POP & OOP PARADIGMS

Two Paradigms of Programming:

- As you know, all computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, 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. Problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived.

OOP

- 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. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Procedure oriented Programming:

- In this approach, the problem is always considered as a sequence of tasks to be done. A number of functions are written to accomplish these tasks.
- Here primary focus on —Functions‖ and little attention on data.
- There are many high level languages like COBOL, FORTRAN, PASCAL, C used for conventional programming commonly known as POP.
- POP basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.

- Normally a flowchart is used to organize these actions and represent the flow of control logically sequential flow from one to another.
- In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data.
- Global data are more vulnerable to an in advent change by a function.
- In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all the functions that access the data. This provides an opportunity for bugs to creep in.

Drawback:

- It does not model real world problems very well, because functions are action oriented and do not really corresponding to the elements of the problem.

OOP CONCEPTS

- Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects.. We sometimes divide objects into two categories: animate and inanimate. Animate objects are —alive‖ in some sense—they move around and do things. Inanimate objects, on the other hand, do not move on their own .Objects of both types, however, have some things in common. They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleep crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We will study the kinds of attributes and behaviors that software objects have.

- Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees.
- Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes and behaviors just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

OOP:

- OOP allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

- DEF: OOP is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of such modules on demand.
- I.E. , an object a considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

ORGANIZATION OF OOP

OOP WITH JAVA

- Languages like Java are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement an object-oriented design as a working system.
- Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. In Java, the unit of programming is the class from which objects are eventually instantiated (created). Java classes contain methods (which implement operations and are similar to functions in C) as well as fields (which implement attributes).

- Computer programs, such as the Java programs, AND other OO programs are composed of lots of interacting software objects

- Java programmers concentrate on creating classes. Each class contains fields, and the set of methods that manipulate the fields and provide services to clients (i.e., other classes that use the class).
- The programmer uses existing classes as the building blocks for constructing new classes. Classes are to objects as blueprints are to houses. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.

Classes

- Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the —bank teller‖ class needs to relate to the —customer‖ class, the —cash drawer‖ class, the —safe‖ class, and so on. These relationships are called associations.

- Packaging software as classes makes it possible for future software systems to reuse the classes.

- Object-oriented programming is at the core of Java. In fact, all Java programs are object- oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java.

Brief history of java

CLASSES AND OBJECTS

- A class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.
- A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited.
- Classes can (and usually do) get much more complex. The general form of a **class** definition is shown here:

```
class classname {
   type instance-variable1;
  type instance-variable2;
// ...
   type instance-variableN;
   type methodname1(parameter-list) { /
/ body of method
 }
type methodname2(parameter-list) {
// body of method
 }
// ... type methodnameN(parameter-list) {
 // body of method
 }
}
```

- The data, or variables, defined within a **class** are called ***instance variables.***
- The code is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class.
- In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

DATA TYPES

- Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. 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. Java does not support unsigned, positive-only integers. Many other Computer languages, including C/C++, support both signed and unsigned integers.
- **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

- Syntax: byte b, c;

  short s; short t;

- 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); } }

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. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

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*] ...] ;

The *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. 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'.
```

Scope &life time of a Variable

- So far, all of the variables used have been declared at the start of the **main( )** method. However, Java allows variables to be declared within any block (A block is begun with an opening curly brace and ended by a closing curlybrace). A block defines a *scope.* **Thus, each time you start a new block, you are creating a new scope**.

- As you probably know from your previous programming experience, **a scope determines what objects are visible to other parts of your program**. It also **determines the lifetime of those objects**.

OPERATORS

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
- The following table lists the arithmetic operators: Operator Result + Addition, – Subtraction (also unary minus), * Multiplication, / Division, % Modulus, ++ Increment, += Addition assignment, –= Subtraction assignment ,*= Multiplication assignment /= Division assignment, %= Modulus assignment – – Decrement

- The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The Bitwise Operators

- Java defines several *bitwise operators* which can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**.
- These operators act upon the individual bits of their operands. They are summarized in the following table: Operator Result ~ Bitwise unary NOT & Bitwise AND | Bitwise OR ^ Bitwise exclusive OR >> Shift right >>> Shift right zero fill << Shift left &= Bitwise AND assignment |= Bitwise OR assignment ^= Bitwise exclusive OR assignment >>= Shift right assignment >>>= Shift right zero fill assignment <<= Shift left assignment

Relational Operators

- The *relational operators* determine the relationship that one operand has to the other.
- Specifically, they determine equality and ordering.

- The relational operators are shown here:
- Operator Result == Equal to , != Not equal to ,

> Greater than, < Less than, >= Greater than or equal to, <= Less than or equal to ,

The outcome of these operations is a **boolean** value.

The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

The Assignment Operator

- You have been using the assignment operator b4 now. Now it is time to take a formal look at it.
- The *assignment operator* is the single equal sign, **=**. The assignment operator works in Java much as it does in any other computer language. It has this general form:

  var = expression;

Here, the type of *var* must be compatible with the type of *expression.*

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

- int x, y, z; x = y = z = 100; // set x, y, and z to 100

- This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**.

- Using a —chain of assignment‖ is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if- then-else statements.

This operator is the **?**, and it works in Java much like it does in C, C++, and C#.

It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered.

The **?** has this general form:

expression1 **?** *expression2* **:** *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value.

- If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.

- The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**.

Type Conversion and Casting

- If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. 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 (from big to small e.g. byte to int).

- 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.

- 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.

Class exercises

int var1 = (double) 10; // not allowed

byte var2= (int)120.0 // allowed.120.0 gets cut to 120 which is then an integer which is within the range of a byte.

byte var3 = (int)33500.0 //not allowed bcos the value isnt within the range of a byte;

double var4= (int)18.2 // allowed . Gets cut to 18 which is an integer.

int var5 = 20f // not allowed, bcos u cannot convert from float to int.

however

int var5 = (int) 20f // is allowed since it is casted explicitly

// 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); } }
```