Java Notes

vinay bommana

Contents

Chapter 1

Classes and Interfaces

Inheritance

- Inheritance is a mechanism in which one object acquires all/ some of the properties and behaviours of **parent** object.
- Object - oriented programming allows classes to *inherit* commonly used state and behaviour from other classes.
- In Java, each class is allowed to to have one direct `super` class,
  and each `super` class has the potential for an unlimited number of subclasses.

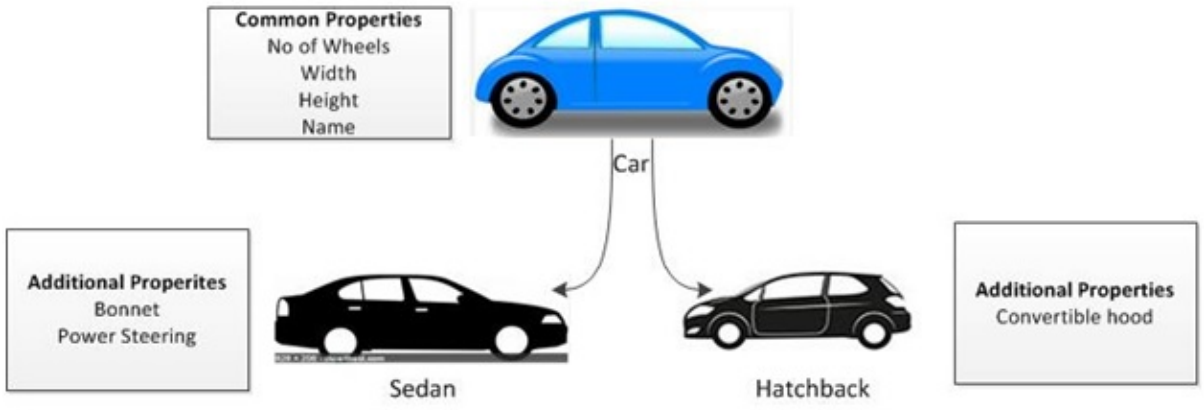

Figure 1.1: inheritance example

- Inheritance represents `is - a` relationship.
- The keyword `extends` is used to inherit properties from the existing class to a new class.
- The class from which the properties are inherited is called `super class`.
- The class which inherits its properties is called `sub class`.

super keyword

- Uses

  - `super` is used to refer immediate `Super` class instance variable.
  - super is used to invoke immediate `Super` class method.
  - `super()` is used to invoke immediate `Super` class constructor.

- If a method overrides the `Super` class's methods,
  we can invoke the overriden method through the use of `super` keyword.

```
public class Superclass {
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}

public class Subclass extends Superclass {
    @Override
    public void printMethod() {
        super.printMethod();
        System.out.println("Printed in Subclass.");
    }

    public static void main(String[] args) {
        Subclass s = new Subclass();
        s.printMethod();
    }
}
```

Compiling and executing the following Subclass prints the following output:

```
Printed in Superclass.
Printed in Subclass.
```

- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the super class.
- If the superclass does not have a no-argument constructor, you will get a
  compile time error.
- `Object` *does* have such a constructor, so if `Object` is the only superclass
  there is no problem.
- If a subclass constructor invokes a superclass constructor, either implicitly or explicitly, there will a whole chain of constructors called, all the way
  back to the constructor of `Object`. This is called **constructor chaining**

##Interfaces * Interfaces declares collection of **abstract** methods *that must be implemented by the implementing class* * Implementing class must define all the methods declared in the **Interface** or else the class must be declared as abstract. * Variables declared in as interfaces are `public` , `static` , and `final` by default.

Anonymous Inner Classes

Anonymous inner classes combines the process of defining an inner class and creating an object of inner class into one step. it is a class without any name *anonymous*

- They can't have constructors (*no name*)
- No `extends` or `implements` keyword is required.
- The Name given after `new` is taken as `class` or `interface`.

##HashMap class **syntax** for the HashMap invocation is:

```
HashMap<E> h = new HashMap<E>();
```

##Stack class

```
public class Stack<E> extends Vector<E>
```

Stack is a subclass of `Vector` that implements a standard last-in, first-out stack. stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by `Vector`, and add several of its own.

`stack()`

When a stack is first created, it contains no items

Apart from the methods inherited from its parent class `Vector`, Stack defines the following methods -

1. `boolean empty()`
   Tests if this stack is empty.
   Returns true if the stack is empty, and returns false if the stack contains elements
2. `Object peek()`
   Returns the element on the top of the stack, but does not remove it.
3. `Object pop()`
   Returns the element on the top of the stack, removing it in the process.
4. `Object push(Object element)`
   Pushes the element onto the stack. Element is also returned.
5. `int search(Object element)`
   Searches for element in the stack.
   If found, its offset from the top of the stack is returned.
   Otherwise, 1 is returned.

syntax for implementation of stack

```
Stack st = new Stack();

try {
    System.out.println(st);
}catch (EmptyStackException e) {
    System.out.println("Empty stack");
}
```

A more complete and consistent set of **LIFO** stack operations is provided by the *Deque* interface and its implementation

```
Deque<Integer> stack = new ArrayDeque<Integer>();
```

Methods to ponder:

`push()` has the exactly the same effect as `Vector.addElement()`

##HashSet

```
public class HashSet<E> extends AbstractSet<E> implements Set<E>
, Cloneable, Serializable
```

* Note that this implementation is not synchronized.

- If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it *must* be `synchronized`
- It should be noted that a Set has no duplicate elements

This class is a member of the **Java Collections Framework**

Syntax for the implementation

```
HashSet<String> set = new HashSet<String>();
set.add("jon");
```

##Comparator Interface

```
public interface Comparator<T>
```

- This interface is found in `java.util` package

- A comparison function, which imposes a *total* ordering on some collection of objects.
- Comparators can be passed to a sort method `Collections.sort()` or `Arrays.sort()` to allow precise control over the sort order.
- Comparators can also be used to control the order of certain data structures
  viz sorted sets or sorted maps, or to provide an ordering for collections of objects that don't have a natural ordering.

####Methods * `compare(T o1, T o2)` compares its two args for order * `equals(Object obj)` indicates whether some other object is *equal to* this comparator.

####Example

```
import java.util.*;
class NameComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        Student s1 = (Student) o1;
        Student s2 = (Student) o2;

        return s1.name.compareTo(s2.name);
    }
}
```

In the implementation we use
`Collections.sort(arrayList, new NameComparator());`

- The `Comparator` implementation is further simplified by using lambda expressions (refer `LambdaExpressions` chapter)

Event Handling

Event handling is the mechanism which controls the events and decides what should happen if an event occurs. This mechanism has the code which is known as `Event Handler` and that gets executed when an event occurs. Java uses the `Delegation Event Model` to handle the events. This model defines the standard mechanism to generate and handle the events.

- Any program that uses GUI (graphical user interface) such as Java application written for windows is event driven.
- Event describes the change of state of any object.
- **Example**: Pressing a button, Entering a character in Textbox

Reacting to an Event

- When an event is detected, a method is invoked. The invoked method is called `Call-Back Method`.
- This Style of programming is called EventDriven Programming.

Handling events

- A source generates an Event and send it to one or more listeners registered with the source.
- Once event is received by the listener, they process the event and then return.
- Events are supported by a number of java packages
  `java.util`, `java.awt`,`java.awt.event`.

The Event handling involves four types of classes:

1. `Event Sources.`
2. `Event Classes.`
3. `Event Listeners.`
4. `Event Adapters.`

Example of Event Handling

```
import java.awt.*;
import java.awt.event.*;
class MyButtons extends Frame implements ActionListener {
    MyButtons() {
        Button b1 = new Button("Red");
        Button b2 = new Button("Green");
        Button b3 = new Button("Blue");
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(b3);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String st = ae.ActionCommand();
        if(str.equals("Red")) {
            setBackground(Color.red);
        }
        if(str.equals("Green")) {
            setBackground(Color.green);
        }
        if(str.equals("Blue")) {
            setBackground(Color.blue);
```

```
        }
    }

    public static void main(String[] args) {
        MyButtons mb = new MyButtons();
        mb.setSize(500, 400);
        mb.setTitle("My Buttons");
        mb.show();
        mb.addWindowListener(new MyClass());
    }
}

class MyClass extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
```

####Java Event Classes and Listener Interfaces

Event Classes	Listener Interfaces
`ActionEvent`	`ActionListener`
`MouseEvent`	`MouseListener` and `MouseMotionListener`
`MouseWheelEvent`	`MouseWheelListener`
`KeyEvent`	`KeyListener`
`ItemEvent`	`ItemListener`
`TextEvent`	`TextListener`
`AdjustmentEvent`	`AdjustmentListener`
`WindowEvent`	`WindowListener`
`ComponentEvent`	`ComponentListener`
`ContainerEvent`	`ContainerListener`
`FocusEvent`	`FocusListener`

####Steps to perform Event Handling.

- Register the component with the Listener.

For registering the component with Listener, many classes provide the registration methods.

#####For Example:

- Button
  `public void addActionListener(ActionListener a){}`

- MenuItem
  `public void addActionListener(ActionListener a){}`

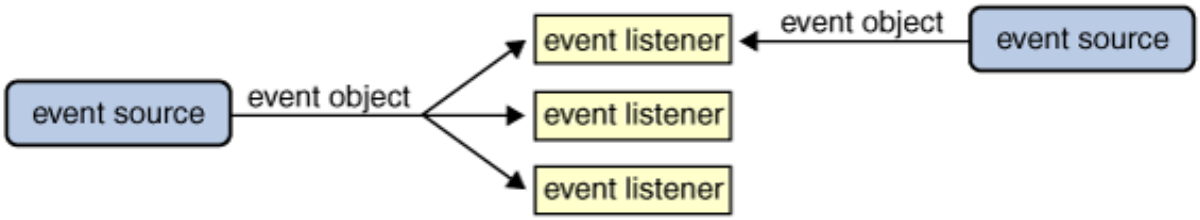

Figure 1.2: event_handle

- TextField
```
public void addActionListener(ActionListener a){}
public void addTextListener(TextListener a){}
```

- TextArea
```
public void addTextListener(TextListener a){}
```

- Checkbox
```
public void addItemListener(ItemListener a){}
```

- Choice
```
public void addItemListener(ItemListener a){}
```

- List
```
public void addActionListener(ActionListener a){}
public void addItemListener(ItemListener a){}
```

Event Handling Codes.

We can put event handling code into one of the following places:

1. Same Class
2. Other Class
3. Anonymous Class

Factory methods

Factory methods are static methods that return an instance of the native class.

Factory methods: * have names, unlike constructors, which can clarify code * do not need to create a new object upon each invocation - objects can be cached and reused, if necessary. * can return a subtype of their return type - in particular, can *return an object whose implementation class is unknown to the caller*. * This is a very valuable and widely used feature in many frameworks which use interfaces as the return type of static methods.

```
public final class ComplexNumber {
    /**
    * static factory method returns an object of this class.
    */
    public static ComplexNumber valueOf(float aReal, float aImaginary) {
        return new ComplexNumber(aReal, aImaginary);
    }

    /**
    * Caller cannot see this private constructor.
    *
    *
    * The only way to build a ComplexNumber is by calling the static factory method.
    */
    private ComplexNumber(float aReal, float aImaginary) {
        fReal = aReal;
        fImaginary = aImaginary;
    }

    private float fReal;
    private float fImaginary;

}
```

Another Example

```
public class Coordinate {
    private double x_coord;
    private double y_coord;

    private Coordinate(double x, double y) {
        x_coord = x;
        y_coord = y;
    }

    public static final Coordinate fromXY(double x, double y) {
    return new Coordinate(x, y);
    }

    public static final Coordinate fromPolar(double dist, double angle) {
        return new Coordinate(dist * Math.cos(angle), dist * Math.sin(angle));
    }
}
```

Builder Patterns in java

#I/O Streams * Java I/O (*Input* and *Output*) is used to process the input and produce the output. * Java uses the concept of stream to make I/O operation fast. * The `java.io` package contains all the classes required for input and output operations.

##Streams * A Stream is a sequence of data. * In Java a stream is composed of bytes. * It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with console.

1. `System.out` : standard output stream.
2. `System.in` : standard input stream.
3. `System.err` : standard error stream.

```
int i = System.in.read(); // returns ASCII code of first character.
System.out.print((char)i); // will print the character.
```

###Byte Streams * Treat the stream as a collection of bytes. * Used when data that is input or output is to be treated as bytes. * All byte streams are derived from either `InputStream` or `OutputStream`.

##Input and Output Streams * An Input stream moves data from external source. * An Output stream sends data to external target. * All Streams behave in the same manner even if the actual physical devices are different.

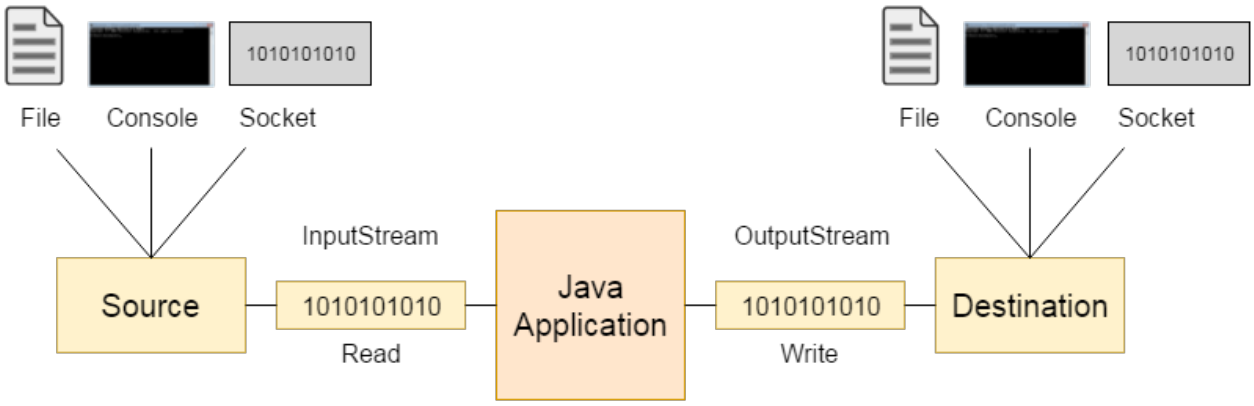

Figure 1.3: java io flow

###OutputStream Class * OutputStream class is an abstract class. * It is super class of all classes representing an output stream of bytes. * An Output stream accepts output bytes and sends them to some sink.

####Useful methods of OutputStream | Method | Description | | ———————————— | ———————————————————— | | `public void write(int)throws IOException` | is used to write a byte to the current output stream | | ———————————— | ———————————————————— | | `public void write(byte[]) throws IOException` | is used to write an array of byte to the current output stream. | | ———————————— | ———————————————————— | | `public void flush() throws IOException` | flushed the current output stream | | ———————————— | ———————————————————— | | `public void close() throws IOException` | is used to close the current output stream. |

###InputStream Class * `InputStream` class is an abstract class. * It is the super class of all classes representing an input stream of bytes. * `InputStream` subclasses include `FileInputStream`, `BufferedInputStream` and the `PushbackInputStream` etc.,

####Useful Methods in InputStream class

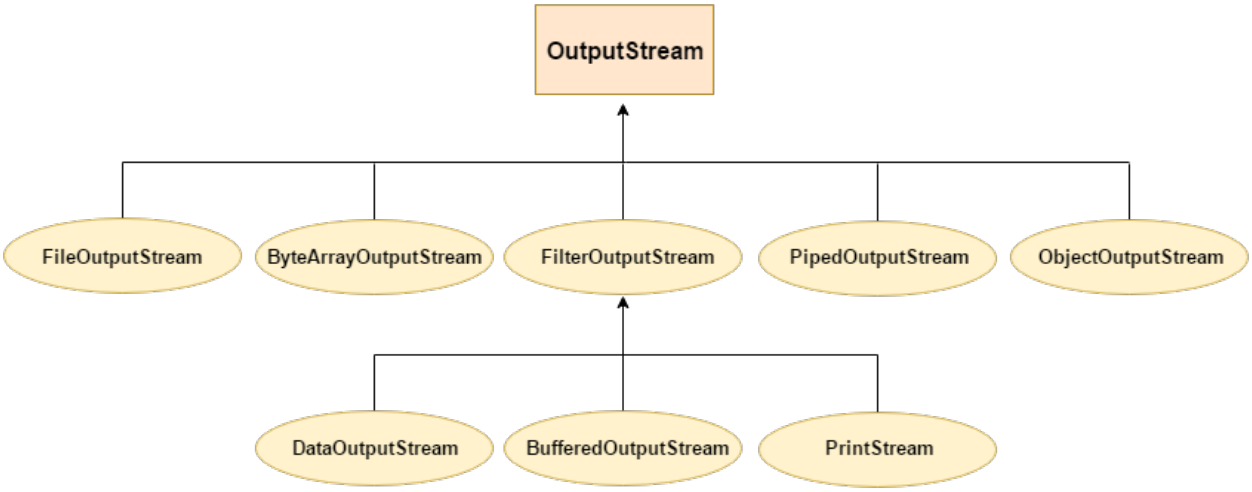

Figure 1.4: java OutputStream

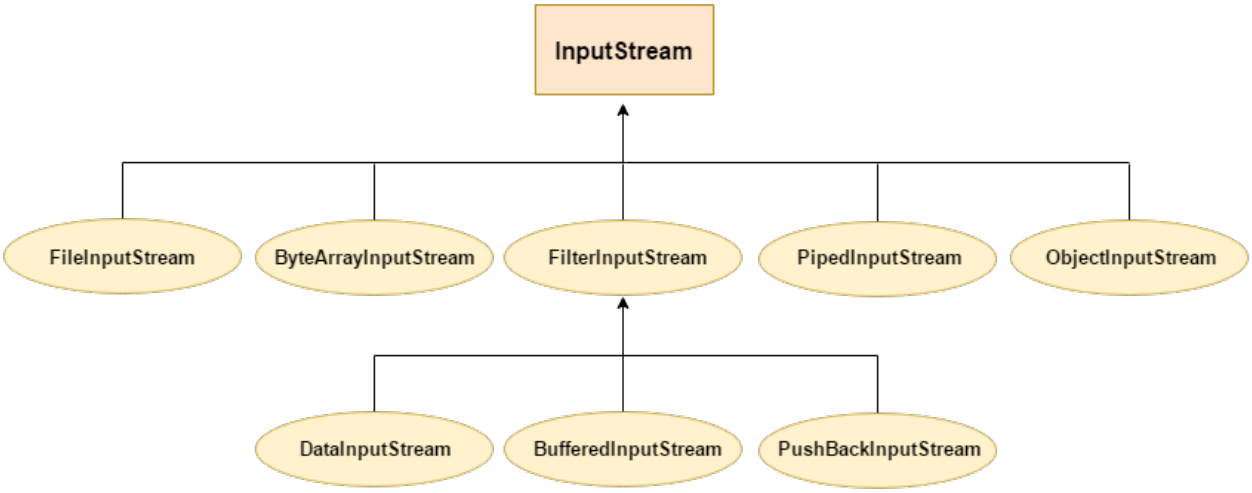

Figure 1.5: java input hierarchy

Method	Description
`public abstract int read() throws IOException`	reads the next byte of data from the InputStream. -1 at EOF
`public int availabe() throws IOException`	returns an estimate of the number of bytes that can be read
`public void close() throws IOException`	is used to close the current input stream

- Java `InputStream`'s are used for reading byte based data

```
InputStream inputstream = new FileInputStream(~/input.txt);
int data = inputstream.read();
while(data != -1) {
    //do something with the data
    data = inputstream.read();
}
inputstream.close();
```

###Character Streams * Treat the streams as collection of characters. * They use Unicode and therefore, can be internationalized. * At the top of the hierarchy there are two `abstract` classes.
- `Reader`
- `Writer`

##BufferedReader class

```
public class BufferedReader extends Reader
```

- It improves the performance by buffering characters so as to prove efficient reading of characters.
- It is derived from the `Reader` class
- Buffer size may be specified using parameters to `constructor()` or default size may be used.

```
BufferedReader(Reader in)
BufferedReader(Reader in, int buffersize)
```

####Methods

`readLine()` method.

- it reads a line of text from the reader provided as parameter to constructor. A line is considered to be determined by linefeed (ASCII code 10) or carriage return (ASCII code 13) or by both.

- returns `null` on end-of-file(EOF).

####Example

```
import java.io.*;

class LineRead {
    public static void main(String[] args) {
        //Passing filename through command line argument
        FileReader fr = new FileReader(args[0]);
        BufferReader br = new BufferedReader(fr);
        String line;
        line = br.readLine();

        while(line! = null) {
            System.out.println(line);
            line = br.readLine();
        }
        br.close();
        fr.close();
    }
}
```

#Lambda Expressions * Lambda expressions are introduced in Java 8. * Lambda expression facilitates functional programming and simplifies the development.

A Lambda expression is characterised by the following syntax-

```
parameter -> expression body
```

###Example

```
interface Addable {
    int add(int a, int b);
}

public class AdditionDemo {
    public static void main(String[] args) {
        //Multiple parameters in lambda expressions
        Addable ad1 = (a, b) -> (a + b);
        System.out.println(ad1.add(10, 20));

        //Multiple parameters with datatypes
        Addable ad2 = (int a, int b) -> (a + b);
        System.out.println(ad2.add(100, 200));
    }
}
```

- Here the `interface Addable` is called `functional interface`.

###Simple implementation of for each loop

```
import java.util.*;
public class ForEachExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("c");

        list.forEach(
            (n) -> System.out.println(n);
        );
    }
}
```

###Java Lambda Expression : single parameter

```
@FunctionalInterface
interface Sayable {
    public String say(String name);
}

public class SingleParameterExample {
    public static void main(String[] args) {
        Sayable s1 = (name) -> {
            return "Hello, " + name;
        }

        System.out.println(s1.say("jon"));
    }
}
```

###Java Lambda Expression : Comparator

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Product {
    int id;
    String name;

    public Product(int id, String name) {
        super(); // called implicitly anyway
        this.id = id;
        this.name = name;
    }
}

public class ComparatorExample {
    public static void main(String[] args) {
        List<Product> list = new ArrayList<Product>();
```

```
        list.add(new Product(1, "Ice"));
        list.add(new Product(2, "Long Claw"));
        list.add(new Product(3, "Dawn"));

        System.out.println("Sorting according to name");
        //lambda expression implementation
        Collections.sort(list, (p1, p2) -> {
            return p1.name.compareTo(p2.name);
        });

        //printing the sorted list
        for(Product p: list) {
            System.out.println(p.id + " " + p.name);
        }
    }
}
```

#BigNumbers

##BigInteger Class

`public class BigInteger extends Number implements Comparable<BigInteger>`

- BigInteger class is used for mathematical operation which involves very **big** integer calculations.
- which are outside the limit of all available primitive data types.
- BigIntegers are `Immutable` arbitrary-precision integers.
- Additional to that of the `java.lang.Math`, BigInteger provides operations for
  - modular arithmetic
  - GCD calcualtion
  - primality testing
  - prime generation
  - bit manipulation
  - and a few other miscellanous operations.

####Example

```
import java.math.BigInteger;

public class MainClass {
    public static void main(String[] args) {
        System.out.println("Here's a Long.MAX_VALUE: " + Long.MAX_VALUE); // 9223372036854775807
        BigInteger bInt = new BigInteger("99999999999999999999999999999");
    }
}
```

Adding to BigInteger values.

```
import java.math.BigInteger;

public class AddingTwoNum {
    public static void main(String[] args) {
        BigInteger a = new BigInteger("999999999999999");
        BigInteger b = new BigInteger.valueOf(1);
```

```
        a = a.add(b);
        System.out.println(a);
    }
}
```

For subtracting two `BigInteger` numbers use `a.sub(b)` regarding the former eg.,
For multiplying two `BigInteger` numbers use `a.multiply(b)` regarding the former eg.,
For dividing two `BigInteger` numbers use `a.divide(b)` regarding the former eg.,

- Other methods common that can be applied to the BigInteger class
  - `gcd()`
  - `max()`
  - `min()`
  - `mod()`
  - `remainder()`
  - `pow(_exponent_)`

####Comparison between two BigIntegers

- In order to compare two `BigInteger` numbers we need to use `compareTo` method.
- The method declaration for `compareTo`

```
public int compareTo(BigInteger val)
```

- for example we need to write a while loop to until a `BigInteger` is less than `BigInteger.ZERO` then

```
while(a.compareTo(b) > 0) {
    // do something
}
```

- `compareTo` method returns -1 for less than, 0 for equal to, 1 for greater than the number.

##BigDecimal Class

```
public class BigInteger extends Number implements Comparable<BigInteger>
```

- Immutable, arbitrary-precision signed decimal numbers.
- A `BigDecimal` consists of an arbitrary precision integer *unscaled* value and a 32 bit integer-scale.
- The `java.math.BigDecimal` class provides operations for:
  - arithmetic
  - scale manipulation
  - rounding
  - comparison
  - hashing
  - format conversion
- Two types of operations are provided for manipulating the scale of a Decimal number:

  - scaling / rounding operations
  - decimal point motion operations

- These are used for monetary values.

####Example for BigDecimal and Monetary values

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class MonetaryExample {
    public static void main(String[] args) {
        BigDecimal total = new BigDecimal("123.45");
        BigDecimal discountPercent = new BigDecimal("0.10");
        BigDecimal discount = total.multiply(discountPercent);
        BigDecimal beforeTax = total.subtract(discount);
        beforeTax = beforeTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal salesTaxPercent = new BigDecimal("0.05");
        BigDecimal salesTax = beforeTax.multiply(salesTaxPercent);
        salesTax = salesTax.setScale(2, RoundingMode.HALF_UP);
        BigDecimal result = beforeTax.add(salesTax);
        result = result.setScale(2, RoundingMode.HALF_UP);
        System.out.println("Subtotal: " + total);
        System.out.println("Discount: " + discount);
        System.out.println("SubTotal after discount: " + beforeTax);
        System.out.println("Sales Tax: " + salesTax);
        System.out.println("Total: " + result);
    }
}
```

```
java.math.BigDecimal.setScale(int newScale, RoundingMode roundingMode)
```

####for example: - new BigDecimal(“2.5467”).setScale(3, RoundingMode.HALF_UP) - results in 2.547

```
BigDecimal smallNumber = new BigDecimal("0.01234");
smallNumber.round(new MathContext(2, RoundingMode.HALF_UP)));
// Result   = 0.012
smallNumber.setScale(2, RoundingMode.HALF_UP));
// Result   = 0.01

BigDecimal number = new BigDecimal("1.01234");
number.round(new MathContext(2, RoundingMode.HALF_UP)));
// Result   = 1.0
number.setScale(2, RoundingMode.HALF_UP));
// Result   = 1.01
```

In general the rounding modes and precision setting determine how operations return results
with a limited number of digits when the exact result has more digits (perhaps infinitely many in the case of division) than the number of digits returned.
First, the total number of digits to return is specified by the MathContext’s precision setting;
this determines the result’s precision.
The digit count starts from the leftmost nonzero digit of the exact result.
The rounding mode determines how any discarded trailing digits affect the returned result.

Chapter 2

Regular Expressions

- Java Regular Expressions are very similar to perl.
- Java provides `java.util.regex` package for pattern matching for regular expressions.

Regular expression

A regular expression is a special sequence of characters that helps to match or find other strings or set of strings, using a specialized syntax held in a pattern.

- `java.util.regex` package consists of mainly three classes:
  - **Pattern Class**
    * A `Pattern` object is a compiled representation of a regular expression.
    * To create a pattern, invocation of one of the `public static` **compile** methods is required, which will return a pattern object.
    * These methods accept a Regular expression as their first object.
  - **Matcher Class**
    * `Matcher` Object interprets the pattern.
    * the `matcher` object is obtained by invoking the `Matcher` method on the **Pattern** object.

Chapter 3

javap

javap is a command line tool that disassembles Java class files: it takes apart our class files, and reveals what's inside. The tool translates a binary format class file to human readable code.

- As for what the `p` in `javap` stands for, all evidence points to `prints`, since the javap command prints out the bytecode within the class.
- A nice way in which we can `javap`, is to explore exceptions.

Chapter 4

8 guidelines of exception handling

1. Use exceptions only for exceptional scenarios.
2. Use checked exceptions for recoverable conditions and runtime exceptions for programming errors.
3. Avoid unnecessary use of checked exceptions.
4. Favour the use of standard exceptions.
5. Throw exceptions appropriate to the abstraction.
6. Document all exceptions thrown by each method.
7. Include failure - capture information in detail messages.
8. Don’t ignore exceptions

Chapter 5

Java Notes

by vinay bommana

- classes and interfaces
- Streams
- lambda expressions
- Big Numbers
- Regular expressions