Contents

Recap of the previous Lecture (if needed)

1 Theory

Date 01: May 27, 2014
Date 08: May 27, 2015

1.1 History

Initially developed by Sun Micro Systems. Now owned by Oracle Corp.

1.2 Some important Aspects of Java

- **Simple:**
  - Looks familiar to existing programmers: related to C and C++
  - Poorly defined features such as operator overloading, multiple inheritance, automatic coercions, etc are removed from java.
  - Has no header files and eliminated C preprocessor
  - No Pointers
  - Automatic Garbage Collection.
  - A rich predefined class library, almost all standard data structures are define in java.
- **Fully Object-Oriented:**
- **Distributed:**Java supports various levels of network connectivity through classes in the java.net package. Network Programming in java is easy.
- **Interpreted & Architecture-Neutral:**
- **Multithreaded:**
- **Used for a wide range of applications:** Results in a number of versions:

- Java Platform, Enterprise Edition (Java EE). is the industry standard for enterprise Java computing.
- Java Platform, Standard Edition (Java SE).

- Java Embedded. For resource-constrained devices and desktop-class systems.
- Java for Mobile Devices (Micro Edition (Java ME))

Accordingly to a survey (JavaOne 2010 conference): Java is now the most widely used software development language in the world.

2 The First HelloWorld Program

```
// Output will be a message: name the program as Welcome1.java
// Text-printing program.
public class Welcome1
{
// main method begins execution of Java application
   public static void main( String[] args )
   {
       System.out.println( "Java is fun" );
   } // end of main method
} // end of Welcome1 class
```

Some important points:

Class Name standards: A public class X must be placed in a file that has the same name as X.java. *this is not the case with C or C++*

By convention, class names *begin with a capital letter and capitalize the first letter of each word* they include (e.g., SampleClassName). A class name is an ***identifier***: a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that *does not begin with a digit* and *does not contain spaces.*

Some valid identifiers are Welcome1, $value, _value, m_inputField1 and button7. The name 7button is not a valid identifier because it begins with a digit, and the name input field is not a valid identifier because it contains a space.

Method and Main Method: Looks like:
`public static void main( String[] args )`.
This is the ***starting point*** of every ***Java Application.***

Keyword **void** indicates that this method will not return any information. Later, well see how a method can return information.

Compiling and Executing: To compile the program, type:
`javac Welcome1.java`
It will generate Welcome1.class file as the byte-code.

Next run it by:

```
java Welcome1
```

A minor modification of println:

```
 // Fig. 2.3: Welcome2.java
// Printing a line of text with multiple statements.

        public class Welcome2
        {
// main method begins execution of Java application
                public static void main( String[] args )
                {

                                        System.out.print( "Welcome to "
                                            );
                                        System.out.println( "Java
                                           Programming!" );
                } // end method main
        } // end class Welcome2
```

Displaying Multiple Lines of Text with a Single Statement:
Use **newline characters** \n

```
 // Fig. 2.4: Welcome3.java
// Printing multiple lines of text with a single statement.

        public class Welcome3
        {
        // main method begins execution of Java application
        public static void main( String[] args )
                {
        System.out.println( "Welcome\nto\n Java Programming!" )
            ;
                } // end method main
        } // end class Welcome3
```

Using printf: Formatted text similar to C language.

Example:

```
System.out.printf( "%s\n%s\n",
 "Welcome to", "Java Programming!" );
```

The **static** modifier of main method. A static method is special, because you can call it **without first creating an object of the class in which the method is declared**.

Enhance Readability by Comments of the program: 3 ways to do it:

1. single line: Use `//`
2. multiline: Use `/* text here */`
3. javadoc utility:Javadoc is a tool for generating API documentation in HTML format from doc comments in source code.

Example Simple Comments:

```
/**
* Some description will here about the class after this tag.
* only writes which targets are being executed, and
* any messages that get logged.
*
*/
public class myclass{



}
```

3 javadoc

What is Javadoc?

Traditionally, when you write a program you must write a separate document recording how the program works. When the program changes, the separate documentation must change as well. *For large programs that are constantly being changed and improved, it is a tedious and error-prone task to keep the documentation consistent with the actual code.*

Javadoc is a tool that helps with this problem. Instead of writing and maintaining separate documentation, the programmer writes specially-formatted comments in the Java code itself. Javadoc takes these comments and transforms them into documentation in HTML (web page) format.

Few Points:

- You should have a Javadoc comment *immediately before each class, method and top-level variable* i.e. that means a global or instance variable, *not* a local variable within a method).

- Your Javadoc comment at the beginning of the *main* class file should *describe the entire program.*

General format of Javadoc Comments

All Javadoc comments have the following format:

```
/**
 * Summary sentence.
 * More general information about the
 * program, class, method or variable which
 * follows the comment, using as many lines
 * as necessary.
 *
 * zero or more tags to specify more specific kinds
 * of information, such as parameters and return
 * values for a method
 */
```

A Few Points:

- You start out with the normal beginning of comment delimiter `(/*)` followed by another `(*)`.
- All following lines start with an asterisk lined up under the first asterisk in the first line.
- The last line contains just the normal end of comment delimiter `(*/)`.
- Type `javadoc [-options] sources.java` . The IDE will generate a separate **doc** folder.

  In the default settings *only public class and public method* will be rendered for doc generation. If you want to include private/protected/default class or method you must call it by `javadoc -private souces.java` In IDE you can select the same settings in running javadoc.

Without tag

Without tag you can create a document just like an html document. Here is an example of a Javadoc comment without tags which describes the variable declarated immediately below it:

```
 /**
* <h1>Add Two Numbers!</h1>
* The AddNum program implements an application that
* simply adds two given integer numbers and Prints
* the output on the screen.
* <p>
*
<b>Note:</b> Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
* @author  Abu Raihan M Kamal
* @version 1.0
* @since   2016-06-15
* @see www.google.ie
*/
```

Note: You still can and should have comments within methods, describing local variables and the computing going on inside the methods. There is, however, no Javadoc format for these comments. Use // or /*..*/, whichever you prefer.

3.1 Tags

The general form of a tag is:
`* @name comment`
Each tag should start on a new line.

Tags for Classes

- `@author` : In a Javadoc comment before a class definition, you should have an author tag, specifying who wrote the program.
- `@version` : Adds a "Version" heading with the specified version-text to the generated docs when the -version option is used.
- `@since` : Adds a "Since" heading with the specified since-text to the generated documentation.
- `@see` : Adds a "See Also" heading with a link or text entry that points to reference.

Tag for only Methods

- `@param` : ¡exact name of parameter¿ short description of parameter.

- `@return` : short description of return value
- `@throws` : The @throws and @exception tags are synonyms.

A complete example:

```
 import java.io.*;

/**
* <h1>Add Two Numbers!</h1>
* The AddNum program implements an application that
* simply adds two given integer numbers and Prints
* the output on the screen.
* <p>
* <b>Note:</b> Giving proper comments in your program makes it more
* user friendly and it is assumed as a high quality code.
*
* @author  Your Name
* @version 1.1
* @since   2016-06-18
*/
public class AddNum {
   /**
   * This method is used to add two integers. This is
   * a the simplest form of a class method, just to
   * show the usage of various javadoc Tags.
   * @param numA This is the first paramter to addNum method
   * @param numB  This is the second parameter to addNum method
   * @return int This returns sum of numA and numB.
   */
   public int addNum(int numA, int numB) {
      return numA + numB;
   }

   /**
   * This is the main method which makes use of addNum method.
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
   */
```

```
    public static void main(String args[]) throws IOException
    {

        AddNum obj = new AddNum();
        int sum = obj.addNum(10, 20);

        System.out.println("Sum of 10 and 20 is :" + sum);
    }
}
```

3.1.0.1 Hand On Example:

See AddNum.java program in /home/raihan/workspace/JavaDoc

4 Taking Input (similar to scanf)

Our next application reads (or inputs) two **integers** (whole numbers, such as 22, 7, 0 and 1024) typed by a user at the keyboard, computes their sum and displays it. This program must keep track of the numbers supplied by the user for the calculation later in the program. Programs remember numbers and other data in the computers memory and access that data through program elements called **variables**.

Start of the sample source code: Taking Input

```
 import java.util.Scanner;
public class Addition{

public static void main( String[] args ){

Scanner input = new Scanner( System.in );
int number1; // first number to add
int number2; // second number to add
int sum; // sum of number1 and number2

System.out.print( "Enter first integer: " ); // prompt
number1 = input.nextInt(); // read first number from user

System.out.print( "Enter second integer: " ); // prompt
number2 = input.nextInt(); // read second number from user

sum = number1 + number2; // add numbers, then store total in
    sum

System.out.printf( "Sum is %d\n", sum ); // display sum
```

```
}// end method main

}// end class Addition
```

End of the sample source code: Taking Input

Few Points about the above source:

- A great strength of Java is its **rich set of predefined classes** that you can **reuse** rather than *reinventing the wheel.* These classes are grouped into packagesnamed groups of related classesand are collectively referred to as the **Java class library**, or the **Java Application Programming Interface (Java API)**.

  import declaration that helps the compiler locate a class thats used in this program. It indicates that this example uses **Javas predefined Scanner class** (discussed shortly) from **package java.util.**

  All import declarations must appear before the first class declaration in the file. Placing an import declaration inside or after a class declaration is a syntax error. **Notes on Scanner class:**

  Scanner input = new Scanner(System.in);

  is **a variable declaration** statement that specifies the name (input) and type (Scanner) of a variable thats used in this program. A Scanner enables a program to read data (e.g., numbers and strings) for use in a program. **The data can come from many sources, such as the user at the keyboard or a file on disk.** Before using a Scanner, you **must create** it and **specify the source** of the data.

  The = in line 11 indicates that Scanner variable input should be initialized (i.e., prepared for use in the program) in its declaration with the result of the expression to the right of the equals signnew Scanner(System.in). This expression uses the **new** keyword to create a Scanner object that reads characters typed by the user at the keyboard. The standard input object, System.in, enables applications **to read bytes of information typed by the user**. The Scanner translates these bytes into types (like ints) that can be used in a program.

Start of the sample source code

```
public static void main(String[] args) {
  System.out.println("Hello, World");
```

```
System.out.printf( "%s\n%s\n", "Welcome to", "Java Programming
    !" );

        A myA=new A();

        myA.showvalue();
    }
```

End of the sample source code

Recap of the previous Lecture (if needed)

5 Theory

Date: May –, 2014

6 Quick References

6.1 Operators

See **Fig. 2.11** in the text book. Page 53.

6.2 Equality and Relational Operators

See **Fig. 2.14**. Page 57.

6.3 2.9 Wrap-Up

Page 60–

7 Chapter 3

7.1 Declaring a Class w a Meth, and Instantiating an Object of a Cl.

Run first **GradeBook.java** then **GradeBookTest.java** (contains main meth.)

Each class declaration that begins with keyword public must be stored in a file having the same name as the class and ending with the .java file-name extension. Thus, classes GradeBook and GradeBookTest must be declared in separate files, because each class is declared public.

The keyword public is an **access modifier**.

public: available to the publicit can be called from methods of other classes

7.2 Declaring a Method with a Parameter

: Like C. See page **77**.

7.3 set Methods and get Method

. See page **80**.

7.4 Access Modifiers public and private

See page **80**.

Variables or methods declared with access modifier *private* are accessible *only to methods of the class in which theyre declared.*

Summary of access modifiers:

Access Modifier	Withing class	withing package	outside pkg,by subclass only	outside pkg
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

7.5 Constructors

Each time you create an object, a constructor of that class gets called. You can make use of the constructor to initialize the newly created object by setting the initial state of the object, and you can acquire some resources (such as file handles). The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Every class has a constructor. If you do not explicitly write a constructor for a class, the Java compiler provides a default constructor (without any parameter) for that class.

Normally without any explicit Constructors all variables are initialized to *null or 0 or false.*

We have seen **default constructor with no parameters**.

- A constructor must have the **same name as the class** and will have **no return type**.

Constructor without parameters (Default)

```
class Bike1{

    Bike1()
    { System.out.println("Bike is created");
    }

  public static void main(String args[]){

  Bike1 b=new Bike1();
}
```

```
}
```

Constructor with parameters

```
  class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```

Example of Constructor Overloading

```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Method with identical name

```
public void Student4() {
```

```
    id = 101;
    age = 25;

}
```

Note:A constructor does not have a return type, and here youve given a void return type. *This is not a constructor, but it is a method* named Student4 in this class! Beware: Java allows you to declare methods with same name as the class name, but it is not a good practice to make use of this feature (since it will confuse the programmers reading your code).

See program 3.10 on page **85,86**.

7.6 GUI and Graphics (will be covered later)

Run **Dialog1.java**.

8 Chapter 4

8.1 ifelse statement

Dangling-else Problem.

```
if ( x > 5 )
if ( y > 5 )
System.out.println( "x and y are > 5" );
else
System.out.println( "x is <= 5" );
```

Use Block:

```
 if ( x > 5 )
{
if ( y > 5 )
System.out.println( "x and y are > 5" );
}
else
System.out.println( "x is <= 5" );
```

8.2 while Repetition Statement

```
 while ( product <= 100 )
product = 3 * product;
```

8.3 Compound Assignment Operators

c = c + 3;
can be reduced to

c += 3;

8.4 Increment ++ and decrement – operators

```
c = 5;

System.out.println( c++ );// prints 5 then postincrements
System.out.println( c ); // prints 6


c = 5;

System.out.println( ++c ); // preincrements then prints 6
System.out.println( c ); // prints 6
```

System.out.println(c++);

9 Chapter 5

Control statement. See page 165. How to take inputs until some special symbol or combination is pressed.

Swith case:

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

while(input.hasNext()){

grade = input.nextInt();

}
```

Lines 5054 prompt the user to enter integer grades and to type the end-of-file indicator to terminate the input. **The end-of-file indicator is a system-dependent keystroke combination** which the user enters to indicate that theres no more data to input.

On UNIX/Linux/Mac OS X systems, end-of-file is entered by typing the sequence:

¡Ctrl¿ d

On Win: ¡Ctrl¿ z.

9.1 Logical Operators

- **Conditional AND (&&) Operator**

```
if ( gender == FEMALE && age >= 65 ){


}
```

- Conditional OR (||) Operator

```
 if ( ( semesterAverage >= 90 ) || ( finalExam >= 90 ) )
System.out.println ( "Student grade is A" );
```

- Logical Negation (!) Operator

```
 if ( ! ( grade == sentinelValue ) )
System.out.printf( "The next grade is %d\n", grade );
```

See page 177: prog for truth table.

10 Chapter 6: Methods in Details.

10.1 Why Methods?

Also called function, procedure, sub-program.

- Easy to maintain and manage large program. Called **divide and conquer** approach.
- Software **re-usability**. Already you have used a method already developed: taking input from keyboard. To achieve this, *the name should express the task it is doing.*

10.2 Static Methods, fields, Final Keyword

Already has been explained. The example is the use of **main** method.

Consider the following code:

```
System.out.println( Math.sqrt( 900.0 ) );
```

There was **no need to create a Math object** before calling method sqrt.

The **final** keyword is to make a value constant. Math Class Constants PI and E. They are final values. Final: constantits value cannot change after the field is initialized. PI and E are declared final because their values never change.

+ concatenation. Every primitive value and object in Java has a String representation. If there are **any trailing zeros** in a double value, these will be **discarded** when the number is converted to a Stringfor example 9.3500 would be represented as 9.35.

Notes on static method:

A static method can call only other static methods of the same class directly (i.e., using the method name by itself) and can **manipulate only static variables** in the same class directly. To access the classs non-static members, a static method must use a **reference to an object of the class**. *Java does not allow a static method to access non-static members of the same class directly.*

Chapter 8 will open up this topic more.

Static field: only one copy is created and shared by all instances of objects. Chapter 8 will explain more.

Use of static filed

1. **Normal Use:** Access through class reference.

```
 public class Stuff {
   public static String name = "I'm a static variable";
}


public class Application {

   public static void main(String[] args) {
       System.out.println(Stuff.name);
    }

}

OUTPUT:
I'm a static variable
```

2. **Static Keyword to Create Constants:**
   One common use of static is to create a constant value that's attached to a class.

   The benefit of using both static and final lies with the storage and performance issue. Only one copy is created and its access is very fast.

```
 public class emp {
   public final static int MAX_AGE = 65;
}


public class Application {

   public static void main(String[] args) {
       System.out.println(emp.MAX_AGE);
    }

}
```

3. **Static Variable to Count Objects (global variable)**

```
 // Stuff.java
 public class Stuff {

    static int count = 0;
    public Stuff() {
        count++;
       System.out.println("Stuff object constructed!
       Value of count is:"+count);
   }


  public static int get_salary(){
  return 10000;
}
  public static void show(){
  int x,y;
  x=get_salary();
// y=get_bonus();
System.out.println("From sow method. Call of get_salary
    return:"+x);

}
  public int get_bonus()
{ return 100;
}

}


// Application.java (where main method resides)

public class Application {

   public static void main(String[] args) {
```

```
        final int x=20;
        x=20;
        Stuff.show();
        Stuff stuff1 = new Stuff();
         Stuff stuff2 = new Stuff();
         Stuff stuff3 = new Stuff();
    }

}
```

10.3 Activation Record, Method Call Stack

Uses LIFO stack. Push (add), pop (remove).

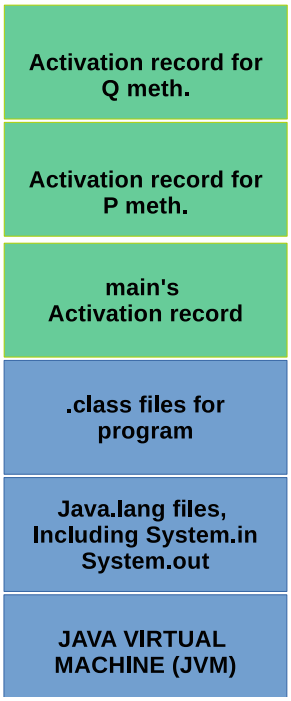

Figure 1: java Activation Record Stac

10.4 Argument Promotion and Casting

promotionconverting an arguments value, if possible.
The statement

```
System.out.println( Math.sqrt( 4 ) );
```

correctly evaluates Math.sqrt(4) and prints the value 2.0. The ruling is that: **no loss of data precision is achieved.**

Legal are:

from float to double
from int to long or double
from byte to int

Not legal:
Double to int.

Casting: Forcing complier to convert explicitly. y=(int) x; (if x is double)

10.5 Package: Basic

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer.

Formal Definition: A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types. Enumerations and annotation types are special kinds of classes and interfaces, respectively, so types are often referred to in this lesson simply as classes and interfaces.

Why Package?

- You and other programmers can easily determine that these types are related.
- The names of your types won't conflict with the type names in other packages because the package creates a new namespace.
- You can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

10.6 Access Modifiers

Access modifiers determine the level of visibility (and therefore access) for a Java entity (a class, method, or field).

Java supports four types of access modifiers:

1. Public
2. Private

3. Protected

4. Default (no access modifier specified)

10.6.1 Public Access Modifier

The public access modifier is the most liberal one. If a class or its members are declared as public, they can be accessed from any other class regardless of the package boundary.

In the FunPaint example, lets assume that the class Circle has a public method area(). This method can be accessed from anywhere, even from another package, as shown in the Listing 3-2 snippet.

Listing 3-2. Circle.java

```
// Shape.java
package graphicshape;
class Shape {
// class definition elided
}
// Circle.java
package graphicshape;
public class Circle extends Shape {
public void area() { //public method
// code for area method elided
}
}
// Circles.java
package graphicshape;
class Circles {
void getArea() {
Circle circle = new Circle();
circle.area(); //call to public method area(), within package
}
}
```

Now from another package:

```
// Canvas.java
package appcanvas;
import graphicshape.Circle;
class Canvas {
void getArea() {
```

```
Circle circle = new Circle();
circle.area(); //call to public method area(), outside package
}
}
```

NOTE: *A public method in a class is accessible to the outside world only if the class is declared as public. If the class does not specify any access modifier, then the public method is accessible only within the containing package.*

10.7 Private Access Modifier

The private access modifier is the most stringent access modifier. A private class member cannot be accessed from outside the class; only members of the same class can access these private members.

Lets add the attribute radius to the class Circle as a private member. In this case, the attribute can be assessed byand only bythe members of the Circle class, as shown in Listing 3-3.

```
 // Shape.java
package graphicshape;
class Shape {
//class definition
}
//Circle.java
package graphicshape;
public class Circle extends Shape {
private int radius; //private field
public void area() { //public method
// access to private field radius inside the class
System.out.println("area:"+3.14*radius*radius);
}
}
// Circles.java
package graphicshape;
class Circles {
void getArea() {
Circle circle = new Circle();
circle.area(); //call to public method area(), within package
}
}
```

```
// Canvas.java
package appcanvas;
import graphicshape.Circle;

class Canvas {
void getArea() {
Circle circle = new Circle();
circle.area(); // call to public method area(), outside package
}
}
```

In this example, radius is accessible only inside the Circle class and not in any other class, regardless of the enclosing package.

10.8 Protected and Default Access Modifier

Protected and default access modifiers are quite similar to each other. If a member method or field is declared as protected or default, then the method or field can be accessed within the package.

What is the difference between protected and default access? One significant difference between these two access modifiers arises when we talk about a subclass belonging to another package than its superclass. In this case, protected members are accessible in the subclass, whereas default members are not.

Summary Table: p-55

10.9 Java API Package

A great strength of Java is the Java APIs thousands of classes.

Package	Main use
java.applet	to create Java appletsprograms that execute in web browsers.
java.awt	Java Abstract Window Toolkit Package (not used now)
java.awt.event	enable event handling for GUI components
java.io	Java Input/Output Package
java.net	communicate via computer networks
java.sql	The JDBC Package DBConnection.
java.util	date and time manipulations, random-number and so on.
javax.swing	Javas Swing GUI components: Ch 14 and 25
projavax.swing.event	enable event handling (eg. Button click)

10.10 Case Study of Random Number Generation

See it. You can find the probability of each dice face. Do it 10000 times, count the frequency and compute the probability. Do the same test 10 times and see if it truly random.

10.11 For Loop for Enum:

The for statement also has another form **designed for iteration through Collections and arrays** This form is sometimes referred to as the enhanced for statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
class EnhancedForDemo {
   public static void main(String[] args){
        int[] numbers =
            {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
            System.out.println("Count is: " + item);
        }
   }
}
```

Output:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

10.12 Variable Scope

- **Class Level Scope:** In Java, there are some variables that **you want to be able to access from anywhere within a Java class**. The scope of these variables will need to be **at the class level**, and there is only one way to create variables at that level just **inside the class but outside of any methods**.

  Example:

```
public class myclass{

private String username;

}
```

- **Method Scope:** Some variables you might want to make temporary and preferably they are used for only one method. This would be an example of method scope. Here's a pretty typical example of a variable in method scope using an example of Java's main method:

```
public static void main(String[] args){

int x=10;

}
```

Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

```
public void setName(String name){

username=name;

}
```

The above example is the typical example of a **setter method**. The **purpose of a setter method** as you might recall is to set a class variable to a particular value **from somewhere outside of the class**.

```
public void setName(String username){

this.username=username;

}
```

*The keyword **this** tells Java that you are referencing the class variable.*

- **Loop Scope:** Any variables created inside of a loop are LOCAL TO THE LOOP. This means that once you exit the loop, the variable can no longer be accessed!

```
for(int i=0;i<10;i++){


}

print(i);
```

- **Shadowing:** Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field of the class, the field is hidden until the block terminates executionthis is called shadowing. **See Figure 6.9**.

10.13 Method Overloading

Methods of the same name can be declared in the same class, as long as they have different sets of parameters (**determined by the number, types and order of the parameters**)this is called method overloading.

The compiler distinguishes overloaded methods by their signaturea combination of the methods name and the number, types and order of its parameters.

Argument lists (Signature) could differ in

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

Method overloading is also known as *Static Polymorphism*.
Static Polymorphism is also known as *compile time binding* or *early binding*.
Some Examples:

```
 // Case 1: Overloading Different Number of parameters
 class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}

class Sample
{
   public static void main(String args[])
   {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
   }
}
```

```
// Case 2: different types of parameters

class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

// Case 3: Sequence of data type of arguments
class DisplayOverloading3
{
   public void disp(char c, int num)
   {
       System.out.println(" I  m  the first definition of method
           disp");
   }
   public void disp(int num, char c)
   {
       System.out.println(" I  m  the second definition of
          method disp" );
   }
}
```

Some invalid Examples:

```
 // Case 1:
int mymethod(int a, int b, float c)

int mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types in arguments.

```
 // Case 2:
int mymethod(int a, int b)

float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since *return type* of method doesnt matter while overloading a method.

11 Chapter 7: Arrays and ArrayLists

All functions of array are obtained from java.util package.

11.1 Declaring and Creating Arrays

Array objects occupy space in memory. Like other objects, arrays are created with keyword **new**.

```
int[] c; // no memory is allocated yet
c = new int[ 12 ];
```

is similar to :

```
int[] c = new int[ 12 ];
```

Its is better to declare each array in separate line:

```
 String b[] = new String[ 100 ]; // create array b
String x[] = new String[ 27 ]; // create array x
```

could be also written as:

```
String[] b = new String[ 100 ], x = new String[ 27 ];
```

Show all elements of an array: use length attribute.

```
int[] array; // declare array named array
array = new int[ 10 ]; // create the array object

for ( int counter = 0; counter < array.length; counter++ )
System.out.printf( "%d  :  %d\n", counter, array[ counter ] );
```

Array Init: No need to use new.

```
int[] array = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
```

11.2 Enhanced for Statement

The enhanced for statement iterates through the elements of an array without using a counter, thus avoiding the possibility of stepping outside the array.

General Syntax is:

```
for ( parameter : arrayName )
statement
```

Lets convert a traditional code into it:

```
 for ( int counter = 0; counter < array.length; counter++ )
total += array[ counter ];
```

It can be done using the Enhanced for Statement:

```
int[] array = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
int total = 0;

for ( int number : array )
total += number;
System.out.printf( "Total of array elements: %d\n", total );
```

11.3 Passing Arrays to Methods

Example:

```
double[] results = new double[ 50 ];


getSummaryResult(results);
```

Every array object knows its own length (via its length field).

```
void getSummaryResult( double[] b ){


}
```

For passing individual element use the index.

11.4 Variable length Argument List

With variable-length argument lists, you can create methods that receive an *unspecified number of arguments.*

- A type followed by an *ellipsis (...)* in a methods parameter list indicates that the method receives a variable number of arguments of that particular type.
- This use of the ellipsis can occur only once in a parameter list, and the ellipsis, together with its type, must be placed at the end of the parameter list.
- Java treats the variable-length argument list as an array whose elements are all of the same type.

Example:

```
 public class varg_test
 {
 // calculate average
 public static double average(double... numbers)
 {
 double total = 0.0; // initialize total

// calculate total using the enhanced for statement
for ( double d : numbers )
total += d;

return total/numbers.length ;
 } // end method average

public static void main( String[] args ){

double x1=10.6;
double x2=20.6;
double x3=40.6;

System.out.println("With 1 argument: " +average(x1));
System.out.println("With 2 argument: " +average(x1,x2));
System.out.println("With 3 argument: " +average(x1,x2,x3));


 } // end of main
} // end of class
```

11.5 Command-Line Arguments

See page number 280. Look specially at the following line:

```
 int arrayLength = Integer.parseInt( args[ 0 ] );
 int[] array = new int[ arrayLength ]; // create array
```

The static method *parseInt* of class Integer converts its String argument to an int.

11.6 Class Arrays

Class *Arrays* helps you avoid reinventing the wheel by providing *static methods* for common array manipulations.

These methods include:

- **sort** for sorting an array (i.e., arranging elements into increasing order)

- **binarySearch** for searching an array (i.e., determining whether an array contains a specific value and, if so, where the value is located)
- **equals** for comparing arrays
- **fill** for placing values into an array.

Systems static arraycopy method: arraycopy(Object src, int srcPos, Object dest, int destPos, int length)

See page 282.

11.7 toString

Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

It is defined in Object class. This method can be overridden to customize the String representation of the Object.

The toString method for class Object returns a string consisting of the *name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object.*

12 Chapter 8: Deeper look at Class

12.1 get and set method

Set methods are also commonly called *mutator methods*, because they typically change an objects statei.e., modify the values of instance variables. Get methods are also commonly called *accessor methods or query methods.*

12.2 Composition

A class can have references to objects of other classes as members.

Suppose we have a class:

```
public class Date{


}
```

We have another class:

```
public class Employee{

private String firstName;
private String lastName;
private Date birthDate;
private Date hireDate;


public Employee( String first, String last, Date dateOfBirth,
 Date dateOfHire )
 {
 firstName = first;
 lastName = last;
 birthDate = dateOfBirth;
 hireDate = dateOfHire;
 } // end Employee constructor


}
```

Now see how to call from another class (with main method).

```
public class EmployeeTest{

public static void main( String[] args ){

Date birth = new Date( 7, 24, 1949 );
Date hire = new Date( 3, 12, 1988 );

Employee employee = new Employee( "Bob", "Blue", birth, hire );



}

}
```

12.3 Garbage Collection and Method finalize

Every class in Java has the methods of class *Object* (package java.lang), one of which is the *finalize* method.

Every object uses system resources, such as memory. We need a disciplined way to give resources back to the system when theyre no longer needed; otherwise, *resource leaks* might occur that would prevent them from being reused by your program or possibly by other programs. *The JVM performs automatic garbage collection to reclaim the memory occupied by objects that are no longer used.* When

there are no more references to an object, the object is eligible to be collected. This typically occurs when the JVM executes its garbage collector. So, memory leaks that are common in other languages like C and C++ (because memory is not automatically reclaimed in those languages) are less likely in Java, but some can still happen in subtle ways. Other types of resource leaks can occur. For example, an application may open a file on disk to modify its contents. If it does not close the file, the application must terminate before any other application can use it.

The *finalize* method is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the objects memory. Method finalize does not take parameters and has return type void. A problem with method finalize is that the garbage collector is not guaranteed to execute at a specified time. In fact, the garbage collector may never execute before a program terminates. Thus, its unclear whether, or when, method finalize will be called. For this reason, most programmers should avoid method finalize.

static Class Members: Every object has its own copy of all the instance variables of the class. In certain cases, *only one copy of a particular variable should be shared by all objects* of a class. A static fieldcalled a class variableis used in such cases. *A static variable represents classwide information*all objects of the class share the same piece of data. The declaration of a static variable begins with the keyword static.

static import: import static java.lang.Math.*;

package : done.

13 Chapter 9 : Inheritance

13.1 introduction: (What and Why)

This chapter continues our discussion of *object-oriented programming (OOP)* by introducing one of its primary capabilities *inheritance*, which is a *form of software reuse* in which a new class is created by absorbing an existing classs members and embellishing them with new or modified capabilities.

When creating a class, rather than declaring completely new members, you can designate that the new class should inherit the members of an existing class. The existing class is called the *superclass*, and the new class is the *subclass*.

The *direct superclass* is the superclass from which the subclass explicitly inherits. An *indirect superclass* is any class above the direct superclass in the class hierarchy, which defines the inheritance relationships between classes. (i.e. Object

class).

Java supports only single inheritance, in which each class is derived from exactly one direct superclass. Unlike C++, Java does not support multiple inheritance (which occurs when a class is derived from more than one direct superclass). Chapter 10, Object- Oriented Programming: Polymorphism, explains how to use *Java interfaces* to realize many of the benefits of multiple inheritance while avoiding the associated problems.

New classes can inherit from classes in class libraries. Organizations develop their own class libraries and can take advantage of others available worldwide. Some day, most new software likely will be constructed from *standardized reusable components*, just as automobiles and most computer hardware are constructed today. This will facilitate the development of more powerful, abundant and economical software.

13.1.1 Superclasses and Subclasses

Superclasses tend to be *more general* while subclasses *more specific.* For example:

Superclass	Subclass
Students	Graduate Student, Ph.D student
Employee	Banker, Teacher, Doctor
Loan	Car Loan, House Loan

13.1.2 protected Members

Access Modifiers Revisited:

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

We have seen access modifiers *public*: accessible from anywhere.
and *private*: accessible from only within the class.

Using *protected* access offers *an intermediate level* of access between public and private.

Ruling is: *A superclasss protected members can be accessed by members of that superclass, by members of its subclasses and by members of other classes in the same package ; protected members also have package access.*

13.1.3 How to create a subclass: example

Use *extends* keywords.

Consider the following code:

```
public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int
        startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }

}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
```

```
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}
```

13.1.4 Few Points

Constructor: Normally constructors are not inherited. Use *super* keyword to use the constructor of the superclass. If no constructor is specified in the subclass then java calls the superclass's default or no-argument constructor.

Override Annotation: Follow the following example:

```
Override // indicates that this method overrides a superclass methodpublic
    double earnings()// do something.
```

It indicate a few things:

- The method with @override indicates that it is overriding the method in its superclass. Compiler knows that there is a similar method in its superclass.
- In overriding a superclass method *common errors* are mismatch of name, number of Arguments, type and order of arguments. When the compiler encounters a method declared with @Override, it compares the methods signature with the superclasss method signatures. If there isnt an exact match, the *compiler issues an error message*, such as *method does not override or implement a method from a supertype.* This indicates that youve accidentally overloaded a superclass method. You can then fix your methods signature so that it matches one in the superclass.

Notes on using *protected* instance variable:

2 Options:

In this example, we declared superclass instance variables as protected so that subclasses could access them. Inheriting protected instance variables *slightly increases performance*, *because* we can directly access the variables in the subclass *without incurring the overhead of a set or get method call.*

In most cases, *however*, its better to use private instance variables *to encourage proper software engineering*, and leave code optimization issues to the compiler. Your *code will be easier to maintain, modify and debug.*

Using protected instance variables creates several *potential problems.*

- First, the subclass object can set an inherited variables value directly without using a set method. A subclass may accidentally modify the value of some very critical class variable such as conversion rate, interest rate.
- With protected instance variables in the superclass, we may need to modify all the subclasses of the superclass if the superclass implementation changes. For example, if for some reason we were to change the names of instance variables firstName and lastName to first and last, then we would have to do so for all occurrences in which a subclass directly references superclass instance variables firstName and lastName. In such a case, the software is said to be fragile or brittle, because a small change in the superclass can break subclass implementation.

Class Object: It has 11 methods: all are inherited by all classes.

1. clone
2. equals
3. finalize
4. getClass
5. hashCode
6. some more...

14 Chapter 10 : Polymorphism

resource info: http://beginnersbook.com/2013/03/polymorphism-in-java/

What is polymorphism in programming?
Polymorphism is the capability of a method to do different things based on the

object that it is acting upon. *In other words, polymorphism allows you define one interface and have multiple implementations.*

A real life example: In a Banking system there are many types accounts. Interest calculations are totally different from one to another.

Few points:

- It is a feature that allows one interface to be used for a general class of actions.
- It plays an important role in allowing objects having different internal structures to share the same external interface.

Following concepts demonstrate different types of polymorphism in java.

1. Method Overloading
2. Method Overriding

1) Method Overloading: In Java, it is possible to define two or more methods of same name in a class, provided that there argument list or parameters are different. This concept is known as Method Overloading.

Example:

```
 class Overload
{
    void demo (int a)
    {
       System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
       System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
       System.out.println("double a: " + a);
       return a*a;
    }
}


class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
```

```
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

2) Method Overriding: Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class. This feature is known as method overriding.

Example:

```
 public class BaseClass
{
    public void methodToOverride() //Base class method
    {
         System.out.println ("I'm the method of BaseClass");
    }
}
public class DerivedClass extends BaseClass
{
    public void methodToOverride() //Derived Class method
    {
         System.out.println ("I'm the method of DerivedClass");
    }
}


public class TestMethod
{
     public static void main (String args []) {
        // BaseClass reference and object
        BaseClass obj1 = new BaseClass();
        // BaseClass reference but DerivedClass object
        BaseClass obj2 = new DerivedClass();
        // Calls the method from BaseClass class
        obj1.methodToOverride();
        //Calls the method from DerivedClass class
        obj2.methodToOverride();
     }
}

Output:
I'm the method of BaseClass
I'm the method of DerivedClass
```

Notes before example:

- a superclass reference at a subclass object
- We then show how invoking a method on a subclass object via a superclass

reference invokes the subclass functionality *the type of the referenced object, not the type of the variable*, determines which method is called.

- An object of a subclass can be treated as an object of its superclass
- You cannot treat a superclass object as a subclass object.
- When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.

super keyword in Overriding: When invoking a superclass version of an overridden method the super keyword is used.

Example:

```
 class Vehicle {
    public void move () {
         System.out.println ("Vehicles are used for moving from
             one place to another ");
    }
}

class Car extends Vehicle {
    public void move () {
      super. move (); // invokes the super class method
      System.out.println ("Car is a good medium of transport ")
          ;
    }
}

public class TestCar {
    public static void main (String args []){
        Vehicle b = new Car (); // Vehicle reference but Car
            object
        b.move (); //Calls the method in Car class
    }
}

OUTPUT:

Vehicles are used for moving from one place to another
Car is a good medium of transport.
```

14.1 Abstract Classes and Methods

Normally we declare class and create object of that type. For Abstract classes for which *you never intend to create objects. Because* theyre used only as superclasses

in inheritance hierarchies, we refer to them as abstract superclasses.

An abstract class is a class that is declared abstractit may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

If a class includes abstract methods, then the class itself must be declared abstract, as in:

```
 abstract class GraphicObject {
  // declare fields
  // declare nonabstract methods
  abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

```
abstract class MyGraphicsObjects extends GraphicObject{
 // declare fields

  // declare nonabstract methods

  abstract int getArea();

}
```

Purpose of Abstract Classes.
To share a common design structure.

Points to Remember:

- The `abstract` keyword can be applied to a class or a method but not to a field.
- An abstract class cannot be instantiated.
- An abstract class can extend another abstract class or can implement an interface.
- An abstract class can be derived from a concrete class! Although the language allows it, it is not a good idea to do so. (bad idea)

- An abstract class need not declare an abstract method, which means it is not necessary for an abstract class to have methods declared as abstract. However, if a class has an abstract method, it should be declared as an abstract class.
- A subclass of an abstract class needs to provide implementation of all the abstract methods; otherwise you need to declare that subclass as an abstract class.

Open *Page 405* of Book for the concrete example.

14.2 Creating and Using Interfaces.

A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some piece of information. The next example introduces an interface named Payable to describe the functionality of any object that must be capable of being paid and thus must offer a method to determine the proper payment amount due.

An interface declaration begins with the keyword interface and *contains only constants and abstract methods.* Unlike classes, *all interface members must be public, and interfaces may not specify any implementation details*, such as concrete method declarations and instance variables.

NOTE: All methods declared in an interface are *implicitly* public abstract methods, and all fields are *implicitly* public , static and final.

Why Interface?

Using an Interface: To use an interface, a concrete class must specify that it *implements* the interface and must declare each method in the interface with the signature specified in the interface declaration.

```
interface BasicChecks {

   bool isValidAccount(int no);

   bool checkLimit(int newValue);

   int getBalance(int increment);


}
```

```
class SavingTransaction implements BasicChecks {

bool isValidAccount(int no){

// own implementation goes here.

}


}
```

Interfaces vs. Abstract Classes In Abstract Classes we have some implementations. An interface is often used in place of an abstract class when theres no default implementation to inheritthat is, no fields and no default method implementations.

14.3 final Classes and Methods

Final Classes:

A final class is a non-inheritable class that is to say, if you declare a class as final, *you cannot subclass it.* In general, *OOP suggests that a class should be open for extension but closed for modification (Open/Closed Principle).* However, in some cases you dont want to allow a class to be subclassed. Two important reasons are:

1. ***To prevent a behavior change by subclassing.*** In some cases, you may think that the implementation of the class is complete and should not change. If overriding is allowed, then the behavior of methods might be changed.By making a class final, the users of the class are assured the unchanged behavior.

2. ***Improved performance.*** All method calls of a final class can be resolved at compile time itself. As there is no possibility of overriding the methods, it is not necessary to resolve the actual call at runtime for final classes, which translates to improved performance.

Consider the following:

```
final class Canvas { /* members */ }

class ExtendedCanvas extends Canvas { /* members */ }
```

If you try to extend a final class, as you just tried to do, youll get the compiler error "cannot inherit from final Canvas".

So, *Final Classes Cannot Be Superclasses.(i.e. It can not be extended.)*

Final Methods and Variables:

In a class, you may declare a method final. The final method cannot be overridden. Therefore, if you have declared a method as final in a non-final class, then you can extend the class but you cannot override the final method. However, other non-final methods in the base class can be overridden in the derived class implementation.

Consider the following:

```
 public abstract class Shape {
//class members...
final public void setParentShape(){
//method body
}

public Shape getParentShape(){
//method body
}
}
```

In this case, the Circle class (subclass of Shape) can override only getParentShape(); if you try to override the final method, you will get following error: "Cannot override the final method from Shape".

So, *Final Methods Cannot Be Overridden.*

Finally, we mention final variables. Final variables are like CD-ROMs: once you write something on them, you cannot write again. In programming, universal constants such as PI can be declared as final since you dont want anyone to modify the value of such constants. Final variables can be assigned only once. If you try to change a final variable after initialization, you will get a complaint from your Java compiler. (It has been already discussed).

Points to Remember:

- The final modifier can be applied to a class, method, or variable. All methods of a final class are implicitly final (hence non-overridable).
- A final variable can be assigned only once. If a variable declaration defines a variable as final but did not initialize it, then it is referred to as *blank final.* You need to initialize a blank final all the constructors you have defined in the class; otherwise the compiler will complain.

15 Chapter 11 : Exception Handling

15.1 Introduction:

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

Exception handling mechanism is based on the core module of C++. It helps you write robust and fault-tolerant programs that can deal with problems and continue executing or terminate gracefully.

What causes Exception? An exception can occur for many different reasons, below given are some scenarios where exception occurs:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.
- Invalid database connection parameters.

15.2 Simple Example

2 cases. with and without.

15.2.1 Without Explicit Exception Handling

Just look at the following code:

```
 // Fig. 11.1: DivideByZeroNoExceptionHandling.java
// Integer division without exception handling.
import java.util.Scanner;
public class DivideByZeroNoExceptionHandling
{
// demonstrates throwing an exception when a divide-by-zero
   occurs
public static int quotient( int numerator, int denominator )
{
return numerator / denominator; // possible division by zero
} // end method quotient
public static void main( String[] args )
{
Scanner scanner = new Scanner( System.in ); // scanner for
   input
System.out.print( "Please enter an integer numerator: " );
```

```
int numerator = scanner.nextInt();
System.out.print( "Please enter an integer denominator: " );
int denominator = scanner.nextInt();
int result = quotient( numerator, denominator );
System.out.printf(
"\nResult: %d / %d = %d\n", numerator, denominator, result );
} // end main
} // end class DivideByZeroNoExceptionHandling
```

```
Error 1:

Please enter an integer numerator: 12
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:9)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:18)
```

```
Error 2:

Please enter an integer numerator: 43
Please enter an integer denominator: hi
Exception in thread "main" java.util.InputMismatchException
at java.util.Scanner.throwFor(Scanner.java:909)
at java.util.Scanner.next(Scanner.java:1530)
at java.util.Scanner.nextInt(Scanner.java:2160)
at java.util.Scanner.nextInt(Scanner.java:2119)
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:17)
```

Explanation of the error codes: Several lines of information are displayed in response to this invalid input. This information is known as a *stack trace*, which includes the name of the exception (java.lang.ArithmeticException) in a descriptive message that indicates the problem that occurred and the method-call stack (i.e., the call chain) at the time it occurred. The stack trace includes the path of execution that led to the exception method by method. This helps you debug the program.

(For Error 1) Starting from the last line of the stack trace, we see that the exception was detected in line 18 of method main .

Each line of the stack trace contains

- the class name and method (DivideByZeroNoExceptionHandling.main)
- followed by the file name and line number (DivideByZeroNoExceptionHandling.java:18).

Look at this:

```
at DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:9
at DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:18)
```

Moving up the stack trace, we see that the exception occurs in line 9, in method quotient . *The top row of the call chain indicates the* ***throw point*** *- the initial point at which the exception occurs.* The throw point of this exception is in line 10 of method quotient .

15.2.2 With Explicit Exception Handling

Now lets see how to handle such errors in a graceful way.

This version of the application uses exception handling so that if the user makes a mistake, the program catches and handles (i.e., deals with) the exceptionin this case, *allowing the user to enter the input again.*

```
 import java.util.InputMismatchException;
import java.util.Scanner;
public class ExceptionHandle
{
// demonstrates throwing an exception when a divide-by-zero
    occurs
public static int quotient( int numerator, int denominator )
throws ArithmeticException
{
return numerator / denominator; // possible division by zero
} // end method quotient
public static void main( String[] args )
{
Scanner scanner = new Scanner( System.in ); // scanner for
    input
boolean continueLoop = true; // determines if more input is
    needed
do
{
try // read two numbers and calculate quotient
{
System.out.print( "Please enter an integer numerator: " );
int numerator = scanner.nextInt();
System.out.print( "Please enter an integer denominator: " );
int denominator = scanner.nextInt();
int result = quotient( numerator, denominator );
```

```
System.out.printf( "\nResult: %d / %d = %d\n", numerator,
denominator, result );
continueLoop = false; // input successful; end looping
} // end try
catch ( InputMismatchException inputMismatchException )
{
System.err.printf( "\nException: %s\n",
inputMismatchException );
scanner.nextLine(); // discard input so user can try again
System.out.println(
"You must enter integers. Please try again.\n" );
} // end catch

catch ( ArithmeticException arithmeticException )
{
System.err.printf( "\nException: %s\n", arithmeticException );
System.out.println(
"Zero is an invalid denominator. Please try again.\n" );
} // end catch
} while ( continueLoop ); // end do...while
} // end main
} // end class ExceptionHandle
```

Main Parts:

- **try .. catch block**: *try block*, which encloses the code that might throw an exception and the code that should not execute if an exception occurs (i.e., if an exception occurs, the remaining code in the try block will be skipped).

  catch block A catch block (also called a catch clause or exception handler) catches (i.e., receives) and handles an exception. A catch block begins with the keyword catch and is followed by a parameter in parentheses (called the exception parameter, discussed shortly) and a block of code enclosed in curly braces.

  At least one catch block or a *finally block* (discussed in Section 11.6) must immediately follow the try block. Each catch block specifies in parentheses an exception parameter that identifies the exception type the handler can process.

  The finally Block: The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs.

- **throws clause**: The *throws* keyword is used in method declaration, in order to *explicitly specify the exceptions that a particular method might throw.*

When a method declaration has one or more exceptions defined using throws clause then *the method-call must handle all the defined exceptions.*

To understand this point consider the following code:

```
  class Demo
{
   static void throwMethod() throws NullPointerException
   {
        System.out.println ("Inside throwMethod");
        throw new NullPointerException ("Demo");
   }
   public static void main(String args[])
   {
        try
        {
           throwMethod();
        }
        catch (NullPointerException exp)
        {
           System.out.println ("The exception get caught" +exp
              );
        }
    }
}
```

- **System.err**: Notice that we use the System.err (standard error stream) object to output error messages.

15.3 Exception Hierarchy

All Java exception classes inherit directly or indirectly from class Exception, forming an inheritance hierarchy.

See *Figure 11.3* in book (page 447).

Class Throwable has two subclasses:

Exception and Error.

- Class Exception and its subclassesfor instance, RuntimeException (package java.lang) and IOException (package java.io)represent exceptional situations that can occur in a Java program and that can be caught by the application.
- Class Error and its subclasses represent abnormal situations that happen in the JVM. Most Errors happen infrequently and should not be caught by applicationsits usually not possible for applications to recover from Errors.

15.3.1 Checked and Unchecked Exceptions

Unchecked Exception:

An Unchecked exception is an exception that occurs at the time of execution, these are also called as Runtime Exceptions, these include programming bugs, such as logic errors or improper use of an API. runtime exceptions are ignored at the time of compilation.

For example, if you have declared an array of size 5 in your program, and trying to call the 6th element of the array then an ArrayIndexOutOfBoundsExceptionexception occurs.

```
public class Unchecked_Demo {

  public static void main(String args[]){
     int num[]={1,2,3,4};
     System.out.println(num[5]);
  }

}
```

If you compile you will get no error, but if you execute the above program you will get exception as shown below:

```
 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
%  at Exceptions.Unchecked.main(Unchecked_Demo.java:8)
```

Note to Unchecked Exception:

- Java compiler does not enforce a catch-or-declare requirement for unchecked exceptions. It is upto the programmer.
- All exception types that are direct or indirect subclasses of class RuntimeException (package java.lang) are unchecked exceptions. These are typically caused by defects in your programs code.

Checked Exception:

A checked exception is an exception that *occurs at the compile time*, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the Programmer should take care of (handle) these exceptions.

For example, if you use FileReader class in your program to read data from a file, if the file specified in its constructor doesn't exist, then an FileNotFoundException occurs, and compiler prompts the programmer to handle the exception.

```
 import java.io.File;
import java.io.FileReader;

public class FilenotFound_Demo {

   public static void main(String args[]){
      File file=new File("E://file.txt");
      FileReader fr = new FileReader(file);
   }

}
```

If you try to compile the above program you will get exceptions as shown below:

```
 C:\>javac FilenotFound_Demo.java
FilenotFound_Demo.java:8: error: unreported exception FileNotFoundException; must
      FileReader fr = new FileReader(file);
                      ^
1 error
```

Note to Checked Exception:

- Java compiler enforces a catch-or-declare requirement for unchecked exceptions. Your code must handle it.
- All classes that inherit from class Exception but not class RuntimeException are considered to be checked exceptions.

16 Database Connectivity

Steps:

1. **Get and set the path of the appropriate jdbc driver:** Download the oracle jdbc driver from www.oracle.com. Perhaps its name should be something like `ojdbc14.jar`. It is a java archive (compressed) file.

   Place the jar file in a specific location in your PC (i.e. `C:\myjava\ojdbc14.jar`).

   Now set the class path:

   i. In Windows: `set CLASSPATH=%CLASSPATH%;C:\myjava\ojdbc14.jar;`
   ii. In Linux: `export CLASSPATH=".:/home/youname/myjava/ojdbc14.jar"`

2. **Install the Database and get the basic information:** Now you install the database (details are omitted). Assume you install the Oracle 10g XE version. Collect the following information:

(a) Name of the Database. It is normally XE in your case.

(b) User name and password of the Database (any valid user, this is not the Operating System user!).

(c) IP address of the Database machine. Your java application can connect to a remote machine if it is networked properly.

3. **Write a simple java application:** Now you are ready to test the connection. Your program should look like following code:

```
  import java.sql.*;
class OracleConScott{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@30.210.22.43:1521:mydatabse","USER","PASSWORD");

Statement stmt=con.createStatement();


while(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getDouble(3));

con.close();

}catch(Exception e)
  { System.out.println(e);
  }
 }
}
```

17 File Processing

17.1 Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , *FileInputStream* and *FileOutputStream*.

Example: Following is an example which makes use of these two classes to copy an input file into an output file:

```
  import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException
   {
      FileInputStream in = null;
      FileOutputStream out = null;

      try {
         in = new FileInputStream("input.txt");
         out = new FileOutputStream("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

17.2 Character Streams

Java Byte streams are used to perform input and output of 8-bit bytes, where as Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , FileReader and FileWriter.. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

```
  import java.io.*;

public class CopyFile {
   public static void main(String args[]) throws IOException
   {
      FileReader in = null;
      FileWriter out = null;

      try {
         in = new FileReader("input.txt");
         out = new FileWriter("output.txt");

         int c;
         while ((c = in.read()) != -1) {
            out.write(c);
         }
      }finally {
         if (in != null) {
            in.close();
         }
         if (out != null) {
            out.close();
         }
      }
   }
}
```

17.3 Read line by line

The BufferedReader class allows to read an input stream line-by-line via its `readLine()` method. In the example (given below) class, the test.txt file is read via a *FileReader*, which in turn is fed to a *BufferedReader*.

The BufferedReader is read line-by-line, and each line is appended to a *StringBuffer*, followed by a linefeed.

```
 import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class ReadFileLineByLine {

        public static void main(String[] args) {
                try {
                        File file = new File("test.txt");
                        FileReader fileReader = new FileReader(
                           file);
                        BufferedReader bufferedReader = new
                           BufferedReader(fileReader);
```

```
                    StringBuffer stringBuffer = new
                        StringBuffer();
                    String line;
                    while ((line = bufferedReader.readLine
                        ()) != null) {
                        stringBuffer.append(line);
                        stringBuffer.append("\n");
                    }
                    fileReader.close();
                    System.out.println("Contents of file:")
                        ;
                    System.out.println(stringBuffer.
                        toString());
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
}
```

17.4 Object Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

17.5 StringTokennizer

In Java, you can StringTokennizer class to split a String into different tokens by defined delimiter (space is the default delimiter).

18 Java Generics

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

18.1 Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the

generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

- All generic method declarations have a *type parameter* section delimited by angle brackets (`< and >`) that precedes the method's return type (`< E >` in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and *act as placeholders for the types of the arguments passed to the generic method*, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that *type parameters can represent only reference types, not primitive types* (like int, double and char).

Example:

```
public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // Display array elements
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:"
           );
```

```
            printArray( charArray ); // pass a Character array
        }
}
```

19 Java Wrapper Classes

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the `java.lang` package, which is imported by default into all Java programs.

Why?

- To include in a Collection. As example of when wrappers are used would be in Collections, you can have an `ArrayList<Integer>`, but not an `ArrayList<int>`.
- Wrapper classes are used to convert any primitive type into an object.The primitive data types are not objects, they do not belong to any class, they are defined in the language itself. While *storing in data structures which support only objects*, it is required to convert the primitive type to object first, so we go for wrapper class.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int   x = 25;

Integer   y = new Integer(30);
```

Below table lists wrapper classes in Java API with constructor details:

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Example: Here we will use only three methods of Integer Wrapper class such as

int compareTo(int i)[Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.]

boolean equals(Object intObj) [Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.].

floatValue() returns the value of this Integer as a float.

See Java API documentation for details.

```
public class WrapperDemo {



        public static void main (String args[]){

                Integer intObj1 = new Integer (25);

                Integer intObj2 = new Integer ("25");

                Integer intObj3= new Integer (35);

                //compareTo demo

                System.out.println("Comparing using compareTo
                    Obj1 and Obj2: " + intObj1.compareTo(intObj2
                    ));

                System.out.println("Comparing using compareTo
                    Obj1 and Obj3: " + intObj1.compareTo(intObj3
                    ));



                //Equals demo

                System.out.println("Comparing using equals Obj1
                     and Obj2: " + intObj1.equals(intObj2));

                System.out.println("Comparing using equals Obj1
                     and Obj3: " + intObj1.equals(intObj3));

                Float f1 = new Float("2.25f");

                Float f2 = new Float("20.43f");

                Float f3 = new Float(2.25f);

                System.out.println("Comparing using compare f1
                    and f2: " +Float.compare(f1,f2));
```

```
                System.out.println("Comparing using compare f1
                    and f3: " +Float.compare(f1,f3));



                //Addition of Integer with Float

                Float f = intObj1.floatValue() + f1;

                System.out.println("Addition of intObj1 and f1:
                     "+ intObj1 +"+" +f1+"=" +f );

        }


}
```

Output:

```
Comparing using compareTo Obj1 and Obj2: 0
Comparing using compareTo Obj1 and Obj3: -1
Comparing using equals Obj1 and Obj2: true
Comparing using equals Obj1 and Obj3: false
Comparing using compare f1 and f2: -1
Comparing using compare f1 and f3: 0
Addition of intObj1 and f1: 25+2.25=27.25
```

20 GUI