JAVA PROGRAMMING

UNIT-I

Java History

- Computer language innovation and development occurs for two fundamental reasons:

  1) to adapt to changing environments and uses

  2) to implement improvements in the art of programming
- The development of Java was driven by both in equal measures.
- Many Java features are inherited from the earlier

languages:

B → C → C++ → Java

Before Java: C

- Designed by Dennis Ritchie in 1970s.
- Before C: BASIC, COBOL, FORTRAN, PASCAL
- C- structured, efficient, high-level language that could replace assembly code when creating systems programs.
- Designed, implemented and tested by programmers.

Before Java: C++

- **Designed by Bjarne Stroustrup in 1979.**
- **Response to the increased complexity of programs and respective improvements in the programming paradigms and methods:**
  - **1) assembler languages**
  - **2) high-level languages**
  - **3) structured programming**
  - **4) object-oriented programming (OOP)**
- **OOP – methodology that helps organize complex programs through the use of inheritance, encapsulation and polymorphism.**
- **C++ extends C by adding object-oriented features.**

Java: History

- In 1990, Sun Microsystems started a project called Green.
- Objective: to develop software for consumer electronics.
- Project was assigned to James Gosling, a veteran of classic network software design. Others included Patrick Naughton, ChrisWarth, Ed Frank, and Mike Sheridan.
- The team started writing programs in C++ for embedding into
  - toasters
  - washing machines
  - VCR's
- Aim was to make these appliances more "intelligent".

Java: History (contd.)

- C++ is powerful, but also dangerous. The power and popularity of C derived from the extensive use of pointers. However, any incorrect use of pointers can cause memory leaks, leading the program to crash.
- In a complex program, such memory leaks are often hard to detect.
- Robustness is essential. Users have come to expect that Windows may crash or that a program running under Windows may crash. ("This program has performed an illegal operation and will be shut down")
- However, users do not expect toasters to crash, or washing machines to crash.
- A design for consumer electronics has to be *robust*.
- Replacing pointers by references, and automating memory management was the proposed solution.

Java: History (contd.)

- Hence, the team built a new programming language called Oak, which avoided potentially dangerous constructs in C++, such as pointers, pointer arithmetic, operator overloading etc.
- Introduced automatic memory management, freeing the programmer to concentrate on other things.
- Architecture neutrality (Platform independence)
- Many different CPU's are used as controllers. Hardware chips are evolving rapidly. As better chips become available, older chips become obsolete and their production is stopped. Manufacturers of toasters and washing machines would like to use the chips available off the shelf, and would not like to reinvest in compiler development every two-three years.
- So, the software and programming language had to be *architecture neutral*.

Java: History (contd)

- It was soon realized that these design goals of consumer electronics perfectly suited an ideal programming language for the Internet and WWW, which should be:
    - ❖ object-oriented (& support GUI)
    - ❖ – robust
    - ❖ – architecture neutral
- Internet programming presented a BIG business opportunity. Much bigger than programming for consumer electronics.
- Java was "re-targeted" for the Internet
- The team was expanded to include Bill Joy (developer of Unix), Arthur van Hoff, Jonathan Payne, Frank Yellin, Tim Lindholm etc.
- In 1994, an early web browser called WebRunner was written in Oak. WebRunner was later renamed HotJava.
- In 1995, Oak was renamed Java.
- A common story is that the name Java relates to the place from where the development team got its coffee. The name Java survived the trade mark

search.

Java History

- Designed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun Microsystems in 1991.
- The original motivation is not Internet: platform-independent software embedded in consumer electronics devices.
- With Internet, the urgent need appeared to break the fortified positions of Intel, Macintosh and Unix programmer communities.
- Java as an “Internet version of C++”? No.
- Java was not designed to replace C++, but to solve a

different set of problems.

The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:
    - ❖ Simple
    - ❖ Secure
    - ❖ Portable
    - ❖ Object-oriented
    - ❖ Robust
    - ❖ Multithreaded
    - ❖ Architecture-neutral
    - ❖ Interpreted
    - ❖ High performance
    - ❖ Distributed

❖ Dynamic

- **simple** – Java is designed to be easy for the professional programmer to learn and use.
- **object-oriented:** a clean, usable, pragmatic approach to objects, not restricted by the need for compatibility with other languages.
- **Robust:** restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.
- **Multithreaded:** supports multi-threaded programming for writing program that perform concurrent computations

- **Architecture-neutral:** Java Virtual Machine provides a platform independent environment for the execution of Java byte code
- **Interpreted and high-performance:** Java programs are compiled into an intermediate representation – byte code:

  a) can be later interpreted by any JVM

  b) can be also translated into the native machine code for efficiency.

- **Distributed:** Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file.
- **Dynamic:** substantial amounts of run-time type information to verify and resolve access to objects at run-time.
- **Secure:** programs are confined to the Java execution environment and cannot access other parts of the computer.

- **Portability:** Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet.
- For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. The same mechanism that helps ensure security also helps create portability.
- Indeed, Java's solution to these two problems is

both elegant and efficient.

Data Types

- Java defines eight simple types:
  1)byte – 8-bit integer type
  2)short – 16-bit integer type
  3)int – 32-bit integer type
  4)long – 64-bit integer type
  5)float – 32-bit floating-point type
  6)double – 64-bit floating-point type
  7)char – symbols in a character set
  8)boolean – logical values true and false

- byte: 8-bit integer type.

  Range: -128 to 127.

  Example: byte b = -15;

  Usage: particularly when working with data streams.
- short: 16-bit integer type.

  Range: -32768 to 32767.

  Example: short c = 1000;

  Usage: probably the least used simple type.

- **int:** 32-bit integer type.

  Range: -2147483648 to 2147483647.

  Example: int b = -50000;

  Usage:

  1) Most common integer type.
  2) Typically used to control loops and to index arrays.
  3) Expressions involving the byte, short and int values are promoted to int before calculation.

- **long:** 64-bit integer type.
  Range: -9223372036854775808 to 9223372036854775807.
  Example: long l = 10000000000000000;
  Usage: 1) useful when int type is not large enough to hold the desired value
- **float:** 32-bit floating-point number.
  Range: 1.4e-045 to 3.4e+038.
  Example: float f = 1.5;
  Usage:

1) fractional part is needed
2) large degree of precision is not required

- **double:** 64-bit floating-point number.

  Range: 4.9e-324 to 1.8e+308.

  Example: double pi = 3.1416;

  Usage:

  1) accuracy over many iterative calculations

  2) manipulation of large-valued numbers

char: 16-bit data type used to store characters.

Range: 0 to 65536.

Example: char c = ‘a’;

Usage:

1) Represents both ASCII and Unicode character sets; Unicode defines a

character set with characters found in (almost) all human languages.

2) Not the same as in C/C++ where char is 8-bit

and represents ASCII only.

- **boolean:** Two-valued type of logical values.

  Range: values true and false.

  Example: boolean b = (1<2);

  Usage:

  1) returned by relational operators, such as 1<2

  2) required by branching expressions such as if or for

Variables

- declaration – how to assign a type to a variable
- initialization – how to give an initial value to a variable
- scope – how the variable is visible to other parts of the program
- lifetime – how the variable is created, used and destroyed
- type conversion – how Java handles automatic type conversion
- type casting – how the type of a variable can be narrowed down

- type promotion – how the type of a variable can be expanded

Variables

- Java uses variables to store data.
- To allocate memory space for a variable JVM requires:
  1) to specify the data type of the variable
  2) to associate an identifier with the variable
  3) optionally, the variable may be assigned an initial value

- All done as part of variable declaration.

Basic Variable Declaration

- datatype identifier [=value];
- datatype must be
  - A simple datatype
  - User defined datatype (class type)
- Identifier is a recognizable name confirm to identifier rules
- Value is an optional initial value.

Variable Declaration

- We can declare several variables at the same time:

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

  Examples:

```
int a, b, c;
int d = 3, e, f = 5;
byte g = 22;
double pi = 3.14159;
char ch = 'x';
```

Variable Scope

- Scope determines the visibility of program elements with respect to other program elements.
- In Java, scope is defined separately for classes and methods:

  1) variables defined by a class have a global scope

  2) variables defined by a method have a local scope

  A scope is defined by a block:

  ```
  {
  …
  }
  ```

  A variable declared inside the scope is not visible outside:

  ```
  {
  int n;
  ```

```
}
n = 1;// this is illegal
```

L 2.5

Variable Lifetime

- Variables are created when their scope is entered by control flow and destroyed when their scope is left:
- A variable declared in a method will not hold its value between different invocations of this method.
- A variable declared in a block looses its value when the block is left.
- Initialized in a block, a variable will be re-initialized with every re-entry. Variables lifetime

is confined to its scope!

Arrays

- An array is a group of liked-typed variables referred to by a common
- name, with individual variables accessed by their index.
- Arrays are:
    1) declared
    2) created
    3) initialized
    4) used
- Also, arrays can have one or several dimensions.

Array Declaration

- Array declaration involves:

1) declaring an array identifier
2) declaring the number of dimensions
3) declaring the data type of the array elements

- Two styles of array declaration:

```
type array-variable[];
          or
type [] array-variable;
```

Array Creation

- After declaration, no array actually exists.
- In order to create an array, we use the new operator:

```
type array-variable[];
array-variable = new type[size];
```

- This creates a new array to hold size elements of type type, which reference will be kept in the variable array-variable.

Array Indexing

- Later we can refer to the elements of this array through their indexes:
- array-variable[index]
- The array index always starts with zero!
- The Java run-time system makes sure that all array indexes are in the correct range, otherwise raises a run-time error.

Array Initialization

- Arrays can be initialized when they are declared:
- int monthDays[] = {31,28,31,30,31,30,31,31,30,31,30,31};
- Note:

1) there is no need to use the new operator

2) the array is created large enough to hold all specified elements

Multidimensional Arrays

- Multidimensional arrays are arrays of arrays:
  1) declaration: int array[][];
  2) creation: int array = new int[2][3];
  3) initialization

     int array[][] = { {1, 2, 3}, {4, 5, 6} };

Operators Types

- Java operators are used to build value expressions.
- Java provides a rich set of operators:
  1) assignment
  2) arithmetic
  3) relational
  4) logical
  5) bitwise

Arithmetic assignments

+=	v += expr;	v = v + expr ;
-=	v -=expr;	v = v - expr ;
*=	v *= expr;	v = v * expr ;
/=	v /= expr;	v = v / expr ;
%=	v %= expr;	v = v % expr ;

Basic Arithmetic Operators

+	op1 + op2	ADD
-	op1 - op2	SUBSTRACT
*	op1 * op2	MULTIPLY
/	op1 / op2	DIVISION
%	op1 % op2	REMAINDER

Relational operator

==	Equals to	Apply to any type
!=	Not equals to	Apply to any type
>	Greater than	Apply to numerical type
<	Less than	Apply to numerical type
>=	Greater than or equal	Apply to numerical type
<=	Less than or equal	Apply to numerical type

Logical operators

&	op1 & op2	Logical AND
|	op1 | op2	Logical OR
&&	op1 && op2	Short-circuit AND
||	op1 || op2	Short-circuit OR
!	! op	Logical NOT
^	op1 ^ op2	Logical XOR

Bit wise operators

~	~op	Inverts all bits
&	op1 & op2	Produces 1 bit if both operands are 1
|	op1 |op2	Produces 1 bit if either operand is 1
^	op1 ^ op2	Produces 1 bit if exactly one operand is 1
>>	op1 >> op2	Shifts all bits in op1 right by the value of op2
<<	op1 << op2	Shifts all bits in op1 left by the value of op2

Expressions

- An *expression* is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language, that evaluates to a single value.
- Examples of expressions are in bold below:

  int **number = 0**;

  anArray[0] = 100;

  System.out.println (**"Element 1 at index 0: " + anArray[0]**);

  int **result = 1 + 2**; // result is now 3 if(**value1 == value2**)

  System.out.println(**"value1 == value2"**);

Expressions

- The data type of the value returned by an expression depends on the elements used in the expression.
- The expression number = 0 returns an int because the assignment operator returns a value of the same data type as its left-hand operand; in this case, number is an int.
- As you can see from the other expressions, an expression can return other types of values as well, such as boolean or String.
- The Java programming language allows you to construct compound expressions from various smaller expressions as long as the data type required by one part of the expression matches the data type of the other.
- Here's an example of a compound expression: 1 * 2 * 3

Control Statements

- Java control statements cause the flow of execution to advance and branch based on the changes to the state of the program.
- Control statements are divided into three groups:
- 1) selection statements allow the program to choose different parts of the execution based on the outcome of an expression
- 2) iteration statements enable program execution to repeat one or more statements
- 3) jump statements enable your program to execute in a non-linear fashion

Selection Statements

- Java selection statements allow to control the flow of program’s execution based upon conditions known only during run-time.
- Java provides four selection statements:

  1) if
  2) if-else
  3) if-else-if
  4) switch

Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:
  1) while
  2) do-while
  3) for

Jump Statements

- Java jump statements enable transfer of control to other parts of program.
- Java provides three jump statements:
    1) break
    2) continue
    3) return
- In addition, Java supports exception handling that can also alter the control flow of a program.

Type Conversion

- Size Direction of Data Type
  - Widening Type Conversion (Casting down)
    - Smaller Data Type → Larger Data Type
  - Narrowing Type Conversion (Casting up)
    - Larger Data Type → Smaller Data Type
- Conversion done in two ways
  - Implicit type conversion
    - Carried out by compiler automatically
  - Explicit type conversion
    - Carried out by programmer using casting

Type Conversion

- Widening Type Converstion
  - Implicit conversion by compiler automatically

byte -> short, int, long, float, double
short -> int, long, float, double
char -> int, long, float, double
int -> long, float, double
long -> float, double
float -> double

L 3.6

Type Conversion

- Narrowing Type Conversion
  - Programmer should describe the conversion explicitly

byte -> char
short -> byte, char
char -> byte, short
int -> byte, short, char
long -> byte, short, char, int
float -> byte, short, char, int, long
double -> byte, short, char, int, long, float

L 3.7

Type Conversion

- byte and short are always promoted to int
- if one operand is long, the whole expression is promoted to long
- if one operand is float, the entire expression is promoted to float
- if any operand is double, the result is double

Type Casting

- General form: (targetType) value
- Examples:
- 1) integer value will be reduced module bytes range:

```
int i;
byte b = (byte) i;
```

- 2) floating-point value will be truncated to integer value:

```
float f;
int i = (int) f;
```

Simple Java Program

- A class to display a simple message:

```
      class MyProgram
      {
      public static void main(String[] args)
       {
System.out.println(“First Java program.");
      }
}
```

What is an Object?

- Real world objects are things that have:
  1) state
  2) behavior
  Example: your dog:
- state – name, color, breed, sits?, barks?, wages tail?, runs?
- behavior – sitting, barking, waging tail, running
- A software object is a bundle of variables (state) and methods (operations).

What is a Class?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

Object Creation

- A variable is declared to refer to the objects of type/class String:

  String s;
- The value of s is null; it does not yet refer to any object.
- A new String object is created in memory with initial “abc” value:
- String s = new String(“abc”);
- Now s contains the address of this new object.

Object Destruction

- A program accumulates memory through its execution.
- Two mechanism to free memory that is no longer need by the program:
  1) manual – done in C/C++
  2) automatic – done in Java
- In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.
- Garbage collector is parts of the Java Run-Time Environment.

Class

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:
- A class is a template for objects
- An object is an instance of a class

Class Definition

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:

```
class classname {
        type instance-variable-1;
        …
        type instance-variable-n;
        type method-name-1(parameter-list) { … }
        type method-name-2(parameter-list) { … }
        …
        type method-name-m(parameter-list) { … }
    }
```

Example: Class Usage

```
class Box {
double width;
double height;
double depth;
}
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
vol = mybox.width * mybox.height * mybox.depth;
System.out.println ("Volume is " + vol);
} }
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.
    1) it is syntactically similar to a method:
    2) it has the same name as the name of its class
    3) it is written without return type; the default return type of a class
- constructor is the same class
- When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

```
class Box {
double width;
double height;
double depth;
Box() {
System.out.println("Constructing Box");
width = 10; height = 10; depth = 10;
}
double volume() {
return width * height * depth;
}
}
```

L

Parameterized Constructor

```
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
double volume()
 { return width * height * depth;
}
}
```

Methods

- General form of a method definition:

```
        type name(parameter-list) {
                                  … return value;
                                  …
                              }
```

- Components:

  1) type - type of values returned by the method. If a method does not return any value, its return type must be void.

  2) name is the name of the method

  3) parameter-list is a sequence of type-identifier lists separated by commas

  4) return value indicates what value is returned by the

method.

Example: Method

- Classes declare methods to hide their internal data structures, as well as for their own internal use: Within a class, we can refer directly to its member variables:

```
class Box {
double width, height, depth;
void volume() {
System.out.print("Volume is ");
System.out.println(width * height * depth);
}
}
```

Parameterized Method

- Parameters increase generality and applicability of a method:
- 1) method without parameters

```
int square() { return 10*10; }
```

- 2) method with parameters

```
int square(int i) { return i*i; }
```

- Parameter: a variable receiving value at the time the method is invoked.
- Argument: a value passed to the method when it is invoked.

Access Control: Data Hiding and Encapsulation

- Java provides control over the *visibility* of variables and methods.
- *Encapsulation,* safely sealing data within the *capsule* of the class Prevents programmers from relying on details of class implementation, so you can update without worry
- Helps in protecting against accidental or wrong usage.
- Keeps code elegant and clean (easier to maintain)

Access Modifiers: Public, Private, Protected

- *Public:* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.
- Default(No visibility modifier is specified): it behaves like public in its package and private in other packages.
- *Default Public* keyword applied to a class, makes it available/visible everywhere. Applied to a method or variable, completely visible.

- *Private* fields or methods for a class only visible within that class. Private members are *not* visible within subclasses, and are *not* inherited.
- *Protected* members of a class are visible within the class, subclasses and *also* within all classes that are in the same package as that class.

Visibility

```
public class Circle {
     private double x,y,r;

          // Constructor
      public Circle (double x, double y, double r) {
               this.x = x;
               this.y = y;
               this.r = r;
      }
      //Methods to return circumference and area
     public double circumference() { return 2*3.14*r;}
     public double area() { return 3.14 * r * r; }
}
```

Keyword this

- Can be used by any object to refer to itself in any class method
- Typically used to
  - Avoid variable name collisions
  - Pass the receiver as an argument
  - Chain constructors

Keyword this

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:

```
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

Garbage Collection

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:
- 1) The garbage collector keeps track of how many references an object has.
- 2) It removes an object from memory when it has no longer any references.
- 3) Thereafter, the memory occupied by the object can be allocated again.
- 4) The garbage collector invokes the finalize method.

finalize() Method

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:
- 1) implemented inside a class as:

```
protected void finalize() { … }
```

- 2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out

Method Overloading

- It is legal for a class to have two or more methods with the same name.
- However, Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:
- 1) by the number of arguments, or
- 2) by the types of arguments
- Overloading and inheritance are two ways to

implement polymorphism.

Example: Overloading

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
double test(double a) {
System.out.println("double a: " + a); return a*a;
}
}
```

Constructor Overloading

```
class Box {
double width, height, depth;
Box(double w, double h, double d) {
width = w; height = h; depth = d;
}
Box() {
width = -1; height = -1; depth = -1;
}
Box(double len) {
width = height = depth = len;
}
double volume() { return width * height * depth; }
}
```

Parameter Passing

- Two types of variables:
  1) simple types
  2) class types
- Two corresponding ways of how the arguments are passed to methods:
- 1) by value a method receives a cope of the original value; parameters of simple types
- 2) by reference a method receives the memory address of the original value, not the value itself; parameters of class types

Call by value

```
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.print("a and b before call: “);
System.out.println(a + " " + b);
ob.meth(a, b);
System.out.print("a and b after call: ");
System.out.println(a + " " + b);
}
}
```

Call by refference

- **As the parameter hold the same address as the argument, changes to the object inside the method do affect the object used by the argument:**

```
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.print("ob.a and ob.b before call: “);
System.out.println(ob.a + " " + ob.b);
ob.meth(ob);
System.out.print("ob.a and ob.b after call: ");
System.out.println(ob.a + " " + ob.b);
}
}
```

Recursion

- A recursive method is a method that calls itself:

1) all method parameters and local variables are allocated on the stack
2) arguments are prepared in the corresponding parameter positions
3) the method code is executed for the new arguments
4) upon return, all parameters and variables are removed from the stack
5) the execution continues immediately after the invocation point

Example: Recursion

```
class Factorial {
int fact(int n) {
if (n==1) return 1;
return fact(n-1) * n;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.print("Factorial of 5 is ");
System.out.println(f.fact(5));
} }
```

L 8.2

String Handling

- **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects.
- For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** constant

- Java defines one operator for **String** objects: **+**.
- It is used to concatenate two strings. For example, this statement
- String myString = "I" + " like " + "Java.";

  results in **myString** containing

  "I like Java."

- The **String** class contains several methods that you can use. Here are a few. You can
- test two strings for equality by using

  equals( ). You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:
- // Demonstrating some String methods.

```
      class StringDemo2 {
      public static void main(String args[]) {
            String strOb1 = "First String";
            String strOb2 = "Second String";
            String strOb3 = strOb1;
System.out.println("Length of strOb1: " +
                              strOb1.length());
System.out.println ("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))

System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
} }
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
```

```
strOb1 != strOb2
strOb1 == strOb3
```

JAVA PROGRAMMING

UNIT-II

Inheritance

- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes
  - deriving new classes from existing classes
  - the `protected` modifier
  - creating class hierarchies
  - abstract classes
  - indirect visibility of inherited members

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Inheritance and Visibility

Designing for Inheritance

Inheritance and GUIs

Creating a Subclass

- *A class is to an Object what a blueprint is to a house*
- A class establishes the characteristics and the behaviors of the object
- No memory space is reserved for the data (variables)
- Classes are the plan; objects are the embodiment of that plan
- Many houses can be built from the same blueprint

Inheritance

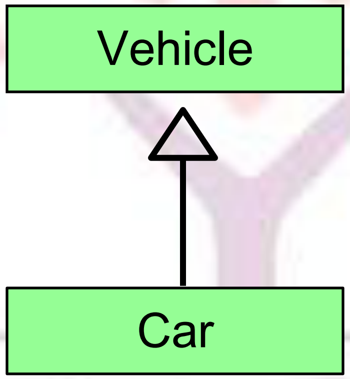
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class,* or *super class*, or *base class*
- The derived class is called the *child class* or *subclass*
- As the name implies, the child inherits characteristics of the parent

Inheritance

- That is, the child class inherits the methods and data defined by the parent class
- We can refer to these inherited methods and variables as if they were declared locally in the class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent

Inheritance

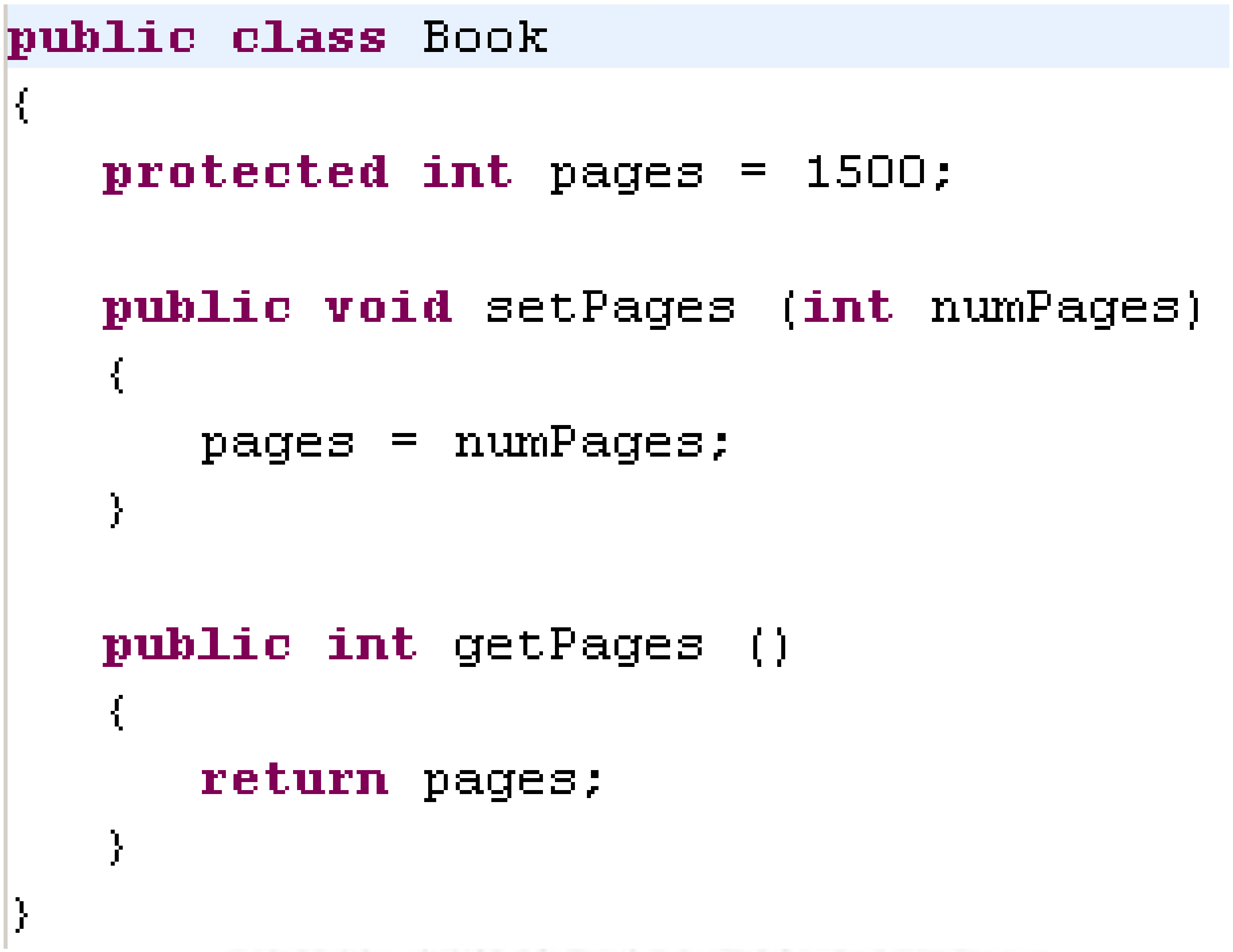
- A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- *Software reuse* is a fundamental benefit of inheritance
- By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

- In Java, we use the reserved word `extends` to establish an inheritance relationship

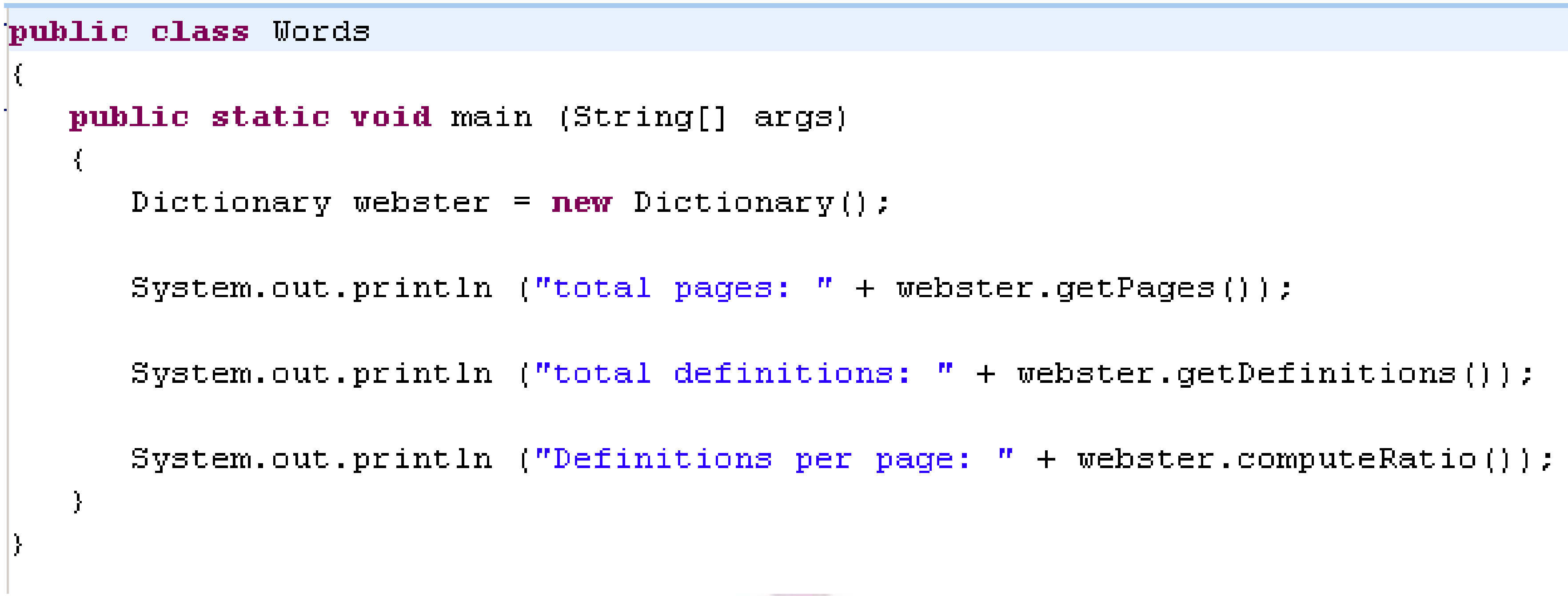
```
class Car extends Vehicle
{
   // class contents
}
```

- See `Words.java` (page 440) Listing 8.1
- See `Book.java` (page 441) Listing 8.2
- See `Dictionary.java` (page 442) Listing 8.3



```
public class Book
{
    protected int pages = 1500;

    public void setPages (int numPages)
    {
        pages = numPages;
    }

    public int getPages ()
    {
        return pages;
    }
}
```

```
public class Dictionary extends Book
{
   private int definitions = 52500;

   public double computeRatio()
   {
      return definitions/pages;
   }

   public void setDefinitions (int numDefinitions)
   {
      definitions = numDefinitions;
   }

   public int getDefinitions ()
   {
      return definitions;
   }
}
```



```
public class Words
{
    public static void main (String[] args)
    {
        Dictionary webster = new Dictionary();

        System.out.println ("total pages: " + webster.getPages());

        System.out.println ("total definitions: " + webster.getDefinitions());

        System.out.println ("Definitions per page: " + webster.computeRatio());
    }
}
```

The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with private visibility cannot be referenced by name in a child class
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a child class to reference a variable or method directly in the child class

- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility

- A protected variable is visible to any class in the same package as the parent class

- The details of all Java modifiers are discussed

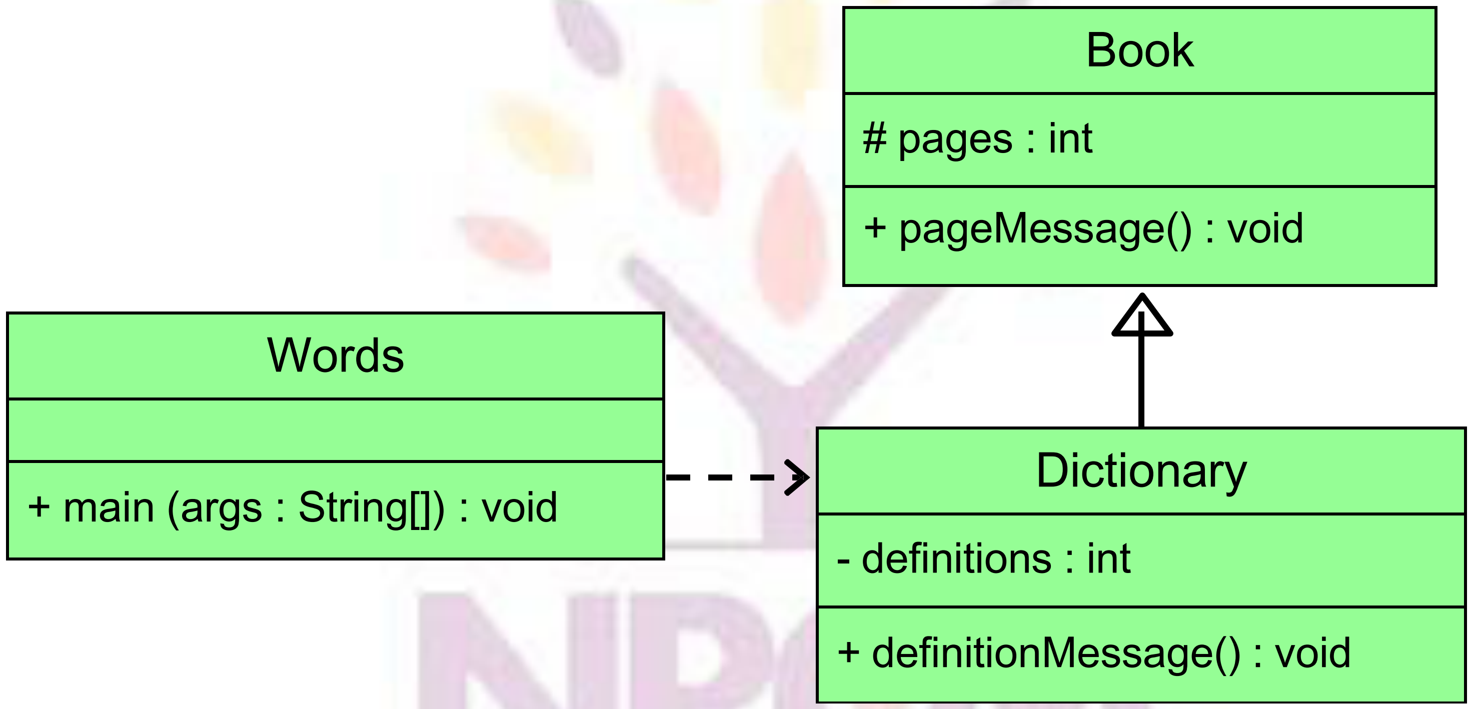
The protected Modifier

- Protected variables and methods can be shown with a hash (#)symbol preceding them in UML diagrams

- NOTE:
  - All methods & variables (even those declared private) are inherited by the child class
  - Their definitions exist and memory is reserved for the variables

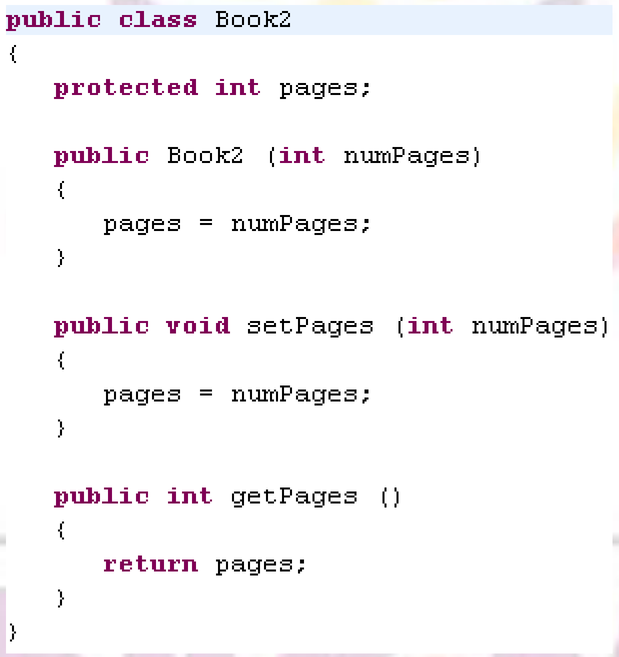
– However they CANNOT be referenced by name

Class Diagram for Words

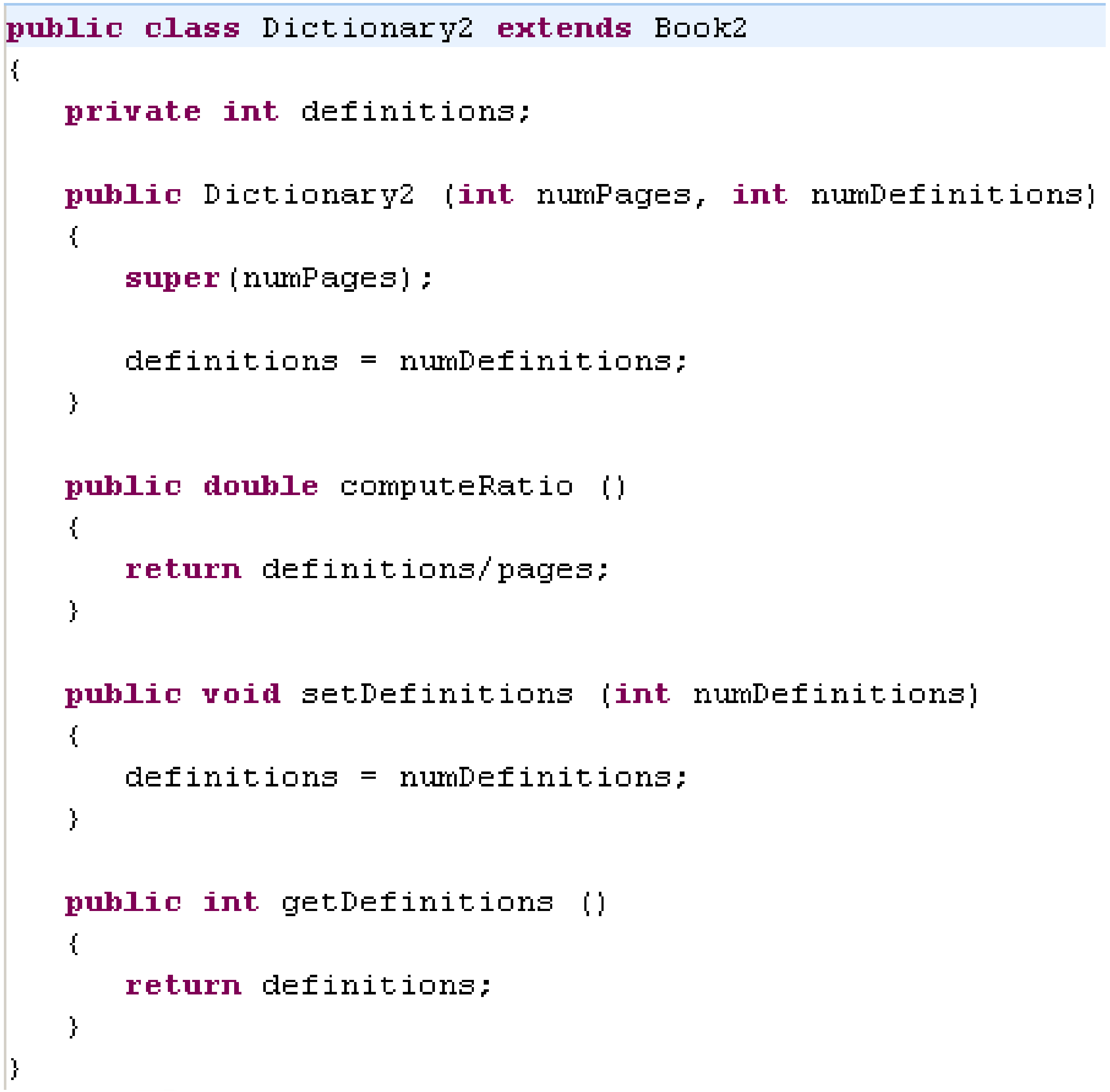


The super Reference

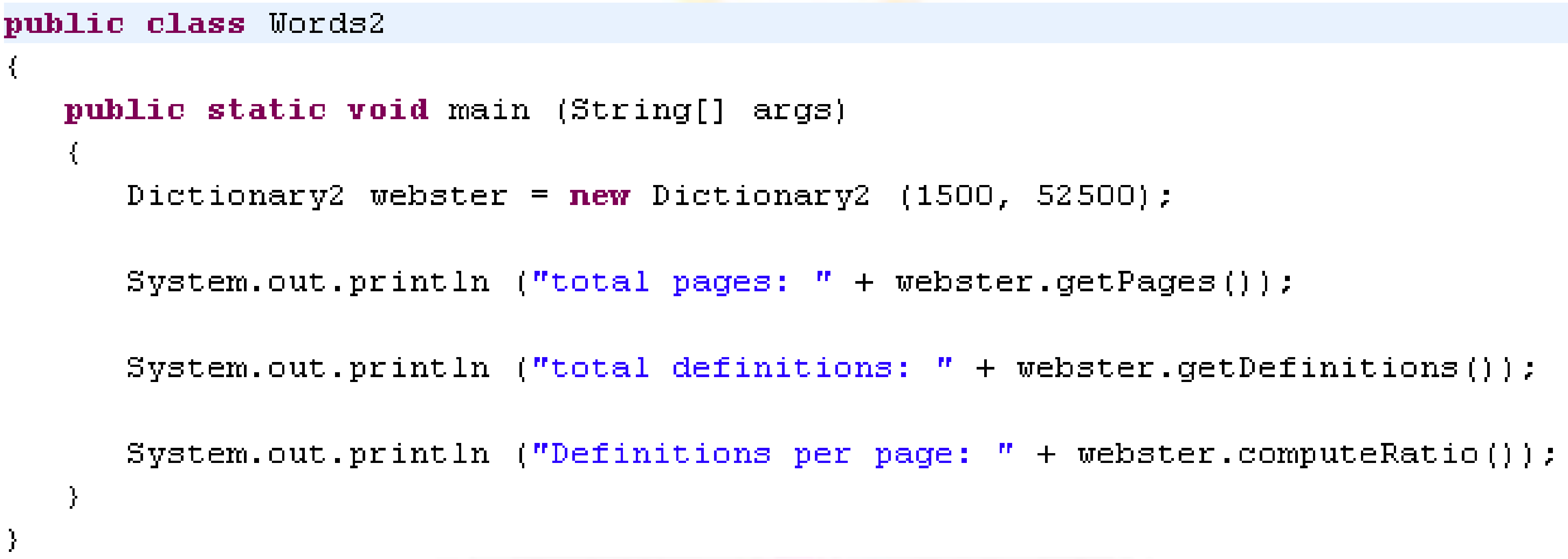
- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- See `Words2.java` (page 445) Listing 8.4



```java
public class Book2
{
   protected int pages;

   public Book2 (int numPages)
   {
      pages = numPages;
   }

   public void setPages (int numPages)
   {
      pages = numPages;
   }

   public int getPages ()
   {
      return pages;
   }
}
```



```java
public class Dictionary2 extends Book2
{
   private int definitions;

   public Dictionary2 (int numPages, int numDefinitions)
   {
      super(numPages);

      definitions = numDefinitions;
   }

   public double computeRatio ()
   {
      return definitions/pages;
   }

   public void setDefinitions (int numDefinitions)
   {
      definitions = numDefinitions;
   }

   public int getDefinitions ()
   {
      return definitions;
   }
}
```



```
public class Words2
{
   public static void main (String[] args)
   {
      Dictionary2 webster = new Dictionary2 (1500, 52500);

      System.out.println ("total pages: " + webster.getPages());

      System.out.println ("total definitions: " + webster.getDefinitions());

      System.out.println ("Definitions per page: " + webster.computeRatio());
   }
}
```

The super Reference

- A child's constructor is responsible for calling the parent's constructor

- If the child constructor invokes the parent (constructor) by using the `super` reference, it
  MUST be the first line of code of the constructor

- The `super` reference can also be used to reference other variables and methods defined in the parent class

Multiple Inheritance

- Java supports *single inheritance*, meaning that a derived class can have only one parent class
- *Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Inheritance and Visibility

Designing for Inheritance

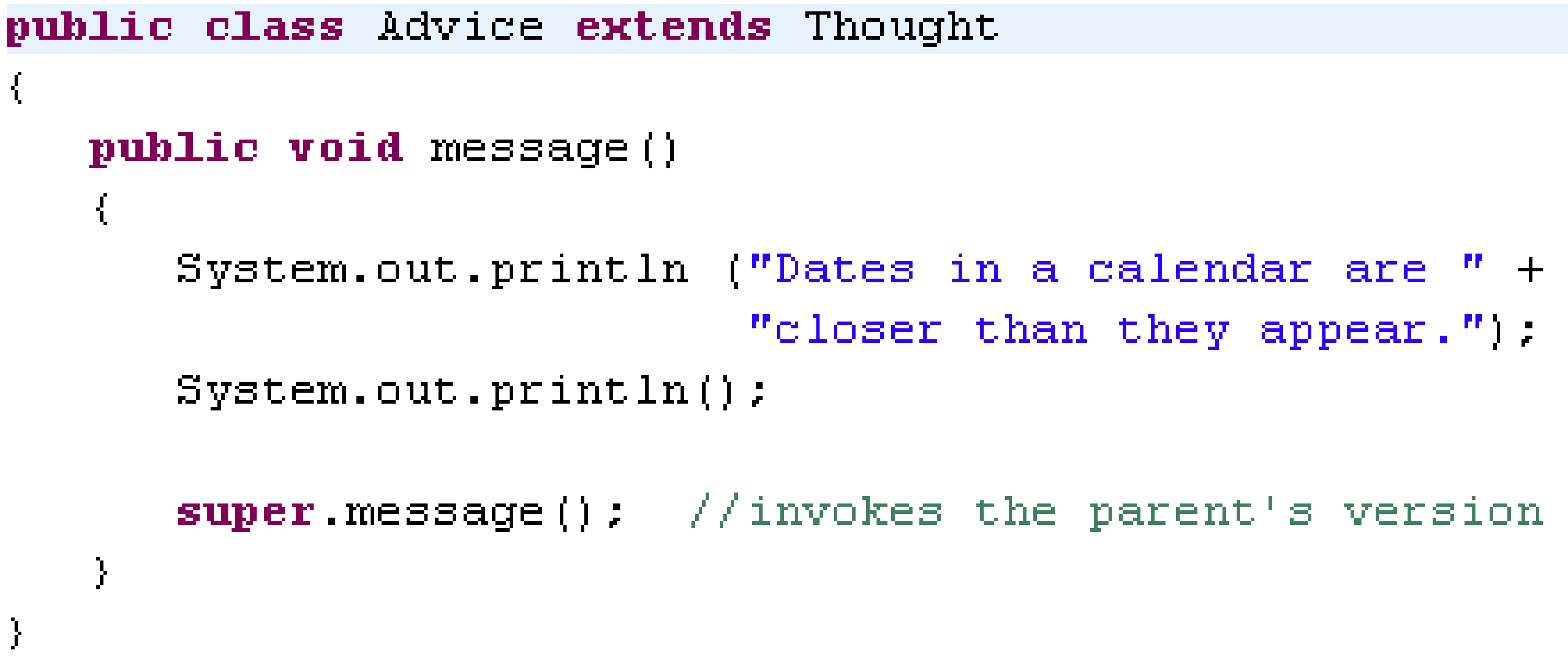
Inheritance and GUIs

The `Timer` Class

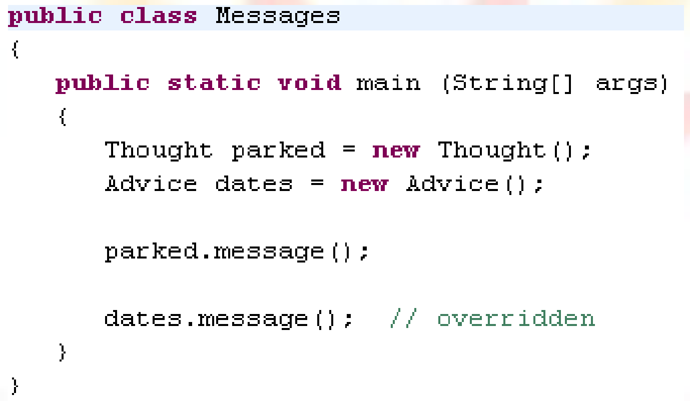
Overriding Methods

- A child class can *override* the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- See `Messages.java` (page 450) Listing 8.7

```
public class Thought
{
    public void message()
    {
        System.out.println ("I'm diagonally parked " +
                            "in a parallel universe.");
        System.out.println();
    }
}
```



```
public class Advice extends Thought
{
   public void message()
   {
      System.out.println ("Dates in a calendar are " +
                          "closer than they appear.");
      System.out.println();

      super.message();  //invokes the parent's version
   }
}
```



```
public class Messages
{
   public static void main (String[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message();  // overridden
   }
}
```

Overriding

- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause unnecessarily

c
o
n
f
u
s
i
n
g

code

8-28

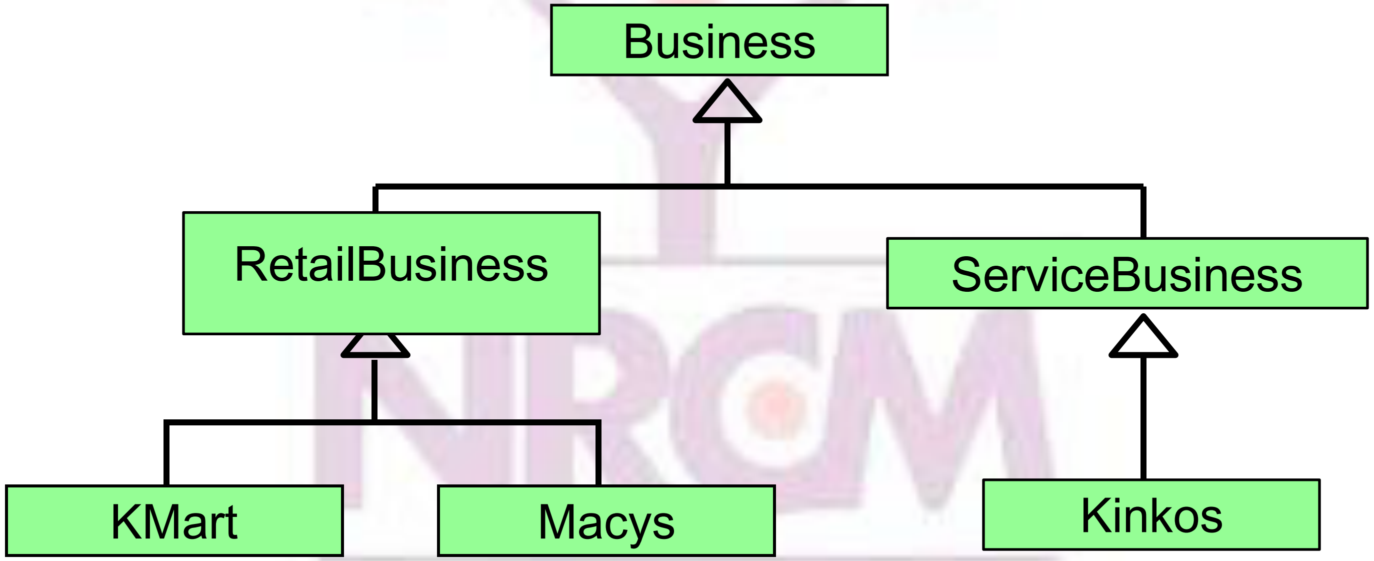
Overloading vs. Overriding

- Overloading deals with multiple methods with the same name in the same class, but with different signatures
- Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature
- Overloading lets you define a similar

operation in different ways for different parameters

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable
- An inherited member is passed continually down the line
- Therefore, a child class inherits from all its

ancestor classes

The Object Class

- A class called `Object` is defined in the `java.lang` package of the Java standard class library

- All classes are derived from the `Object` class

- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains a few useful methods, which are inherited by all classes
- For example, the `toString` method is defined in the `Object` class
- Every time we define the `toString` method, we are actually overriding an inherited definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with some other information

The Object Class

- The `equals` method of the `Object` class returns true if two references are aliases
- We can override `equals` in any class to define equality in some more appropriate way
- As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters
- The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class cannot be instantiated
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
{
    // contents
}
```

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains non-abstract methods with full definitions
- A class declared as abstract does not have to

contain abstract methods -- simply declaring

Abstract Classes

- The child of an abstract class <u>must override</u> the abstract methods of the parent (define it), or it too will be considered abstract
- An abstract method cannot be defined as `final` or `static`
- The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too generic to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces

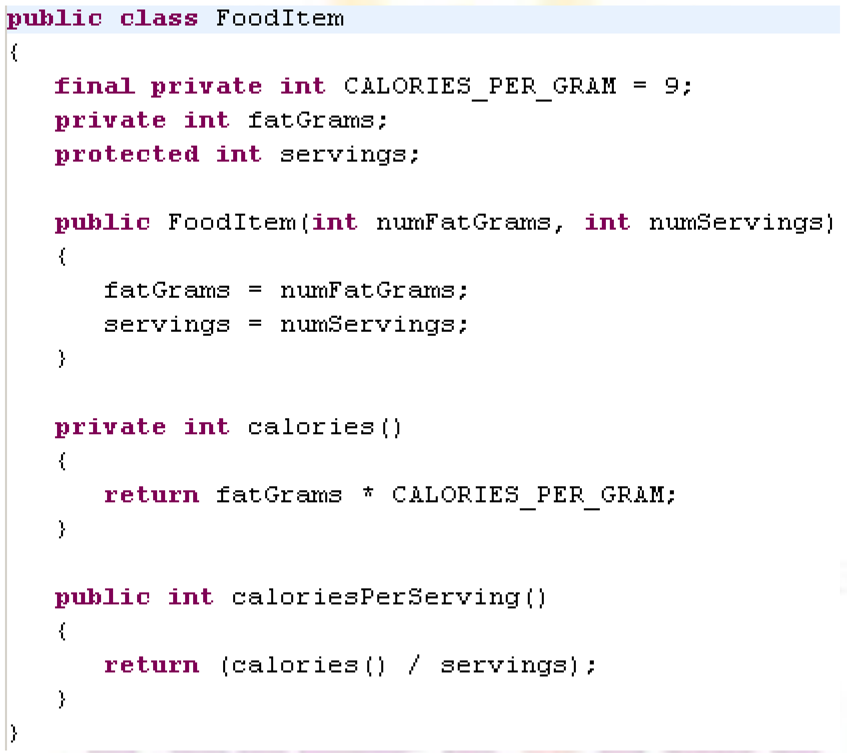
Visibility Revisited

- It's important to understand one subtle issue related to inheritance and visibility
- All variables and methods of a parent class, even private members, are inherited by its children
- As we've mentioned, private members cannot be referenced by name in the child class
- However, private members inherited by child

classes exist and can be referenced indirectly

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
- The `super` reference can be used to refer to the parent class, even if no object of the parent exists
- See `FoodAnalyzer.java` (page 459) Listing 8.10
- See `FoodItem.java` (page 460) Listing 8.11


```
public class FoodItem
{
   final private int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   public FoodItem(int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }

   private int calories()
   {
      return fatGrams * CALORIES_PER_GRAM;
   }

   public int caloriesPerServing()
   {
      return (calories() / servings);
   }
}
```

```
public class Pizza extends FoodItem
{
    public Pizza(int fatGrams)
    {
        super(fatGrams, 8);
    }
}
```

```java
public class FoodAnalyzer
{
   public static void main (String[] args)
   {
      Pizza special = new Pizza (275);

      System.out.println ("Calories per serving: " +
                          special.caloriesPerServing());
   }
}
```

Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainabilty, and reuse of the software
- Let's summarize some of the issues regarding

Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Override methods as appropriate to tailor or change the functionality of a child

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide

Restricting Inheritance

- The `final` modifier can be used to curtail inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all
  - Thus, an abstract class cannot be declared as final
- These are key design decisions, establishing that a method or class should be used as is

Defining a Package

- A package is both a naming and a visibility control mechanism:

  1) divides the name space into disjoint subsets It is possible to define classes within a package that are not accessible by code outside the package.

  2) controls the visibility of classes and their members It is possible to define class members that are only exposed to other members of the same package.

- Same-package classes may have an intimate knowledge of each other, but not expose that knowledge to other packages

Creating a Package

- A package statement inserted as the first line of the source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- means that all classes in this file belong to the myPackage package.
- The package statement creates a name space where such classes are stored.
- When the package statement is omitted, class names are put into the default package which has no name.

Multiple Source Files

- Other files may include the same package instruction:

1.
```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

2.
```
package myPackage;
class MyClass3{ … }
```

- A package may be distributed through several source files

Packages and Directories

- Java uses file system directories to store packages.
- Consider the Java source file:

```
package myPackage;
class MyClass1 { … }
class MyClass2 { … }
```

- The byte code files MyClass1.class and MyClass2.class must be stored in a directory myPackage.
- Case is significant! Directory names must match package names exactly.

Package Hierarchy

- To create a package hierarchy, separate each package name with a dot:

  package myPackage1.myPackage2.myPackage3;

- A package hierarchy must be stored accordingly in the file system:

  1) Unix myPackage1/myPackage2/myPackage3
  2) Windows myPackage1\myPackage2\myPackage3
  3) Macintosh myPackage1:myPackage2:myPackage3

- You cannot rename a package without renaming its directory!

Accessing a Package

- As packages are stored in directories, how does the Java run-time system know where to look for packages?
- Two ways:

  1) The current directory is the default start point - if packages are stored in the current directory or sub-directories, they will be found.

  2) Specify a directory path or paths by setting the CLASSPATH environment variable.

CLASSPATH Variable

- **CLASSPATH - environment variable that points to the root directory of the system's package hierarchy.**
- **Several root directories may be specified in CLASSPATH,**
- **e.g. the current directory and the C:\raju\myJava directory:**

  .;C:\raju\myJava
- **Java will search for the required packages by looking up subsequent directories described in the CLASSPATH variable.**

Finding Packages

- Consider this package statement:

  package myPackage;

- In order for a program to find myPackage, one of the following must be true:

  1) program is executed from the directory immediately above myPackage (the parent of myPackage directory)

  2) CLASSPATH must be set to include the path to myPackage

Example: Package

```
package MyPack;

class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n; bal = b;
}
void show() {
if (bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}  }
```

Example: Package

```
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for (int i=0; i<3; i++) current[i].show();
}
}
```

Example: Package

- Save, compile and execute:
  1)call the file AccountBalance.java
  2) save the file in the directory MyPack
  3)compile; AccountBalance.class should be also in MyPack
  4) set access to MyPack in CLASSPATH variable, or make the parent of MyPack your current directory
  5) run: java MyPack.AccountBalance
- Make sure to use the package-qualified class name.

Importing of Packages

- Since classes within packages must be fully-qualified with their package names, it would be tedious to always type long dot-separated names.
- The import statement allows to use classes or whole packages directly.
- Importing of a concrete class:

  import myPackage1.myPackage2.myClass;
- Importing of all classes within a package:

  import myPackage1.myPackage2.*;

Import Statement

- **The import statement occurs immediately after the package statement and before the class statement:**

  package myPackage;

- **import otherPackage1;otherPackage2.otherClass;**

  class myClass { … }

- **The Java system accepts this import statement by default:**

  import java.lang.*;

- **This package includes the basic language functions. Without such functions, Java is of no much use.**

Example: Packages 1

- A package MyPack with one public class Balance.
  The class has two same-package variables: public constructor and a public show method.

```
package MyPack;
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n; bal = b;
}
public void show() {
if (bal<0) System.out.print("-->> ");
System.out.println(name + ": $" + bal);
}}
```

Example: Packages 2

The importing code has access to the public class Balance of the MyPack package and its two public members:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show();
}
}
```

Java Source File

- Finally, a Java source file consists of:

1) a single package instruction (optional)
2) several import statements (optional)
3) a single public class declaration (required)
4) several classes private to the package (optional)

- At the minimum, a file contains a single public class declaration.

Differences between classes and interfaces

- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- One class can implement any number of interfaces.
- Interfaces are designed to support dynamic method resolution at run time.

- Interface is little bit like a class... but interface is lack in instance variables....that's u can't create object for it.....
- Interfaces are developed to support multiple inheritance...
- The methods present in interfaces r pure abstract..
- The access specifiers public,private,protected are possible with classes, but the interface uses only one spcifier public.....
- interfaces contains only the method declarations.... no definitions.......
- A interface defines, which method a class has to implement. This is way - if you want to call a method defined by an interface - you don't need to know the exact class type of an object, you only need to know that it implements a specific interface.
- Another important point about interfaces is that a class can implement multiple interfaces.

Defining an interface

- Using interface, we specify what a class must do, but not how it does this.
- An interface is syntactically similar to a class, but it lacks instance variables and its methods are declared without any body.
- An interface is defined with an interface keyword.