MODULE 1: Overview of Object oriented Programming and Java Basics

Need for OOP Paradigm

- All computer programs consist of two elements: code and data.
- A program can be conceptually organized around its code or around its data. Some programs are written around "what is happening" and others are written around "who is being affected."
- These are the two paradigms that govern how a program is constructed. The first way is called the process oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success.
- To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code.

Abstraction

- An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces.

The Three OOP Principles

- All object-oriented programming languages provide mechanisms to implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

Encapsulation :

- Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- In Java, the basis of encapsulation is the class.
- When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class.
- Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods.
- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class.

Inheritance

- Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification

Polymorphism

- Polymorphism (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
- More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the

same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (that is, method) as it applies to each situation.

A First Simple Program

```
/*
 This is a simple Java program.
 Call this file "Example.java".
*/
class Example {
 // Your program begins with a call to main().
 public static void main(String[] args) {
 System.out.println("This is a simple Java program.");
 }
}
```

Entering the Program

- In Java, all code must reside inside a class. By convention, the name of the main class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

- To compile the Example program, execute the compiler, javac, specifying the name of the source file on the command line, as shown here:
- C:\>javac Example.java .
- The javac compiler creates a file called Example.class that contains the bytecode version of the program. The Java bytecode is the intermediate representation of your

program that contains instructions the Java Virtual Machine will execute. Thus, the output of javac is not code that can be directly executed.

- To actually run the program, you must use the Java application launcher called java. To do so, pass the class name Example as a command-line argument, as shown here:
- C:\>java Example
- When the program is run, the following output is displayed: This is a simple Java program. When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension.. When you execute java as just shown, you are actually specifying the name of the class that you want to execute. It will automatically search for a file by that name that has the .class extension. If it finds the file, it will execute the code contained in the specified class.
- The program begins with the following lines: /* This is a simple Java program. Call this file "Example.java". */ This is a comment.
- The next line of code in the program is shown here:
- class Example { This line uses the keyword class to declare that a new class is being defined. Example is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).
- The next line in the program is the single-line comment, shown here: // Your program begins with a call to main(). This is the second type of comment supported by Java. A single-line comment begins with a // and ends at the end of the line.
- The next line of code is shown here: public static void main(String[] args) { This line begins the main( ) method.
- The public keyword is an access modifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.
- In this case, main( ) must be declared as public, since it must be called by code outside of its class when the program is started. The keyword static allows main( ) to be called without having to instantiate a particular instance of the class. This is necessary since main( ) is called by the Java Virtual Machine before any objects are made. The keyword void simply tells the compiler that main( ) does not return a value

- Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses.
- In main( ), there is only one parameter, albeit a complicated one. String[ ] args declares a parameter named args, which is an array of instances of the class String. (Arrays are collections of similar objects.) Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed.
- The next line of code is shown here. Notice that it occurs inside main( ). System.out.println("This is a simple Java program."); This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. In this case, println( ) displays the string which is passed to it. As you will see, println( ) can be used to display other types of information, too. The line begins with System.out. While too complicated to explain in detail at this time, briefly, System is a predefined class that provides access to the system, and out is the output stream that is connected to the console.

Two Control Statements:

The if Statement

- The Java if statement works much like the IF statement in any other language. It determines the flow of execution based on whether some condition is true or false. Its simplest form is shown here:
- if(condition) statement;
- Here, condition is a Boolean expression. If condition is true, then the statement is executed. If condition is false, then the statement is bypassed.
- Here is an example:
- if(num < 100) System.out.println("num is less than 100");

- In this case, if num contains a value that is less than 100, the conditional expression is true, and println( ) will execute. If num contains a value greater than or equal to 100, then the println( ) method is bypassed.

Operator	Meaning
<	Less than
>	Greater than
==	Equal to

```
class IfSample {
 public static void main(String[] args) {
 int x, y;
 x = 10;
 y = 20;
 if(x < y) System.out.println("x is less than y");
 x = x * 2;
 if(x == y) System.out.println("x now equal to y");
 x = x * 2;
 if(x > y) System.out.println("x now greater than y");
 // this won't display anything
 if(x == y) System.out.println("you won't see this");
 }
}
```

The output generated by this program is shown here:

```
 x is less than y
 x now equal to y
 x now greater than y
```

The for Loop

- Loop statements are an important part of nearly any programming language because they provide a way to repeatedly execute some task.

- The simplest form of the for loop is shown here:

  for(initialization; condition; iteration) statement;
- In its most common form, the initialization portion of the loop sets a loop control variable to an initial value. The condition is a Boolean expression that tests the loop control variable. If the outcome of that test is true, statement executes and the for loop continues to iterate. If it is false, the loop terminates. The iteration expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the for loop:

```
/*
 Demonstrate the for loop.
 Call this file "ForTest.java".
*/
class ForTest {
 public static void main(String[] args) {
 int x;
 for(x = 0; x<10; x = x+1)
 System.out.println("This is x: " + x);
 }
}
```

This program generates the following output:

```
 This is x: 0
 This is x: 1
 This is x: 2
 This is x: 3
This is x: 4
 This is x: 5
 This is x: 6
 This is x: 7
 This is x: 8
 This is x: 9
```

Data Types, Variables, and Arrays

The Primitive Types

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

- Integers This group includes byte, short, int, and long, which are for whole valued signed numbers.
- Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
- Characters This group includes char, which represents symbols in a character set, like letters and numbers.
- Boolean This group includes boolean, which is a special type for representing true/false values.

Integers

- Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values.
- The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type.
- The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

byte

- The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you're working with a stream of data from a network or file.
- For example, the following declares two byte variables called b and c: byte b, c;

Short

- short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least used Java type. Here are some examples of short variable declarations:
  short s;
  short t;

int

- The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647

long

- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large

```
class Light {
 public static void main(String[] args) {
 int lightspeed;
 long days;
 long seconds;
 long distance;
 // approximate speed of light in miles per second
 lightspeed = 186000;
 days = 1000; // specify number of days here
 seconds = days * 24 * 60 * 60; // convert to seconds
 distance = lightspeed * seconds; // compute distance
 System.out.print("In " + days);
 System.out.print(" days light will travel about ");
 System.out.println(distance + " miles.");
 }
}
```

This program generates the following output:

```
 In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an int variable

Floating-Point Types

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
Double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Each of these floating-point types is examined next.

float

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

Here are some example float variable declarations:

```
float hightemp, lowtemp;
```

double

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
 public static void main(String[] args) {
 double pi, r, a;
r = 10.8; // radius of circle
 pi = 3.1416; // pi, approximately
 a = pi * r * r; // compute area
 System.out.println("Area of circle is " + a);
 }
}
```

Characters

In Java, the data type used to store characters is char. A key point to understand is that Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.

Here is a program that demonstrates char variables:

```
// Demonstrate char data type.
class CharDemo {
 public static void main(String[] args) {
 char ch1, ch2;
 ch1 = 88; // code for X
 ch2 = 'Y';
 System.out.print("ch1 and ch2: ");
 System.out.println(ch1 + " " + ch2);
 }
}
```

This program displays the following output:

```
 ch1 and ch2: X Y
```

Booleans

Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for.

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest {
 public static void main(String[] args) {
 boolean b;
 b = false;
 System.out.println("b is " + b);
 b = true;
 System.out.println("b is " + b);
 // a boolean value can control the if statement
 if(b) System.out.println("This is executed.");
 b = false;
 if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
 System.out.println("10 > 9 is " + (10 > 9));
 }
}
```

The output generated by this program is shown here:

```
 b is false
 b is true
 This is executed.
 10 > 9 is true
```

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [= value][, identifier [= value] …];

Here, type is one of Java's atomic types, or the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
 // d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
```

```
class DynInit {
 public static void main(String[] args) {
 double a = 3.0, b = 4.0;
 // c is dynamically initialized
 double c = Math.sqrt(a * a + b * b);
 System.out.println("Hypotenuse is " + c);
 }
}
```

The Scope and Lifetime of Variables

A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

```
// Demonstrate block scope.
class Scope {
 public static void main(String[] args) {
 int x; // known to all code within main
 x = 10;
 if(x == 10) { // start new scope
 int y = 20; // known only to this block
 // x and y both known here.
 System.out.println("x and y: " + x + " " + y);
 x = y * 2;
 }
 // y = 100; // Error! y not known here
 // x is still known here.
 System.out.println("x is " + x);
 }
}
```

Type Conversion and Casting

If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed Java's Automatic Conversions. When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an int value to a byte variable? This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion.

It has this general form:

```
(target-type) value
```

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;
byte b;
// …
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
 public static void main(String[] args) {
 byte b;
 int i = 257;
 double d = 323.142;
 System.out.println("\nConversion of int to byte.");
 b = (byte) i;
 System.out.println("i and b " + i + " " + b);
 System.out.println("\nConversion of double to int.");
 i = (int) d;
 System.out.println("d and i " + d + " " + i);
 System.out.println("\nConversion of double to byte.");
 b = (byte) d;
 System.out.println("d and b " + d + " " + b);
 }
}
```

This program generates the following output:

```
 Conversion of int to byte.
 i and b 257 1
 Conversion of double to int.
 d and i 323.142 323
 Conversion of double to byte.
 d and b 323.142 67
```

The Type Promotion Rules

Java defines several type promotion rules that apply to expressions. They are as follows: First, all byte, short, and char values are promoted to int, as just described. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands are double, the result is double.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote {
 public static void main(String[] args) {
 byte b = 42;
 char c = 'a';
 short s = 1024;
 int i = 50000;
 float f = 5.67f;
 double d = .1234;
 double result = (f * b) + (i / c) - (d * s);
 System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
 System.out.println("result = " + result);
 }
}
```

Let’s look closely at the type promotions that occur in this line from the program:

```
double result = (f * b) + (i / c) - (d * s);
```

In the first subexpression, f * b, b is promoted to a float and the result of the subexpression is float. Next, in the subexpression i/c, c is promoted to int, and the result is of type int. Then, in d * s, the value of s is promoted to double, and the type of the subexpression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Arrays

- An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index.
- One-Dimensional Arrays A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.
- The general form of a one-dimensional array declaration is type[ ] var-name;
- Here, type declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.
- For example, the following declares an array named month_days with the type "array of int": **int[] month_days;** Although this declaration establishes the fact that month_days is an array variable, no array actually exists. To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days.
- The general form of new as it applies to one-dimensional arrays appears as follows: array-var = new type [size]; Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array.
- This example allocates a 12-element array of integers and links them to month_days: month_days = new int[12]; After this statement executes, month_days will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.
- // Demonstrate a one-dimensional array.
- class Array {
- public static void main(String[] args) {
- int[] month_days;

- month_days = new int[12];
- month_days[0] = 31;
- month_days[1] = 28;
- month_days[2] = 31;
- month_days[3] = 30;
- month_days[4] = 31;
- month_days[5] = 30;
- month_days[6] = 31;
- month_days[7] = 31;
- month_days[8] = 30;
- month_days[9] = 31;
- month_days[10] = 30;
- month_days[11] = 31;
- System.out.println("April has " + month_days[3] + " days.");
- }
- }

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use new. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
 public static void main(String[] args) {
 int[] month_days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
 30, 31 };
 System.out.println("April has " + month_days[3] + " days.");
 }
}
```

When you run this program, you see the same output as that generated by the previous Version

Multidimensional Arrays

- In Java, multidimensional arrays are implemented as arrays of arrays. To declare a

multidimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a two-dimensional array variable called twoD:

```
int[][] twoD = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to twoD.

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
 public static void main(String[] args) {
 int[][] twoD= new int[4][5];
 int i, j, k = 0;
 for(i=0; i<4; i++)
 for(j=0; j<5; j++) {
 twoD[i][j] = k;
 k++;
 }
 for(i=0; i<4; i++) {
 for(j=0; j<5; j++)
 System.out.print(twoD[i][j] + " ");
 System.out.println();
 }
 }
}
```

This program generates the following output:

```
 0 1 2 3 4
 5 6 7 8 9
 10 11 12 13 14
 15 16 17 18 19
```