive about ICT
Disruptive effects on jobs
Changing media landscape
ICTs to facilitate violence (e.g.: drones, surveillance)

Human disposability and the fear of Artificial Intelligence:

Automatisation typically has the consequence of firing employees.

Asimov's laws of robotics:

A robot may not injure a human being or, through inaction, allow a human being to come to harm.

A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

A robot must protect its own existence as long as such protection does not conflict with the First or Second Laws.

ICT for Development:

The term 'development' is not unproblematic

It has a neutral use
E.g., as in “software development”
The strict sense of the word
It has a political connotation
To develop the developing world. As if ‘the developed world’ is static, finish en klaar
Assumes IT intervention will fix socio-economic problems or situations that are from outside casted as a problem
‘IT missionaries’?
ICT4D can be interpreted in both ways, is used in both ways, but perhaps more in the non-neutral interpretation

Potential of ICT4D:

Characteristic of a so-called Developing Country is the need for better and more equitable access to resources
An Information Community (or Society) is the desired outcome of the information revolution sparked by ICT
Knowledge resources can potentially be distributed to the have-nots without taking away from the haves
ICT can be used in a developing country to extend the distribution of knowledge resources
This includes information to support democracy and political accountability

What is the Digital Divide?

denotes the gap/difference between people who do have [easy, cheap, fast] access to the internet with all the information on the Web, and those who do not.

The major disparities in the penetration of the Information Society:
Disparities in the access and use of ICT
The gap between those who have access to the Information Society and those who are deprived of such access

Global Digital Divide (international): The disparity between countries at the forefront of the Information Economy and developing countries

Local Digital Divide (domestic): Disparities between groups in a particular country

What Causes the Digital Divide?

ICT mirrors and exacerbates existing disparities:
poor education and illiteracy
disability
location (rural-urban)
gender
race
income level

South African digital divide grows out of our history of division:
historical backlogs for large groups of people
a particular South African version of colonial history
Digital divide also arises from global circumstances that apply to all countries in the periphery (context: centre-periphery theory of (under)development)

Consequences of the Digital Divide:

Reflected in computer systems with:
cultural bias in the applications and contents
poor digital infrastructure
inappropriate computer equipment

How to bridge the Digital Divide?

On-the-ground initiatives
providing sustainable solutions in
under-serviced communities
Policy reform
government policy will have to change to make ICT more accessible to all

Universal Service and Universal Access:

Universal service is the rather old principle that all residents of a country should have access to basic communications services

The Universal Service and Access Agency of South Africa (USAASA) is a state owned entity of government established through the Electronic Communications Act

to ensure that "every man, woman and child whether living in the remote areas of the Kalahari or in urban areas of Gauteng can be able to connect, speak, explore and study using ICT's."
The Universal Service and Access Fund was established to fund projects and programs that strive to achieve universal service and access to ICTs by all South African citizens.

Peacebuilding: "A range of measures targeted to reduce the risk of lapsing or relapsing into conflict by strengthening national capacities at all levels for conflict management, and to lay the foundation for sustainable peace and development."

Introduction to Values:

Computer Ethics describes the field that examines ethical/moral issues pertaining to computing and information technology
Related terms:

Information ethics refers to a cluster of ethical concerns regarding the flow of information that is either enhanced or restricted by computer technology.

Internet ethics concerns ethical issues involving the Internet in particular

Cyber ethics broadens the scope beyond individual machines or the concerns of computer professionals

We use computer ethics because computer science is a well established term that is not restricted to individual machines nor are its concerns only those of computer professionals.

Ethics is the branch of philosophy which studies morality, "objects of the study of ethics are therefore the moral rules, the ways in which they are created and justified, and the ways in which they are applied or should be applied… spectrum of what is right and wrong …[and deals] with the issue of the good life and the search for happiness"

Morality refers to manners and habits, considered as a "collection of codes of conduct that are created by the conscience, society, or religion" or is universal and "inherent to all human beings despite the particular situation" with as source human reason.

Moral Theory:

Divine command theory: X is good just because God commands it

Cultural relativism: an action is judged good (or bad) based only on the standards that have been adopted by one's society

Teleological (utilitarianism): the right decision what is good or desirable (not the motivation behind it) is the one that causes the most happiness

Deontological ethics (Kant): reasons more important than the ends, by focusing on rights, duties, obligations, and rules

Moral Agency:

Causality: An agent can be held responsible if the ethically relevant result is an outcome of its actions.

Knowledge: An agent can be blamed for the result of its actions if it had (or should have had) knowledge of the consequences of its actions.

Choice: An agent can be blamed for the result if it had the liberty to choose an alternative without greater harm for itself.

Cultural Relativism:

Ethics is relative
What is right for me might not be right for you
There is (always has been) a good deal of diversity regarding right and wrong
Moral beliefs change over time and in a given Society
Social Environment plays an important role in shaping moral ideals
All the above are true individually but don't prove (or disprove) that there is a Universal right or wrong

Golden Rule (ethic of reciprocity):

Never impose on others what you would not choose for yourself.

Action-Based Ethical Theory: Deontology and Teleology

Action-based theories focus entirely upon the actions which a person performs

When actions are judged morally right based upon how well they conform to some set of rules, we have a deontological moral theory.
Greek deontos, expresses “obligation”
actions are essentially right or wrong, without regard to their consequences
Some actions are never justified (ends cannot justify means)

When actions are judged morally right based upon their consequences, we have teleological or consequentialist ethical theory.
Greek telos, meaning “end” or “goal”
Action may be justified by special circumstances (ends can justify means)

Deontology: Duty-based Ethics (Pluralism)

Seven basic moral duties that are binding on moral agents:
Fidelity: one ought to keep promises
Reparation: one ought to right the wrongs that one has inflicted on others
Justice: one ought to distribute goods justly
Beneficence: one ought to improve the lot of others with respect to virtue, intelligence, and happiness
Self-improvement: one ought to improve oneself with respect to virtue and intelligence
Gratitude: one ought to exhibit gratitude when appropriate
Non-injury: one ought to avoid injury to others

Teleology:

Teleological theories give priority to “the good” over “the right”
They evaluate actions by the goal or consequences that they achieve.
Correct actions are those that produce the most good or optimize the consequences of choices,
E.g. Voltaire’s “best of all possible worlds”
Wrong actions are those that do not contribute to the good.
Three examples of the Teleological approach to ethics are Egoism, Utilitarianism and Altruism.

Ethical Egoism:

Egoism focuses on self-interest
Ethical egoism claims that it is necessary and sufficient for an action to be morally right that it maximize one's self-interest.
“Does the action benefit me, as an individual, in any way?”

Utilitarianism:

Utilitarianism embodies the notion of operating in the public interest rather than for personal benefit.
An action is right if it maximises benefits over costs for all involved, everyone counting equal.

Utilitarianism vs. Deontology:

In utilitarianism what makes an action right/wrong
is outside the action
It’s the consequences that make it right/wrong

For deontologists
It’s the principle inherent in the action
If the action done from a sense of duty &
If the principle can be universalised
Then the action is right

Altruism:

“A decision results in benefit for others, even at a cost to some”
an action is ethically right if it brings good consequences to others (even at the cost to yourself)
Altruists choose to align their well-being with others — so they are happy when others thrive, sad when others are suffering.

Nonmaleficence - above all do no harm

Autonomy:

self-determining
Kant: For an individual to be truly human, that person must be free to decide what is in his or her best interest.

Ubuntu:

Ubuntu means humanness/humanity in isiZulu
I am, because we are; and since we are, therefore I am

An action is right just insofar as it promotes shared identity among people grounded on good-will; an act is wrong to the extent that it fails to do so and tends to encourage the opposites of division and ill-will.

Critical Theory:

Marxist tradition known as the Frankfurt School
Has a specific practical purpose
A theory is critical to the extent that it seeks human "emancipation from slavery", acts as a "liberating … influence", and works "to create a world which satisfies the needs and powers" of human beings
Broadly: a critical theory provides the descriptive and normative bases for social inquiry aimed at decreasing domination and increasing freedom in all their forms

Data mining is a process of exploration and analysis of large quantities of data, by automatic or semi-automatic means.
To discover meaningful patterns and rules.

Machine Learning: related, but focuses on prediction based on training data
Data Analytics: also related, but goes with a hypothesis and used for decision making

What is it about computers that make the computer environment different?

- **Speed**: Computers are able to do things faster than ever before
    - data mining (only viable with computers)
- **Storage and accessibility of data**: Vast amount of data can be stored and easily accessible for processing.
    - Big Data
- **Concept of a program**: How should one treat a computer program?
    - Is it a *property* or an *idea* or a *process*?
    - Is it something that may be *copyrighted* or *patented*? [see video]
- **Breadth of Distribution:** IT has presented consumers with a new channel of distribution
    - Faster, easier

- not as regulated internationally

A three-step guide for approaching computer ethics issues:

Step 1. *Identify* a practice involving ICT, or a feature of that technology, that is controversial from a moral perspective.

1a. Disclose any hidden (or opaque) features or issues that have moral implications

1b. If the ethical issue is descriptive, assess the sociological implications for relevant social institutions and socio-demographic and populations.

1c. If the ethical issue is also normative, determine whether there are any specific guidelines, that is, professional codes that can help you resolve the issue.

1d. If the normative ethical issue cannot be resolved in this way then go to Step 2.

Step 2. *Analyse* the ethical issue by clarifying concepts and situating it in a context.

2a. If a policy vacuums exists, go to Step 2b; otherwise, go to Step 3.

2b. Clear up any conceptual muddles involving the policy vacuum and go to Step 3.

Step 3. *Deliberate* on the ethical issue. The deliberation process requires two stages.

3a. Apply one or more moral theories to the analysis of the moral issue, and then go to Step 3b.

3b. Justify the position you reached by evaluating it via the standards and criteria for successful logic argumentation.

Computer Ethics describes the field that examines ethical/ moral issues pertaining to computing and information technology.

Information ethics refers to a cluster of ethical concerns regarding the flow of information that is either enhanced or restricted by computer technology.

Internet ethics concerns ethical issues involving the Internet in particular

Cyber ethics broadens the scope beyond individual machines or the concerns of computer professionals

Problem of ‘many hands’:

• Situation where multiple actors are involved in the development and deployment of technologies, which makes it hard to identify who exactly did what.

• Negatively affects the process of assigning blame when a technological accident occurs, as well as who to praise for its success

Trustworthy Software:

• The enhancement of the overall software and systems culture, with the objective that software should be designed, implemented and maintained in a trustworthy manner.

• Algorithmic transparency (esp. in context of algorithmic decision-making)
New joint statement by the ACM, "intended to support the benefits of algorithmic decision-making while addressing these concerns. These principles should be addressed during every phase of system development and deployment to the extent necessary to minimize potential harms while realizing the benefits of algorithmic decision-making "

Critical Reasoning and Logical Arguments:

• Critical reasoning is a branch of informal logic.
• Critical reasoning tools, especially argument analysis, can help us to resolve many of the disputes in computer ethics.
• A logical argument, or argument, is a form of reasoning comprising various claims, or statements (or sentences).

Logical Arguments:

- A set of statements such that:
- One of them is being said to be true (conclusion)
- The other(s) are being offered as reasons for believing the truth of the one (called premises).

Arguments and Assertions:

- An argument is a set of statements, one of which is being asserted
- An assertion is a single statement (possibly complex) that is being stated as a fact
- Assertions are either true or false
- but arguments are neither true nor false
- arguments are either valid or invalid
- A valid argument is one where the conclusion follows from the premises
- In addition, if the premises are all true then it is sound

Deductive Arguments:

- Deductive arguments are such that if their premises are true then the truth of their conclusion is guaranteed
- A deductive argument is either
- valid (and gives us certainty)
- or invalid and says nothing.
- For example:
  All penguins are black and white
  Penny is a penguin
  Therefore Penny is black and white

Inductive Arguments:

- Inductive arguments are such that the truth of their premises makes the conclusion more probably true.
- Inductive arguments can be either weak or strong (worse or better).
- their conclusions can be slightly more likely to be true
- or much more likely to be true
- Example of a strong inductive argument:
  The sun has risen every day for millions of years
  Therefore the sun will rise tomorrow

The form of a valid deductive argument:

• A valid argument is valid solely in virtue of its logical form, regardless its content
• An example of a valid logical form is:
Premise 1. Every A is a B.
Premise 2. C is an A.
Conclusion: C is a B.
• No matter which values are substituted for A, B, and C, the argument in this form is always valid.
• modus ponens (implication elimination)
Premise 1. A implies B
Premise 2. A is true
Conclusion: B is true

Sound and unsound arguments:

• For a deductive argument to be sound, it must be:
a) valid (i.e., the assumed truth of the premises would guarantee the truth of the argument's conclusion);
b) the (valid) argument's premises must also be true in the actual world.
• A deductive argument is sound if and only if it is valid and all its premises are true.

Counter examples to arguments:

A possible (actual) case where the premises in an argument can be imagined to be (is) true while, at the same time, the conclusion could still be false
• Note that if a deductive argument is valid, no counterexample is possible
• To put it differently: if you can give one counterexample to the argument then it is invalid.
• In informal reasoning, appeal to "the exception to the rule" is sometimes made

Types of Inductive Arguments:

• Inductive generalizations:
• The premise identifies a characteristic of a sample of a population
• The conclusion extrapolates that characteristic to the rest of the population.

• 'Causal' generalizations:

- The premise identifies a correlation between two types of event
- The conclusion states that events of the first type cause events of the second type.

- Arguments from analogy:
- Arguments from analogy take just one example of something
- Then extrapolate from a character of that example...
- .... to the character of something similar to that thing

- Arguments from authority:
- Take one person or group of persons
- Who are, or are assumed to be, right about some things...
- And extrapolate to the claim they are right about other things

Fallacies - faulty reasoning

Fallacies of relevance:

- Citing in support of a conclusion something that is true but irrelevant (non-sequitur):

Bill lives in a large building, therefore his apartment is large.
Every year many people are supported through life by their religious beliefs, so their religious beliefs must be true

- Attacking the person making the argument rather than the argument that is made (ad hominem)

How can we take seriously a position regarding the future of our national defense that has been opposed by MP X, who has been arrested for drunken driving and who has been involved in extramarital affairs?

Fallacies of vacuity:

- Citing in support of a conclusion that very conclusion (circular arguments)
- In a circular argument the conclusion is one of the premises
- Citing in support of a conclusion a premise that assumes the conclusion (begging the question)
- In a question-begging argument the conclusion is assumed by one of the premises

Fallacies of clarity:

- vagueness (fallacy of the heap)

If you have only ten cents you are not rich
If you are not rich and I give you ten cents then you still won't be rich It doesn't matter how many cents I give you, you won't be rich

- misusing borderline cases (slippery slopes)

X could possibly be abused; therefore, we should not allow X

- trading on ambiguity (equivocation)

A feather is light
What is light cannot be dark Therefore, a feather cannot be dark

Straw man I:

- The straw man fallacy is when you misrepresent someone else's position so that it can be attacked more easily, then knock down that misrepresented position, then conclude that the original position has been demolished. It's a fallacy because it fails to deal with the actual arguments that have been made.

Philosophical Analysis:

Philosophical Analysis is an ongoing process
- Claim
- Argument
- Critical Examination of Argument (Redo if wrong)

This process is known as the dialectic approach
- Does not always lead to a conclusion

Rational Analysis:

- establish one or more issues to be analysed.
- for each issue
- the law and principles presented in agreed guidelines are applied.
- one or more alternative options are presented.
- allow you to examine these rationally and
- choose the correct one.
- The analysis will disqualify some options to the ethical issue in favour of others.

Descriptive & Normative claims:

Descriptive statements

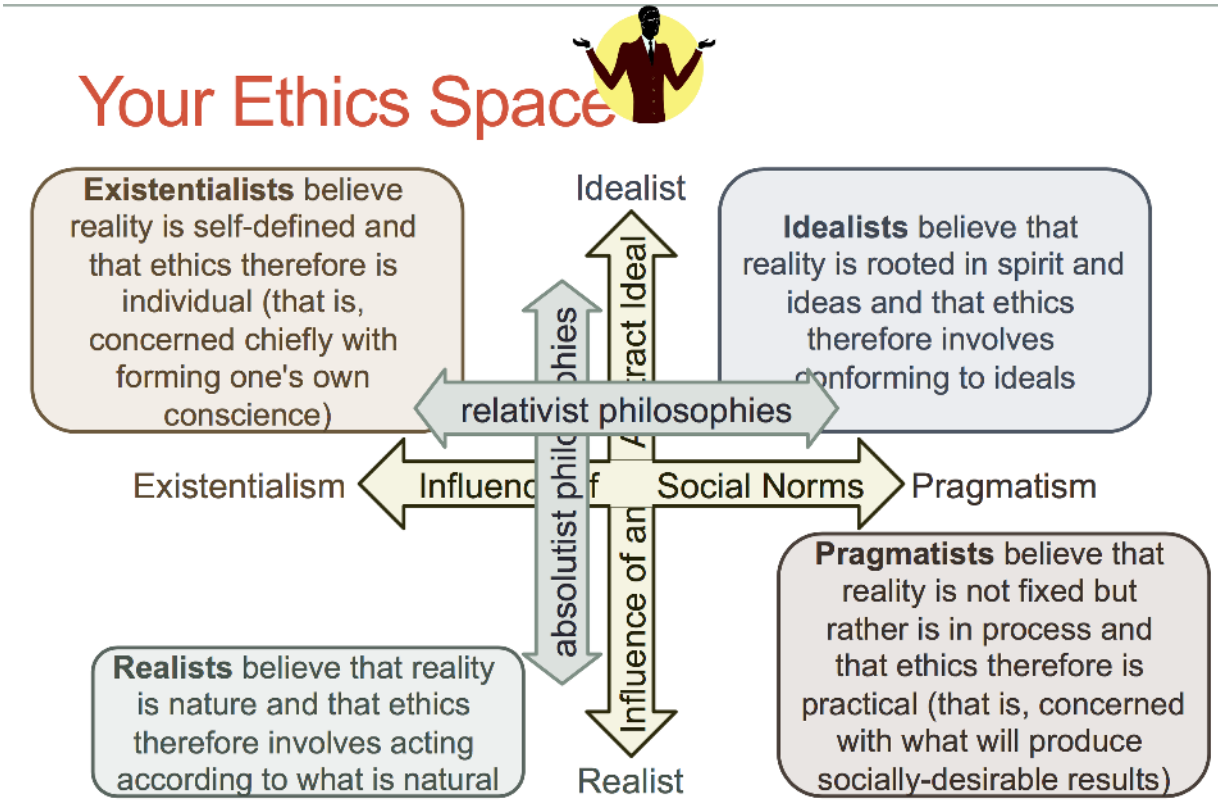
- Describe something as a fact (the sky is blue)
- They can be tested objectively to verify them

Philosophical ethics is Normative

- Explores what people ought to do
- Evaluates arguments, reasons, theories

Ethical theories are prescriptive

- Try to provide an account of why certain behaviours are good/bad or right/wrong



Law:

Law is there to support to orderly functioning of society (well, in theory, that's the intention)
Law tells us to do or not to do something, and consequences

⇒ A recognised, established authority has decided on actions the law permits or prohibits

- Believe it benefits society in some way
- The law is often, but not always,

grounded in ethical principles.

"when we are confronted with an ethical decision, we should first research the law".

A Framework for Ethical Analysis:

a) List all the relevant facts
    - As much as possible, a neutral, logical exercise
    - Interpretation is involved in selecting pertinent facts
    - Facts are not judged
b) List all stakeholders affected by the action
    - Judge whether a stakeholder is important enough to be listed
c) Consider the courses of action for the stakeholders
    - Ask if they have an obligation or duty to do or not do something
    - Evaluate all the reasons that individuals give to justify their actions then consider the reasons in turn to determine if they matter
        - E.g.: Does it matter if they fail to fulfil their duty
        -

Having established courses of action, apply four steps:

1. Formal Guidelines
2. Ethical Theory
3. Legal Issues
4. Weigh up the argument rationally

- Professional codes of conduct help guide ethical decisions
    - rules that state the principal duties of all professionals.
- Corporate or professional codes:
    - Association for Computing Machinery (ACM)
    - Institute of Electrical and Electronics Engineers (IEEE)
    - British Computer Society (BCS)
    - Institute of Information Technology Professionals South Africa (IITPSA)
    - Ubuntu Leadership Code of Conduct