CLASSES AND OBJECTS

Classes and Objects

- Every **object** is an instance of a **class**, which serves as the type of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical members of a class in Java are the following:
  - **Instance variables**, which are also called **fields**, represent the data associated with an object of a class. Instance variables must have a type, which can either be a base type (such as int, float, or double) or any class type.
  - **Methods** in Java are blocks of code that can be called to perform actions. Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an **accessor** method, while an **update** method is one that may change one or more instance variables when called.

Another Example

```
public class Counter {
    private int count; // a simple integer instance variable
    public Counter( ) { } // default constructor (count is 0)
    public Counter(int initial) { count = initial; } // an alternate
    constructor
    public int getCount( ) { return count; } // an accessor method
    public void increment( ) { count++; } // an update method
    public void increment(int delta) { count += delta; } // an update
    method
    public void reset( ) { count = 0; } // an update method
}
```

- This class includes one instance variable, named count, which will have a default value of zero, unless we otherwise initialize it.
- The class includes two special methods known as constructors, one accessor method, and three update methods.

Creating and Using Objects

- Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable**.
- A reference variable is capable of storing the location (i.e., memory address) of an object from the declared class.
  - So we might assign it to reference an existing instance or a newly constructed instance.
  - A reference variable can also store a special value, null, that represents the lack of an object.
- In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class.
- A **constructor** is a method that always shares the same name as its class. The new operator returns a reference to the newly created instance; the returned reference is typically assigned to a variable for further use.

Continued Example

```
public class CounterDemo {
    public static void main(String[ ] args) {
    Counter c; // declares a variable; no counter yet constructed
    c = new Counter( ); // constructs a counter; assigns its reference to c
    c.increment( ); // increases its value by one
    c.increment(3); // increases its value by three more
    int temp = c.getCount( ); // will be 4
    c.reset( ); // value becomes 0
    Counter d = new Counter(5);// declares and constructs a counter having value
    5
    d.increment( ); // value becomes 6
    Counter e = d; // assigns e to reference the same object as d
    temp = e.getCount( ); // will be 6 (as e and d reference the same counter)
    e.increment(2); // value of e (also known as d) becomes 8
    }
}
```

- Here, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses.

The Dot Operator

- One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class.
- This access is performed with the dot (".") operator.
- We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters.

Wrapper Types

- There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with object types (not primitives).
- To get around this obstacle, Java defines a **wrapper** class for each base type.
  - Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic **boxing** and **unboxing**.

Example Wrapper Types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
int j = 8;
Integer a = new Integer(12);
int k = a; // implicit call to a.intValue()
int m = j + a; // a is automatically unboxed before the addition
a = 3 * m; // result is automatically boxed before assignment
Integer b = new Integer("-135"); // constructor accepts a String
int n = Integer.parseInt("2013"); // using static method of Integer
class
```

Literals

- A *literal is any "constant" value that can be used in an assignment or other expression.*
- Java allows the following kinds of literals:
  - null
  - true and false.
  - Integer, floating point, char , string

'\n'	**newline**	**'\t'**	**tab**
'\b'	**backspace**	**'\r'**	**return**
'\f'	**form feed**	**'\\'**	**backslash**
'\''	**single quote**	**'\"'**	**double quote**

Signatures

- If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types.
- A method's name combined with the number and types of its parameters is called a method's **signature,** for it takes all of these parts to determine the actual method to perform for a certain method call.
- A reference variable v can be viewed as a "pointer" to some object o.

Defining Classes

- A **class definition** is a block of code, delimited by braces "**{**" and "**}**" , within which is included declarations of instance variables and methods that are the members of the class.
- Immediately before the definition of a class, instance variable, or method in Java, keywords known as modifiers can be placed to convey additional stipulations about that definition.

Access Control Modifiers

- The **public** class modifier designates that all classes may access the defined aspect.
- The **protected** class modifier designates that access to the defined aspect is only granted to classes that are designated as subclasses of the given class through inheritance or in the same package.
- The **private** class modifier designates that access to a defined member of a class be granted only to code within that class.
- When a variable or method of a class is declared as **static**, it is associated with the class as a whole, rather than with each individual instance of that class.

Parameters

- A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method.
  - A parameter consists of two parts, the parameter type and the parameter name.
  - If a method has no parameters, then only an empty pair of parentheses is used.
- All parameters in Java are **passed by value**, that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body.
  - So if we pass an int variable to a method, then that variable's integer value is copied.
  - The method can change the copy but not the original.
  - If we pass an object reference as a parameter to a method, then the reference is copied as well.

The Keyword this

- Within the body of a method in Java, the keyword **this** is automatically defined as a reference to the instance upon which the method was invoked. There are three common uses:
  1. To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.
  2. To differentiate between an instance variable and a local variable with the same name.
  3. To allow one constructor body to invoke another constructor body.

Casting

- Casting is an operation that allows us to change the type of a value.
- We can take a value of one type and cast it into an equivalent value of another type.
- There are two forms of casting in Java: **explicit casting** and **implicit casting**.

Explicit Casting

- Java supports an explicit casting syntax with the following form:

  (type) exp

- Here "type" is the type that we would like the expression exp to have.
- This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type.

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1; // i1 gets value 3
int i2 = (int) d2; // i2 gets value 3
double d3 = (double) i2; // d3 gets value 3.0
```

Implicit Casting

- There are cases where Java will perform an implicit cast based upon the context of an expression.
- You can perform a **widening cast** between primitive types (such as from an int to a double), without explicit use of the casting operator.
- However, if attempting to do an implicit **narrowing cast**, a compiler error results.

```
int i1 = 42;
double d1 = i1; // d1 gets value 42.0
i1 = d1; // compile error: possible loss of precision
```

17

Sample Program

```
public class CreditCard {
// Instance variables:
private String customer; // name of the customer (e.g., ”John Bowman”)
private String bank; // name of the bank (e.g., ”California Savings”)
private String account; // account identifier (e.g., ”5391 0375 9387 5309”)
private int limit; // credit limit (measured in dollars)
protected double balance; // current balance (measured in dollars)
// Constructors:
public CreditCard(String cust, String bk, String acnt, int lim, double
initialBal) {
customer = cust;
    bank = bk;
    account = acnt;
    limit = lim;
    balance = initialBal;
}
public CreditCard(String cust, String bk, String acnt, int lim) {
    this(cust, bk, acnt, lim, 0.0); // use a balance of zero as default
}
```

18

Sample Program

```
// Accessor methods:
public String getCustomer( ) { return customer; }
public String getBank( ) { return bank; }
public String getAccount( ) { return account; }
public int getLimit( ) { return limit; }
public double getBalance( ) { return balance; }
// Update methods:
public boolean charge(double price) { // make a charge
     if (price + balance > limit) // if charge would surpass limit
     return false; // refuse the charge
     // at this point, the charge is successful
     balance += price; // update the balance
     return true; // announce the good news
}
public void makePayment(double amount) { // make a payment
balance -= amount;
}
// Utility method to print a card's information
public static void printSummary(CreditCard card) {
     System.out.println("Customer = " + card.customer);
     System.out.println("Bank = " + card.bank);
     System.out.println("Account = " + card.account);
     System.out.println("Balance = " + card.balance); // implicit cast
     System.out.println("Limit = " + card.limit); // implicit cast
}
// main method shown on next page...
```

19

Sample Program

```
public static void main(String[ ] args) {
     CreditCard[ ] wallet = new CreditCard[3];
     wallet[0] = new CreditCard("John Bowman", "California Savings", "5391 0375 9387 5309",
     5000);
     wallet[1] = new CreditCard("John Bowman", "California Federal","3485 0399 3395 1954",
     3500);
     wallet[2] = new CreditCard("John Bowman", "California Finance", "5391 0375 9387 5309",
     2500, 300);

     for (int val = 1; val <= 16; val++) {
          wallet[0].charge(3*val);
          wallet[1].charge(2*val);
          wallet[2].charge(val);
     }
     for (CreditCard card : wallet) {
          CreditCard.printSummary(card); // calling static method
     while (card.getBalance( ) > 200.0) {
          card.makePayment(200);
     System.out.println("New balance = " + card.getBalance( ));
}
}
}
}
```

20

Packages

- The Java language takes a general and useful approach to the organization of classes into programs. Every stand-alone public class defined in Java must be given in a separate file. The file name is the name of the class with a .java extension. So a class declared as public class Window is defined in a file Window.java. That file may contain definitions for other stand-alone classes, but none of them may be declared with public visibility.
- To aid in the organization of large code repository, Java allows a group of related type definitions (such as classes and enums) to be grouped into what is known as a ***package. For types to belong to a package named packageName, their source*** code must all be located in a directory named *packageName and each file must* begin with the line:

  package *packageName;*

21

Import Statement

- A type can be refered within a package using its fully qualified name. For example, the Scanner class is defined in the java.util package, and can refer to it as java.util.Scanner.
- We could declare and construct a new instance of that class in a project using the following statement:

  java.util.Scanner input = new java.util.Scanner(System.in);
- However, all the extra typing needed to refer to a class outside of the current package can get tiring. In Java, we can use the import keyword to include external classes or entire packages in the current file. To import an individual class from a specific package, we type the following at the beginning of the file:

  import *packageName.className;*
- Ex:

  import java.util.Scanner;
- and then we were allowed to use the less burdensome syntax:

  Scanner input = new Scanner(System.in);

  ❑To import all the classes from a specific package, we type the following at the beginning of the file import *packageName.*;*

  import java.util.*;

22