UNIT-5 Object Oriented Thinking (Chapter 10,11 - Liang)

5.1 Class abstraction and Encapsulation

Explain about Class Abstraction and Encapsulation

- *Class abstraction is the separation of class implementation from the use of a class.*
- *The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation.*

- ***class abstraction*** separates class implementation from how the class is used. The creator of a class describes the functions of the class and lets the user know how the class can be used as shown in UML diagram.

- The details of implementation are encapsulated and hidden from the user. This is called ***class encapsulation***. For example, you can create a **Circle** object by keeping *radius* instance variable as *private*, provided with (getter) *getRadius()* and (setter) *setRadius()* method.

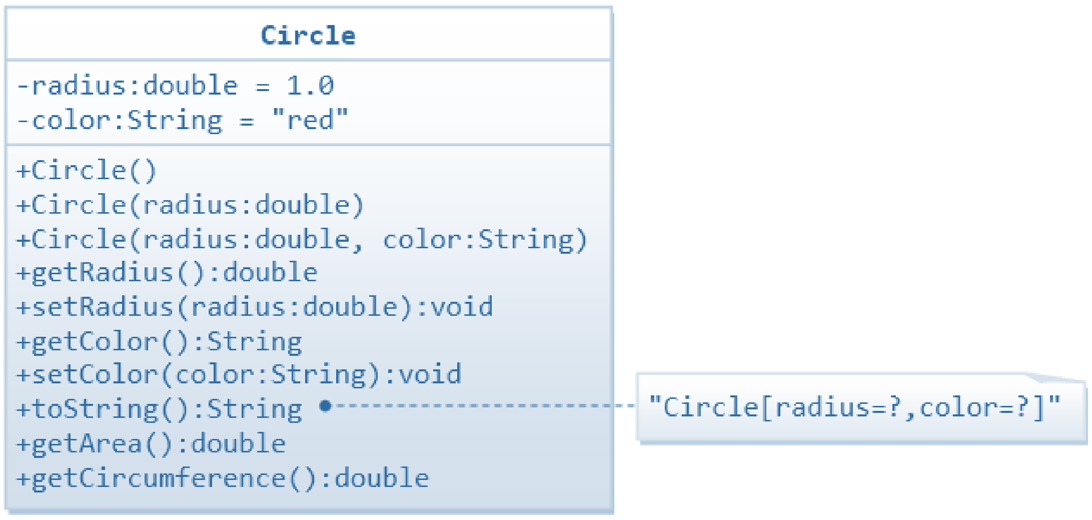


Figure: UML diagram for Circle class that shows abstraction and encapsulation.

Abstraction	Encapsulation
Abstraction is a feature of OOPs that hides the unnecessary detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a single entity or unit so that the data can be protected from the outside world.

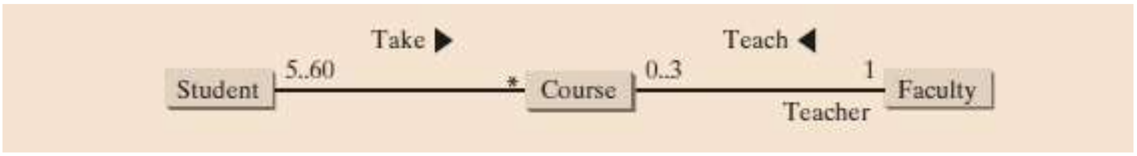
It solves an issue at the design level.	Encapsulation solves an issue at implementation level.
It can be implemented using abstract classes and interfaces.	It can be implemented by using the access modifiers (private, public, protected).
In abstraction, we use abstract classes and interfaces to hide the code complexities.	We use the getters and setters methods to hide the data.

5.2 Thinking in Objects and Class Relationships

Objects and class relationships OR Write a note on class relationships.

- The **procedural** paradigm focuses on designing **methods**.
- The **object-oriented** paradigm **couples data and methods** together into **objects**.
- To design classes, you need to explore the relationships among classes. The common relationships among classes are **association, aggregation, composition, and inheritance**.

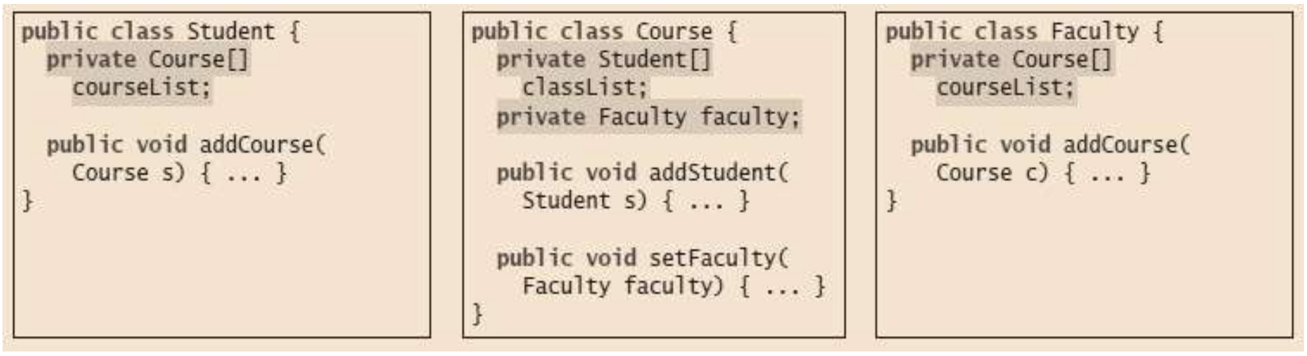
❖ Association Relationship



The labels are *Take* and *Teach*. Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the direction indicates that a student takes a course. *teacher* is the role name for **Faculty**.

The character ***** means an unlimited number of objects, and the interval **m..n** indicates that the number of objects is between **m** and **n**, inclusively.

For example, the relationships in above figure may be implemented using the classes in below figure.



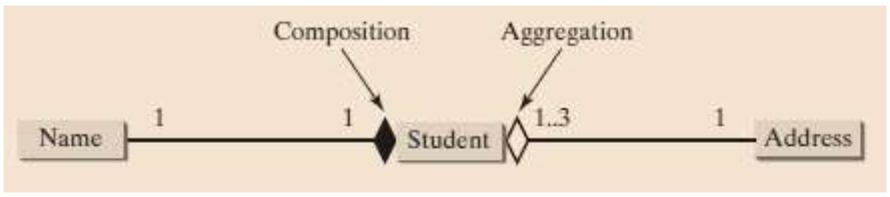
```
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

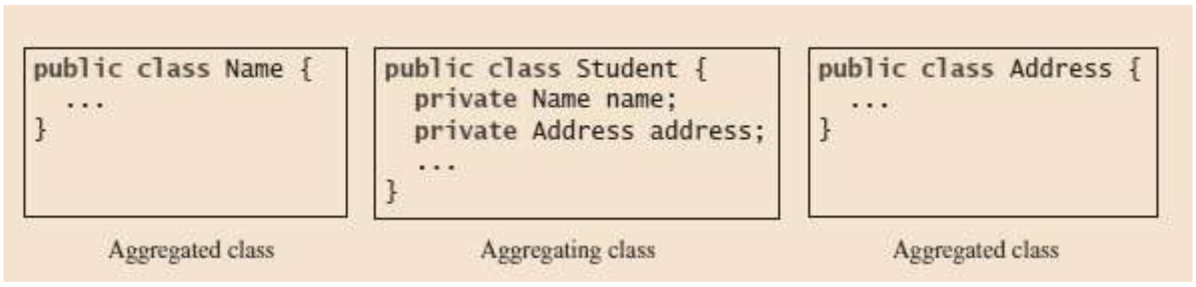
```
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

❖ Aggregation and Composition Relationship

Aggregation models ***has-a*** relationships.
For example, "a student has a name" is a composition relationship between the **Student** class and the **Name** class, whereas "a student has an address" is an aggregation relationship between the **Student** class and the **Address** class, since an address can be shared by several students.



Each student has only one multiplicity (more than 1 student) —to an address—and each address can be shared by up to **3** students. Each student has one name, and a name is unique for each student.



```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

Aggregating class

```
public class Address {
  ...
}
```

Aggregated class

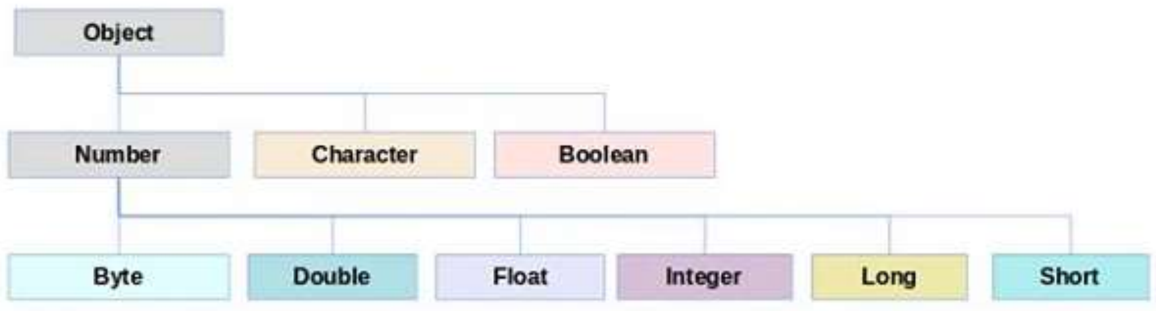
5.3 Primitive data type and wrapper class types

What is Wrapper class in Java? Explain with examples.

A primitive type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API.

- In Java, the wrapper class provides a mechanism for converting primitives to objects and objects to primitives.

These wrapper classes are defined in the *java.lang* package and are hierarchically structured as follows:



- The eight classes of the **java.lang** package are known as wrapper classes in Java.
- The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Example: **Integer** and **Double** wrapper classes inbuilt methods

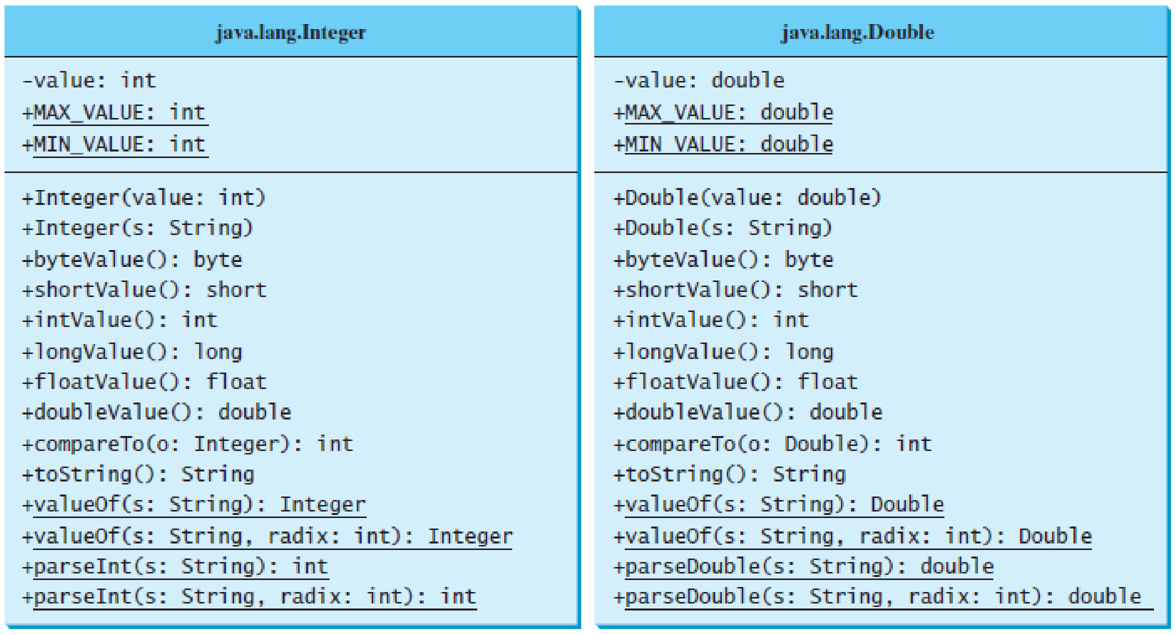


Figure: constants, constructors and methods of Integer and Double wrapper classes

- The wrapper classes do not have no-arg constructors.
- The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

The following statement display the maximum integer (2,147,483,647)

```
System.out.println("The maximum integer is " + Integer.MAX_VALUE);
```

- Numeric wrapper classes are very similar to each other. Each contains the methods **doubleValue(), floatValue(), intValue(), longValue(), shortValue(),** and **byteValue().** These methods "convert" objects into primitive type values.

```
new Double(12.4).intValue() returns 12;
new Integer(12).doubleValue() returns 12.0;
```

- The numeric wrapper classes contain the **compareTo** method for comparing two numbers that returns **1**, **0**, or **-1**, if this number is greater than, equal to, or less than the other number.
- For example,

```
new Double(12.4).compareTo(new Double(12.3)) returns 1;
new Double(12.3).compareTo(new Double(12.3)) returns 0;
new Double(12.3).compareTo(new Double(12.51)) returns -1;
```

- The numeric wrapper classes have a useful static method, **valueOf (String s)**. This method creates a new object initialized to the value represented by the specified string. For example,

```
Double doubleObject = Double.valueOf("12.4");
Integer integerObject = Integer.valueOf("12");
```

These two methods are in the Integer class

```
public static int parseInt(String s)
public static int parseInt(String s, int radix)
```

For example,

```
Integer.parseInt("11", 2)  //radix 2 for binary returns decimal 3
Integer.parseInt("12", 8)  //radix 8 for octal returns decimal 10
Integer.parseInt("13", 10) //radix 10 for decimal returns decimal 10
Integer.parseInt("1A", 16) //radix 16 for hexa-decimal returns decimal 26
```

What is Auto-Boxing and Auto-Unboxing in wrapper classes?

Since J2SE 5.0, auto-boxing and auto-unboxing feature convert primitives into objects and objects into primitives automatically.

Auto-boxing:

- The automatic **conversion of primitive data type into its corresponding wrapper class** is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer

public class WrapperExample1{
      public static void main(String args[]){
      int a=10;
      Integer i=Integer.valueOf(a); //converting int into   Integer explicitly
      Integer j=a; //autoboxing, now compiler will write Integer.valueOf(a) internally
      System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
10 10 10
```

Auto-Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
- It is the reverse process of autoboxing.
- Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2{
      public static void main(String args[]){
            Integer a=new Integer(5);
            int i=a.intValue();     //converting Integer to int explicitly
            int j=a; //unboxing, now compiler will write a.intValue() internally
            System.out.println(a+" "+i+" "+j);
      }
}
```

Output:

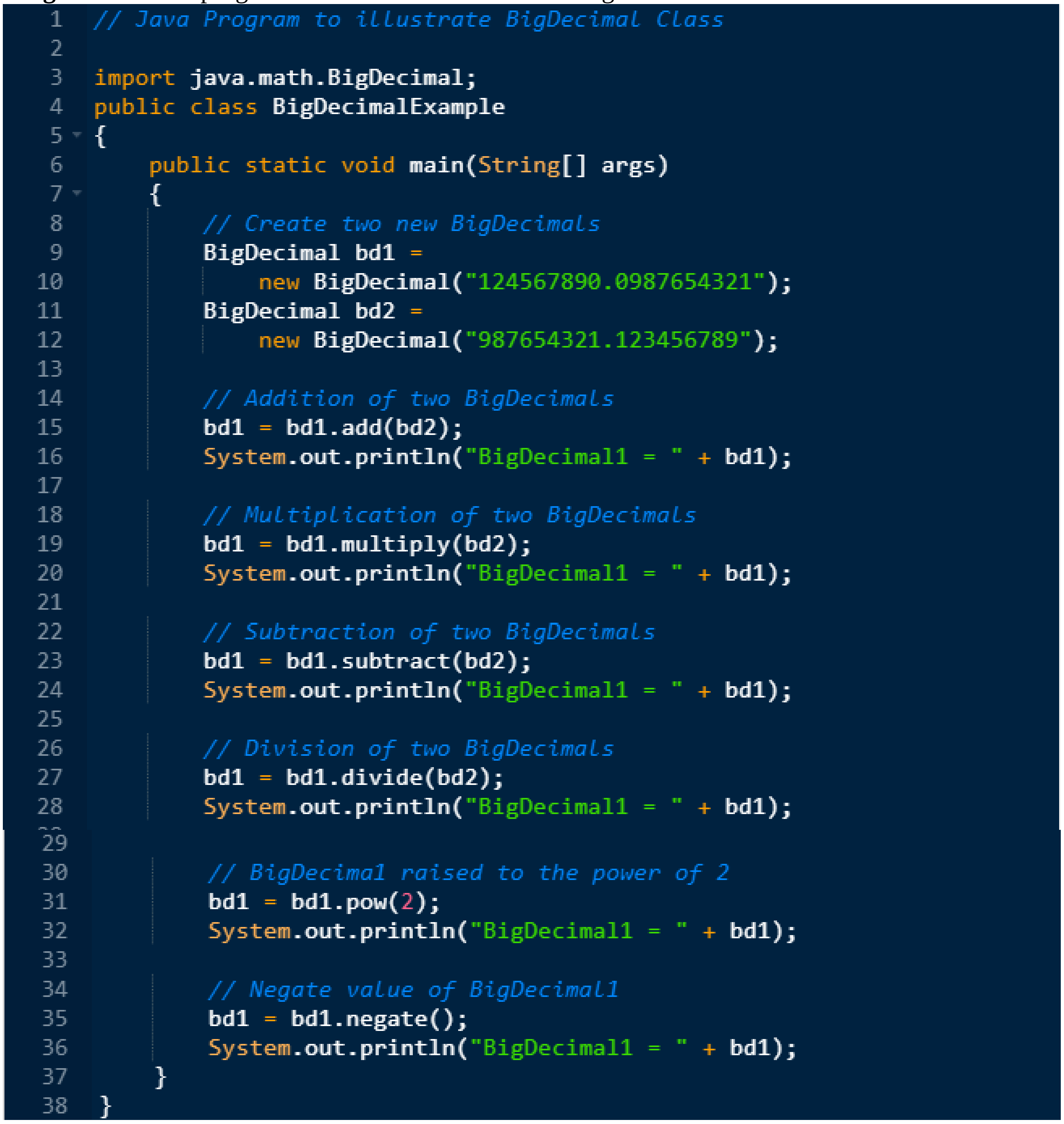
```
5 5 5
```

5.4 BigInteger and BigDecimal class

Explain the use of BigInteger and BigDecimal with the help of programs.

The ***BigInteger*** *and* ***BigDecimal*** *classes can be used to represent integers or decimal numbers of any size and precision.*

Program: Write a program to demonstrate methods of BigDecimal class.

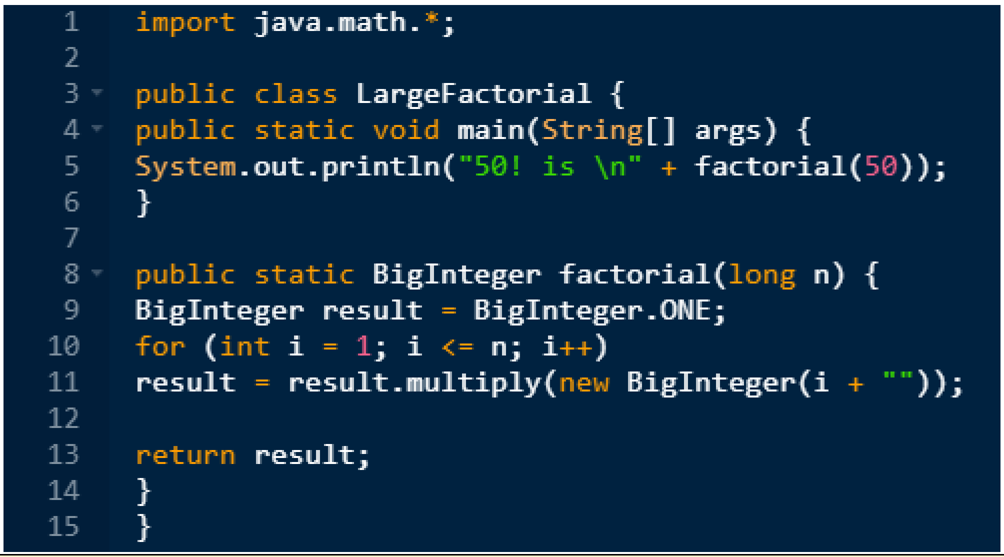


```java
// Java Program to illustrate BigDecimal Class

import java.math.BigDecimal;
public class BigDecimalExample
{
    public static void main(String[] args)
    {
        // Create two new BigDecimals
        BigDecimal bd1 =
            new BigDecimal("124567890.0987654321");
        BigDecimal bd2 =
            new BigDecimal("987654321.123456789");

        // Addition of two BigDecimals
        bd1 = bd1.add(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Multiplication of two BigDecimals
        bd1 = bd1.multiply(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Subtraction of two BigDecimals
        bd1 = bd1.subtract(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // Division of two BigDecimals
        bd1 = bd1.divide(bd2);
        System.out.println("BigDecimal1 = " + bd1);

        // BigDecimal1 raised to the power of 2
        bd1 = bd1.pow(2);
        System.out.println("BigDecimal1 = " + bd1);

        // Negate value of BigDecimal1
        bd1 = bd1.negate();
        System.out.println("BigDecimal1 = " + bd1);
    }
}
```

Output:

```
BigDecimal1 = 1112222211.2222222211
BigDecimal1 = 1098491072963113850.7436076939614540479
BigDecimal1 = 1098491071975459529.6201509049614540479
BigDecimal1 = 1112222210.2222222211
BigDecimal1 = 1237038244911605079.77528397755061728521
BigDecimal1 = -1237038244911605079.77528397755061728521
```

Program: Write a program to find factorial of 50 using BigInteger class.



```java
import java.math.*;

public class LargeFactorial {
public static void main(String[] args) {
System.out.println("50! is \n" + factorial(50));
}

public static BigInteger factorial(long n) {
BigInteger result = BigInteger.ONE;
for (int i = 1; i <= n; i++)
result = result.multiply(new BigInteger(i + ""));

return result;
}
}
```

Output:

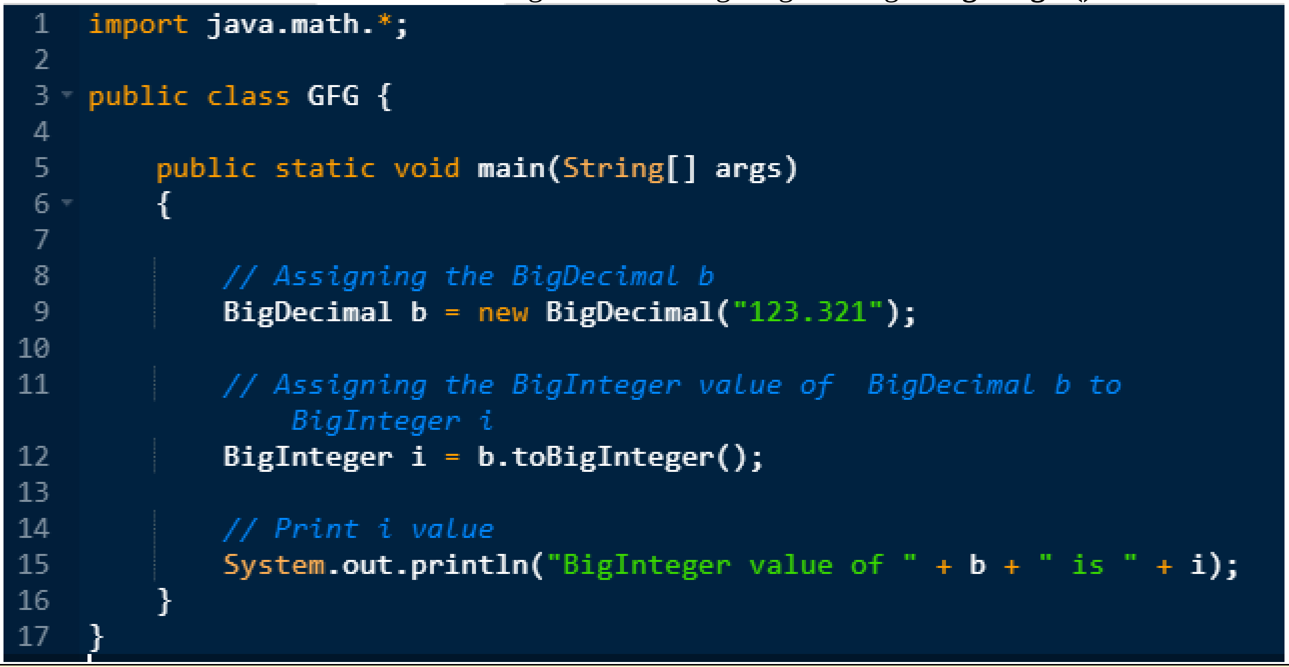
```
50! is
30414093201713378043612608166064768844377641568960512000000000000
```

There is no limit to the precision of a **BigDecimal** object. The **divide** method may throw an **ArithmeticException** if the result cannot be terminated. However, you can use the overloaded **divide(BigDecimal d, int scale, int roundingMode)** method to specify a scale and a rounding mode to avoid this exception, where **scale** is the maximum number of digits after the decimal point. For example, the following code creates two **BigDecimal** objects and performs division with scale **20** and rounding mode **BigDecimal.ROUND_UP**.

```java
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);
System.out.println(c);
```

The output is **0.33333333333333333334**.

Program: Write a program to convert BigDecimal to BigInteger using **toBigInteger()** Method in Java



```java
import java.math.*;

public class GFG {

    public static void main(String[] args)
    {

        // Assigning the BigDecimal b
        BigDecimal b = new BigDecimal("123.321");

        // Assigning the BigInteger value of  BigDecimal b to
          BigInteger i
        BigInteger i = b.toBigInteger();

        // Print i value
        System.out.println("BigInteger value of " + b + " is " + i);
    }
}
```

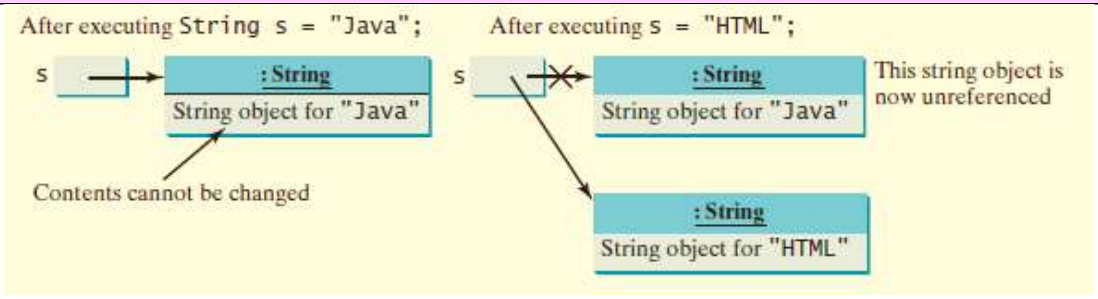
```
Output:
BigInteger value of 123.321 is 123
```

5.5 String class, String Builder and String Buffer class

Immutable Strings and Interned Strings

A **String** object is **immutable**; its contents cannot be changed. Does the following code change the contents of the string?

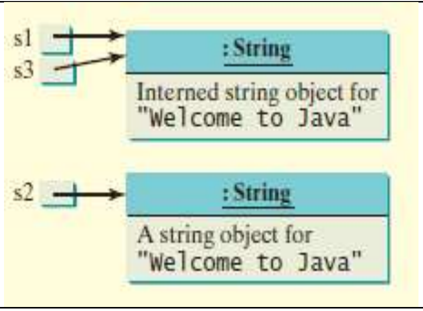
```java
String s = "Java";
s = "HTML";
```


After executing String s = "Java";
s
: String
String object for "Java"
Contents cannot be changed
After executing s = "HTML";
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an ***interned string***.

```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

display
s1 == s2 is false
s1 == s3 is true


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

What is String? Explain about different types of String class methods.

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String

There are two ways to create string in Java:

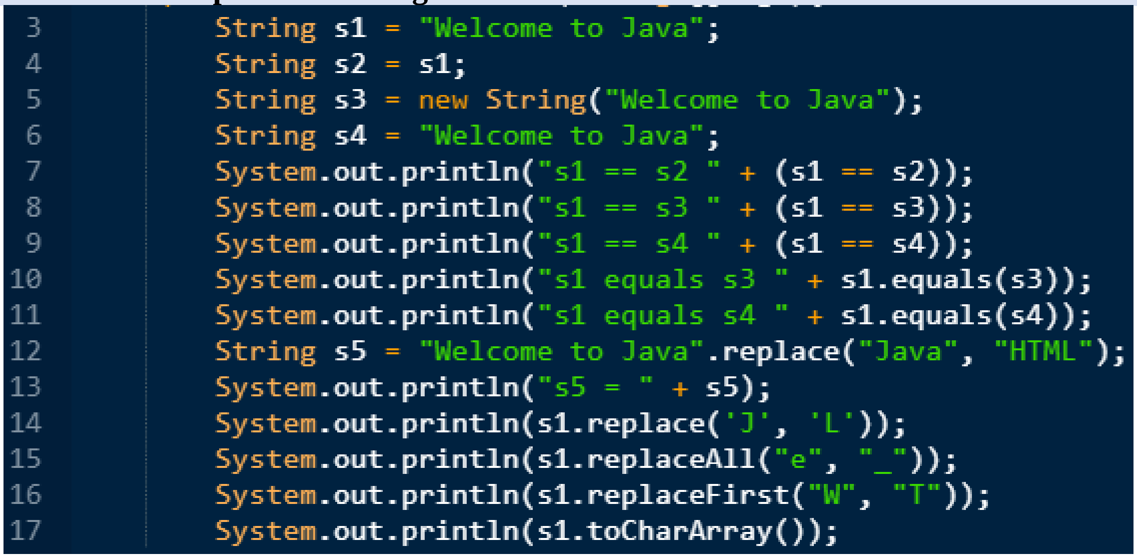
- ***String literal***
  String s = "New LJ";
- **Using *new* keyword**
  String s = new String ("New LJ");

Java String class provides a lot of methods to perform operations on strings such as:

Method	Description	Return Type
charAt(index)	Returns the character at the specified index (position)	char
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat(String str)	Appends a string to the end of another string	String
contains(String)	Checks whether a string contains a sequence of characters	boolean
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals(Object)	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
format(String format, Object... args**)**	Returns a formatted string using the specified locale, format string, and arguments	String
indexOf(int ch)	It returns the specified char value index.	int
indexOf(int ch, int fromIndex)	It returns the specified char value index starting with given index.	char
indexOf(String substring)	It returns the specified substring index.	String
indexOf(String substring, int fromIndex)	It returns the specified substring index starting with given index.	String
intern()	Returns the canonical representation for the string object	String
isEmpty()	Checks whether a string is empty or not	boolean

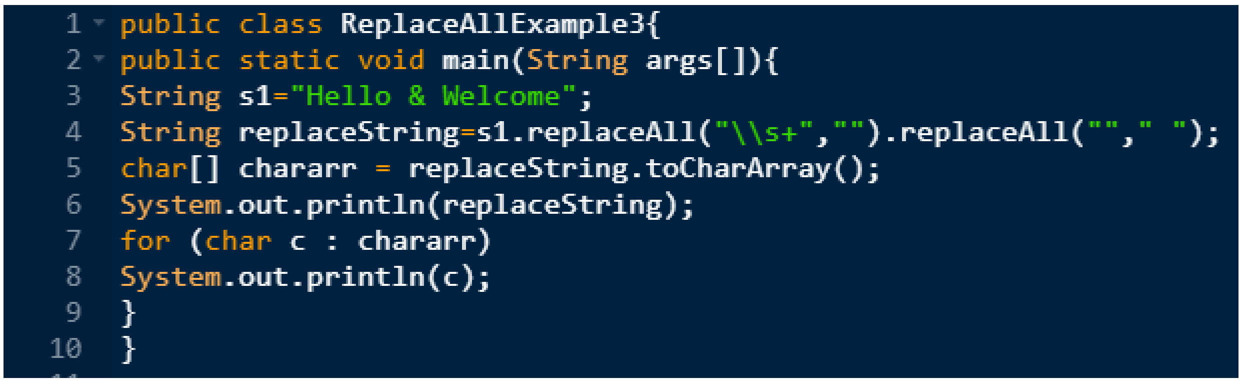
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
matches()	Searches a string for a match against a regular expression, and returns the matches	boolean
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replace(char old, char new)	It replaces all occurrences of the specified char value.	String
replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
startsWith()	Checks whether a string starts with specified characters	boolean
substring(int beginIndex):	It returns substring for given begin index.	String
substring(int beginIndex, int endIndex	It returns substring for given begin index and end index.	String
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified primitive value	String

What is the output of following statements?



```java
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
System.out.println("s1 == s2 " + (s1 == s2));
System.out.println("s1 == s3 " + (s1 == s3));
System.out.println("s1 == s4 " + (s1 == s4));
System.out.println("s1 equals s3 " + s1.equals(s3));
System.out.println("s1 equals s4 " + s1.equals(s4));
String s5 = "Welcome to Java".replace("Java", "HTML");
System.out.println("s5 = " + s5);
System.out.println(s1.replace('J', 'L'));
System.out.println(s1.replaceAll("e", "_"));
System.out.println(s1.replaceFirst("W", "T"));
System.out.println(s1.toCharArray());
```

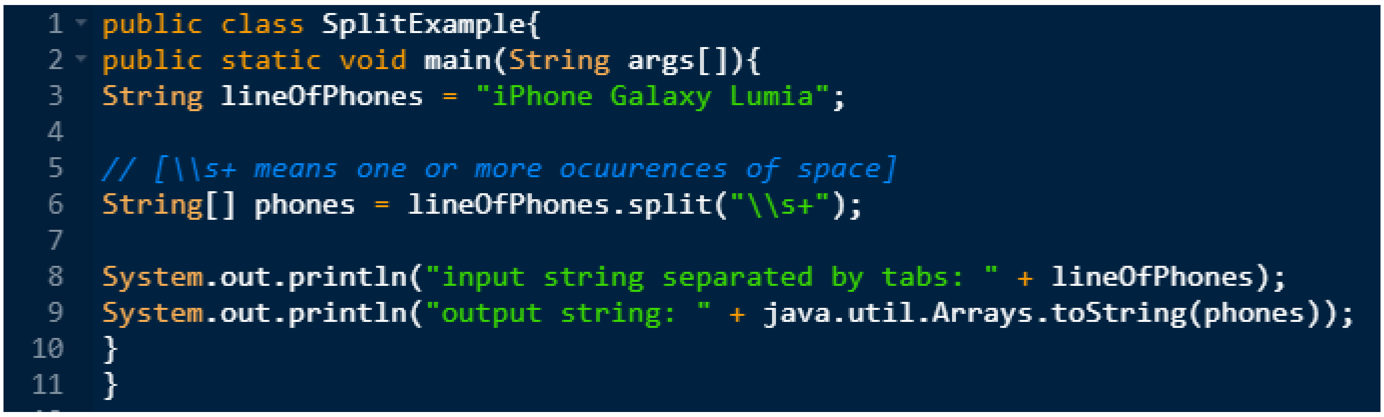
Program: Write a program to convert a string to character array using **replaceAll()** and **toCharArray()** method



```java
public class ReplaceAllExample3{
public static void main(String args[]){
String s1="Hello & Welcome";
String replaceString=s1.replaceAll("\\s+","").replaceAll(""," ");
char[] chararr = replaceString.toCharArray();
System.out.println(replaceString);
for (char c : chararr)
System.out.println(c);
}
}
```

```
Hello & Welcome
String converted to character array
  H   e   l   l   o   &   W   e   l   c   o   m   e
```

Program: Write a program to convert a space separated string to string array using **split()** method



```java
public class SplitExample{
public static void main(String args[]){
String lineOfPhones = "iPhone Galaxy Lumia";

// [\\s+ means one or more ocuurences of space]
String[] phones = lineOfPhones.split("\\s+");

System.out.println("input string separated by tabs: " + lineOfPhones);
System.out.println("output string: " + java.util.Arrays.toString(phones));
}
}
```

```
input string separated by tabs: iPhone Galaxy Lumia
output string: [iPhone, Galaxy, Lumia]
```

String Builder and String Buffer class OR Differentiate String class and StringBuffer class.

No.	String	String Buffer
1	The String class is immutable.	The StringBuffer class is mutable.
2	String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when we concatenate strings.
3	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.
4	String class is slower while performing concatenation operation.	StringBuffer class is faster while performing concatenation operation.

What is the difference between the StringBuffer and StringBuilder classes?

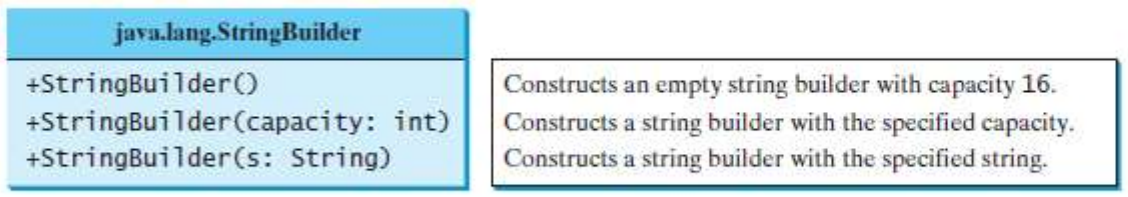
The **StringBuilder** *and* **StringBuffer** *classes are similar to the* **String** *class except that the* **String** *class is immutable.*

StringBuffer methods for modifying the buffer in **StringBuffer** are *synchronized*, which means that only one task is allowed to execute the methods. Use **StringBuffer** if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to **StringBuffer**.

Using **StringBuilder** is more efficient if it is accessed by just a single task, because no synchronization is needed in this case.

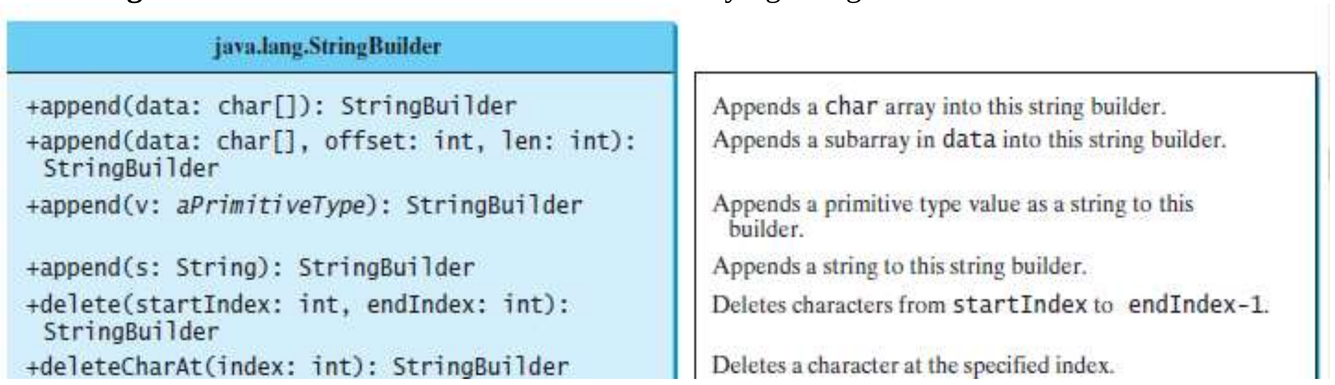
No.	**StringBuffer**	**StringBuilder**
1	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2	StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.

The **StringBuilder/ StringBuffer** class has three constructors and more than 30 similar methods for managing the builder/buffer and modifying strings in the builder/buffer.

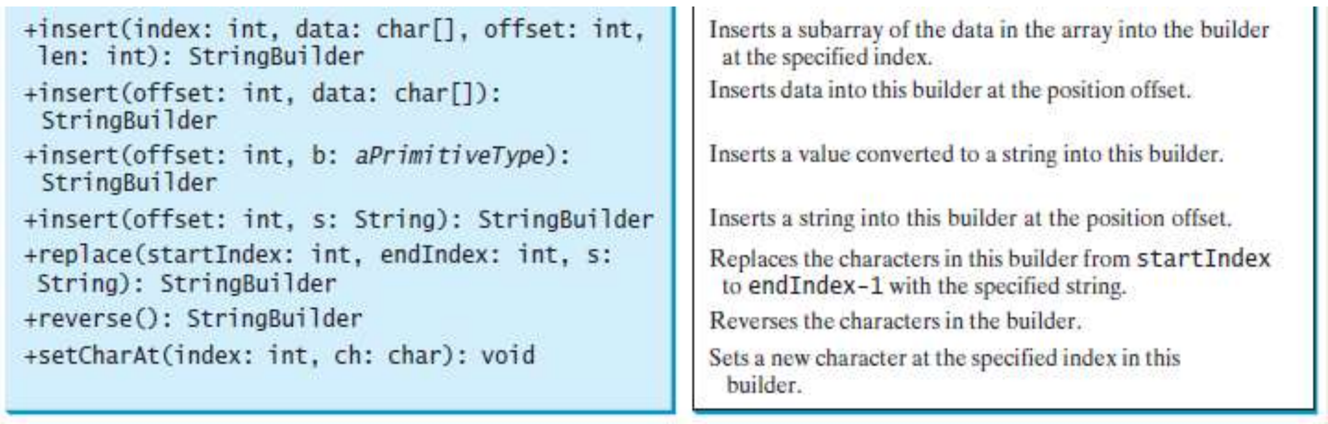


java.lang.StringBuilder	
`+StringBuilder()`	Constructs an empty string builder with capacity 16.
`+StringBuilder(capacity: int)`	Constructs a string builder with the specified capacity.
`+StringBuilder(s: String)`	Constructs a string builder with the specified string.

The **StringBuilder** class contains the methods for modifying string builders.

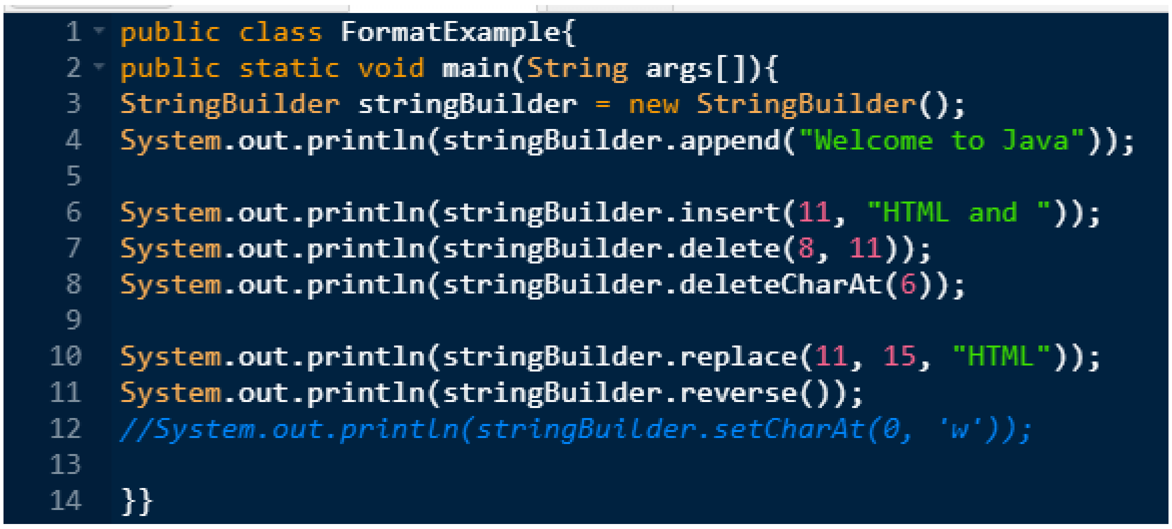


java.lang.StringBuilder	
`+append(data: char[]): StringBuilder`	Appends a `char` array into this string builder.
`+append(data: char[], offset: int, len: int): StringBuilder`	Appends a subarray in `data` into this string builder.
`+append(v: aPrimitiveType): StringBuilder`	Appends a primitive type value as a string to this builder.
`+append(s: String): StringBuilder`	Appends a string to this string builder.
`+delete(startIndex: int, endIndex: int): StringBuilder`	Deletes characters from `startIndex` to `endIndex-1`.
`+deleteCharAt(index: int): StringBuilder`	Deletes a character at the specified index.



`+insert(index: int, data: char[], offset: int, len: int): StringBuilder`	Inserts a subarray of the data in the array into the builder at the specified index.
`+insert(offset: int, data: char[]): StringBuilder`	Inserts data into this builder at the position offset.
`+insert(offset: int, b: aPrimitiveType): StringBuilder`	Inserts a value converted to a string into this builder.
`+insert(offset: int, s: String): StringBuilder`	Inserts a string into this builder at the position offset.
`+replace(startIndex: int, endIndex: int, s: String): StringBuilder`	Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string.
`+reverse(): StringBuilder`	Reverses the characters in the builder.
`+setCharAt(index: int, ch: char): void`	Sets a new character at the specified index in this builder.

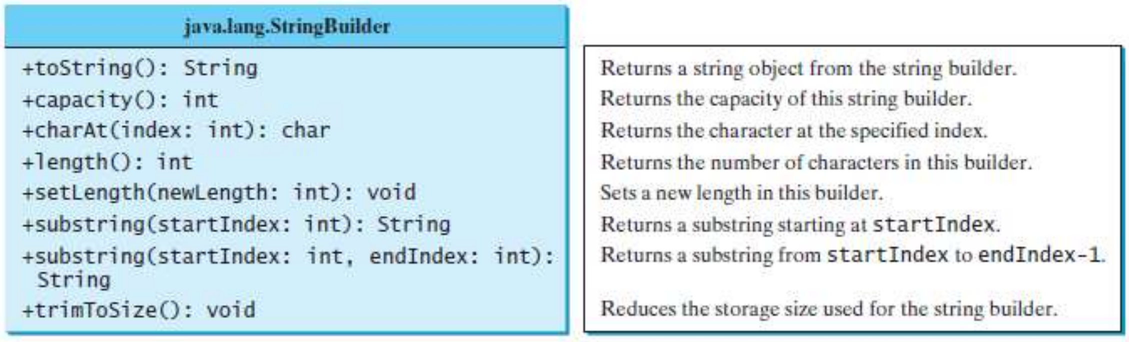
Program: Write a program to demonstrate methods of **StringBuilder/StringBuffer** class



```java
public class FormatExample{
public static void main(String args[]){
StringBuilder stringBuilder = new StringBuilder();
System.out.println(stringBuilder.append("Welcome to Java"));

System.out.println(stringBuilder.insert(11, "HTML and "));
System.out.println(stringBuilder.delete(8, 11));
System.out.println(stringBuilder.deleteCharAt(6));

System.out.println(stringBuilder.replace(11, 15, "HTML"));
System.out.println(stringBuilder.reverse());
//System.out.println(stringBuilder.setCharAt(0, 'w'));

}}
```

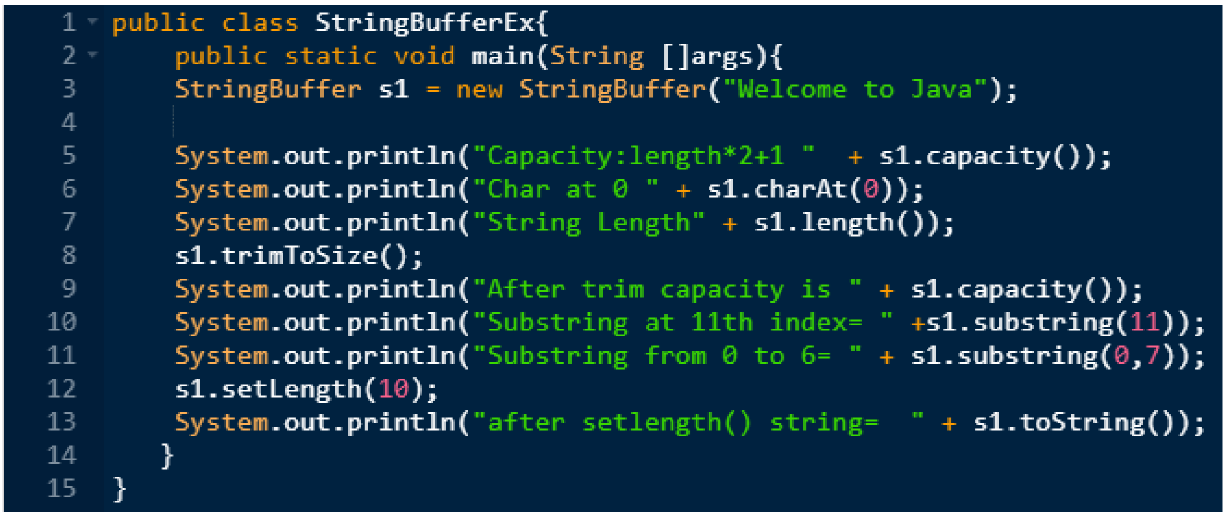
```
Welcome to Java
Welcome to HTML and Java
Welcome HTML and Java
Welcom HTML and Java
Welcom HTMLHTML Java
avaJ LMTHLMTH mocleW
```

The **StringBuilder** class provides the additional methods for manipulating a string builder and obtaining its properties.



java.lang.StringBuilder	
`+toString(): String`	Returns a string object from the string builder.
`+capacity(): int`	Returns the capacity of this string builder.
`+charAt(index: int): char`	Returns the character at the specified index.
`+length(): int`	Returns the number of characters in this builder.
`+setLength(newLength: int): void`	Sets a new length in this builder.
`+substring(startIndex: int): String`	Returns a substring starting at `startIndex`.
`+substring(startIndex: int, endIndex: int): String`	Returns a substring from `startIndex` to `endIndex-1`.
`+trimToSize(): void`	Reduces the storage size used for the string builder.

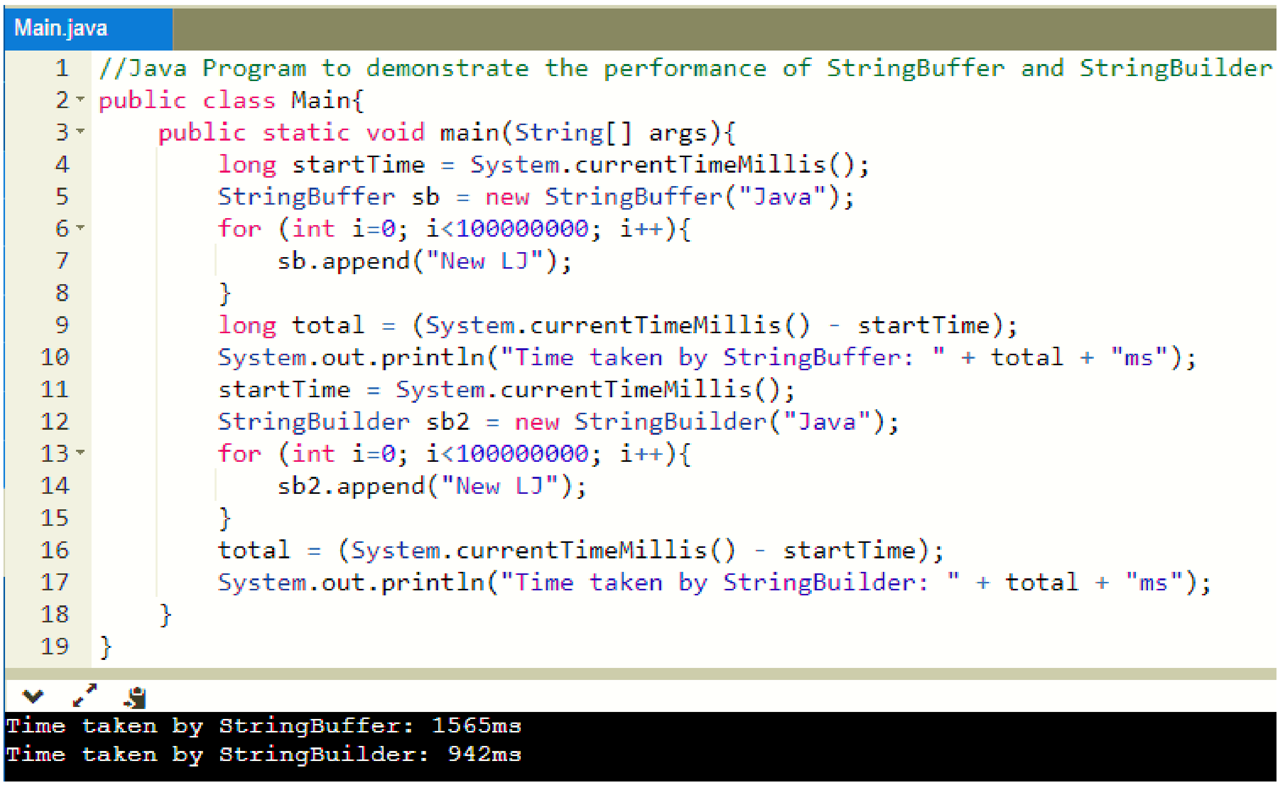
Program: Write a program to demonstrate methods of **StringBuilder/StringBuffer** class



```java
public class StringBufferEx{
    public static void main(String []args){
    StringBuffer s1 = new StringBuffer("Welcome to Java");

    System.out.println("Capacity:length*2+1 "  + s1.capacity());
    System.out.println("Char at 0 " + s1.charAt(0));
    System.out.println("String Length" + s1.length());
    s1.trimToSize();
    System.out.println("After trim capacity is " + s1.capacity());
    System.out.println("Substring at 11th index= " +s1.substring(11));
    System.out.println("Substring from 0 to 6= " + s1.substring(0,7));
    s1.setLength(10);
    System.out.println("after setlength() string=  " + s1.toString());
    }
}
```

```
Capacity:length*2+1 31
Char at 0 W
String Length15
After trim capacity is 15
Substring at 11th index= Java
Substring from 0 to 6= Welcome
after setlength() string=  Welcome to
```

Program: Write a program to demonstrate performance difference of **StringBuilder** and **StringBuffer** class



```java
//Java Program to demonstrate the performance of StringBuffer and StringBuilder
public class Main{
    public static void main(String[] args){
        long startTime = System.currentTimeMillis();
        StringBuffer sb = new StringBuffer("Java");
        for (int i=0; i<100000000; i++){
            sb.append("New LJ");
        }
        long total = (System.currentTimeMillis() - startTime);
        System.out.println("Time taken by StringBuffer: " + total + "ms");
        startTime = System.currentTimeMillis();
        StringBuilder sb2 = new StringBuilder("Java");
        for (int i=0; i<100000000; i++){
            sb2.append("New LJ");
        }
        total = (System.currentTimeMillis() - startTime);
        System.out.println("Time taken by StringBuilder: " + total + "ms");
    }
}
```

```
Time taken by StringBuffer: 1565ms
Time taken by StringBuilder: 942ms
```

5.6 super class and subclass & using super keyword (Inheritance)

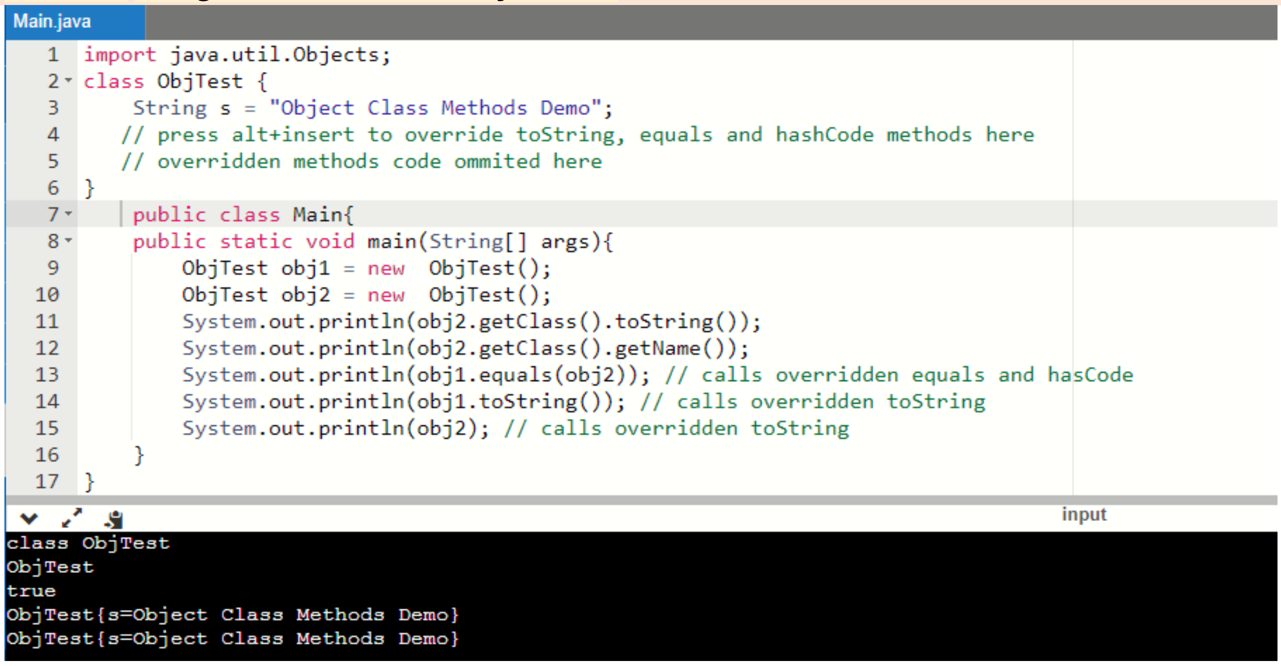
Which is the super class of every class in java? List out its important methods.

- The Object class is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- It is found in java.lang package.

The Object class provides many methods. They are as follows:

1. ***toString()*:** The toString() provides a String representation of an object and is used to convert an object to String.
2. ***hashCode():*** For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects.
3. ***equals(Object obj):*** It compares the given object to "this" object
4. ***getClass():***It returns the class object of "this" object and is used to get the actual runtime class of the object
5. ***finalize*()** method:This method is called just before an object is garbage collected.
6. ***clone():*** It returns a new object that is exactly the same as this object

EXTRA: Using inbuilt methods of ***Object*** class



Main.java

```java
import java.util.Objects;
class ObjTest {
    String s = "Object Class Methods Demo";
   // press alt+insert to override toString, equals and hashCode methods here
   // overridden methods code ommited here
}
    public class Main{
    public static void main(String[] args){
        ObjTest obj1 = new  ObjTest();
        ObjTest obj2 = new  ObjTest();
        System.out.println(obj2.getClass().toString());
        System.out.println(obj2.getClass().getName());
        System.out.println(obj1.equals(obj2)); // calls overridden equals and hasCode
        System.out.println(obj1.toString()); // calls overridden toString
        System.out.println(obj2); // calls overridden toString
    }
}
```

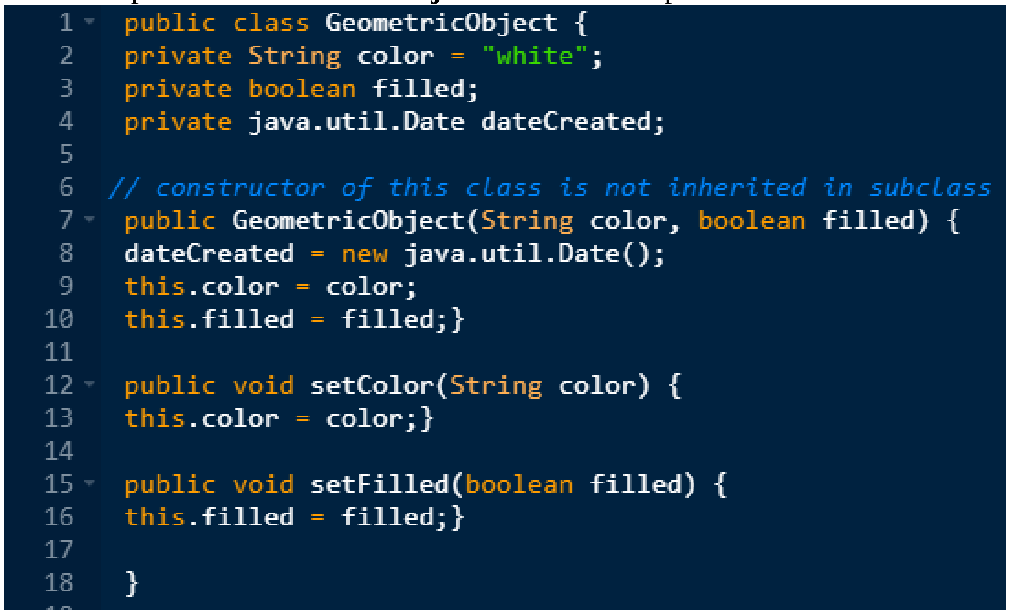
```
class ObjTest
ObjTest
true
ObjTest{s=Object Class Methods Demo}
ObjTest{s=Object Class Methods Demo}
```

Explain Inheritance with suitable example.

- Object-oriented programming allows you to define new classes from existing classes. This is called **inheritance**.
- Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., subclasses).
- In Java terminology, a class **C1** extended from another class **C2** is called a *subclass*, and **C2** is called a *superclass*.

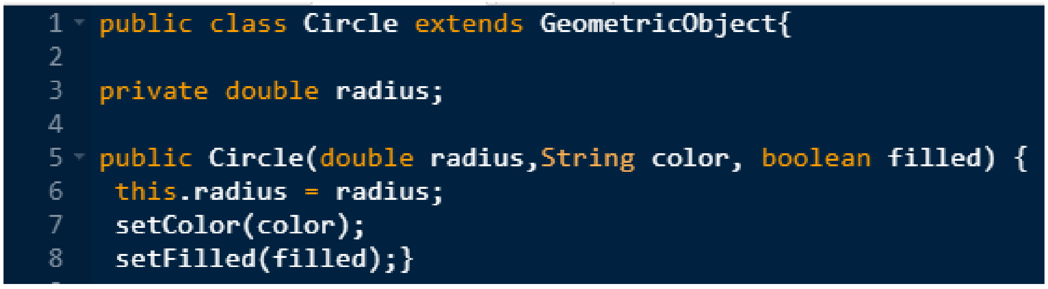
- A superclass is also referred to as a *parent class* or a *base class*, and a subclass as a *child class*, an *extended class*, or a *derived class*.
- A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.

For Example: The **GeometricObject** class is the superclass for **Circle** and **Rectangle**.



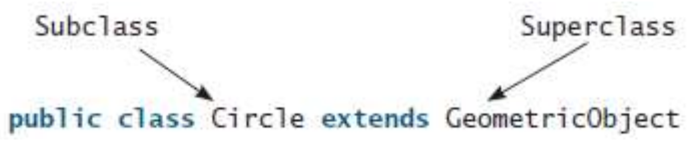
```java
public class GeometricObject {
private String color = "white";
private boolean filled;
private java.util.Date dateCreated;

// constructor of this class is not inherited in subclass
public GeometricObject(String color, boolean filled) {
dateCreated = new java.util.Date();
this.color = color;
this.filled = filled;}

public void setColor(String color) {
this.color = color;}

public void setFilled(boolean filled) {
this.filled = filled;}

}
```

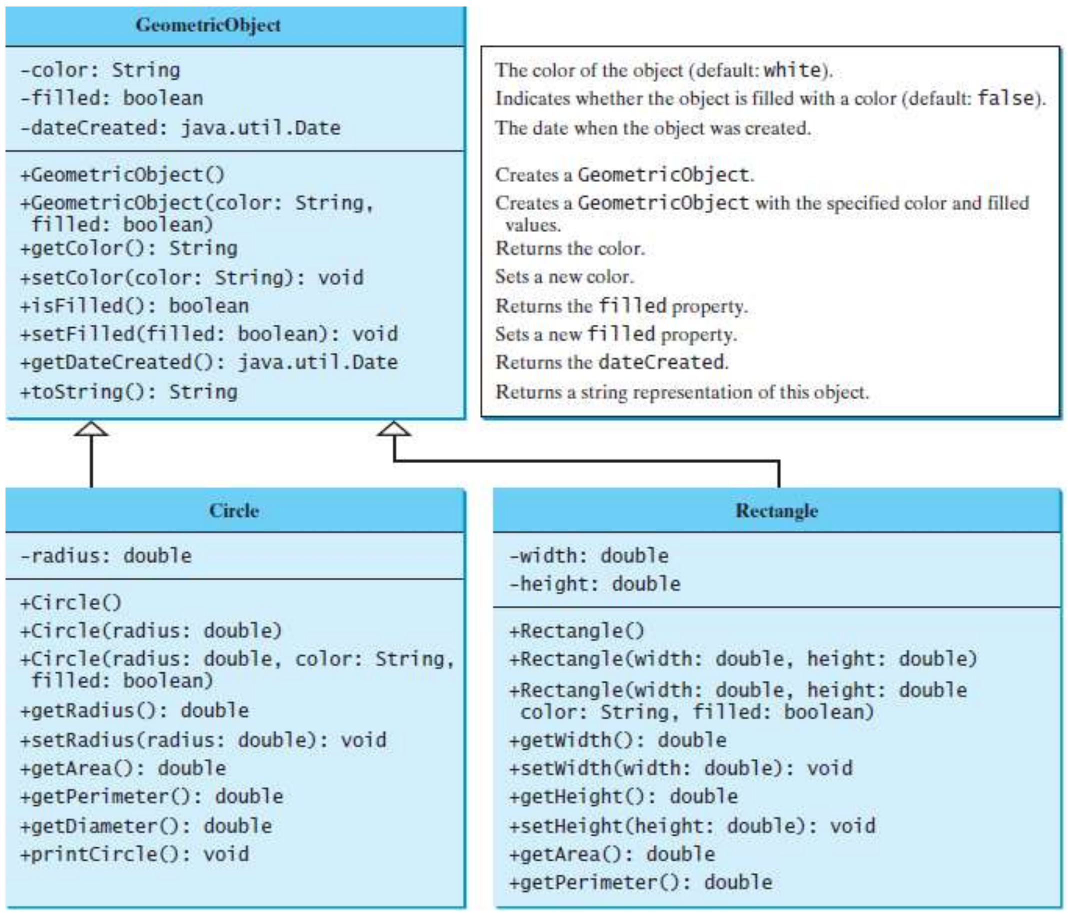
Here, subclass **Circle** can not access **color** and **filled** variables of superclass **GeometricObject** in its constructor**.** They can be accessed using getters/setters only, because constructors are not inherited in subclass.



```java
public class Circle extends GeometricObject{

private double radius;

public Circle(double radius,String color, boolean filled) {
 this.radius = radius;
 setColor(color);
 setFilled(filled);}
```

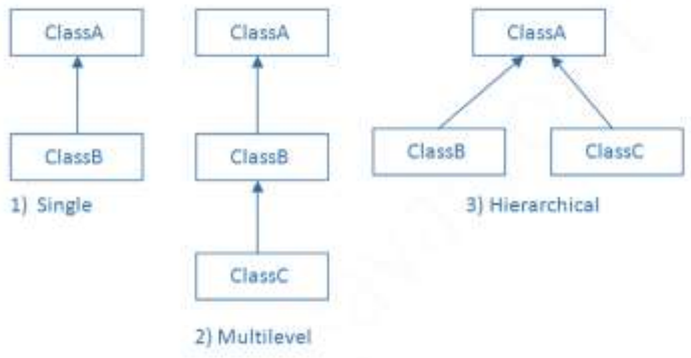
The **Circle** class extends the **GeometricObject** class using the following syntax:




GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

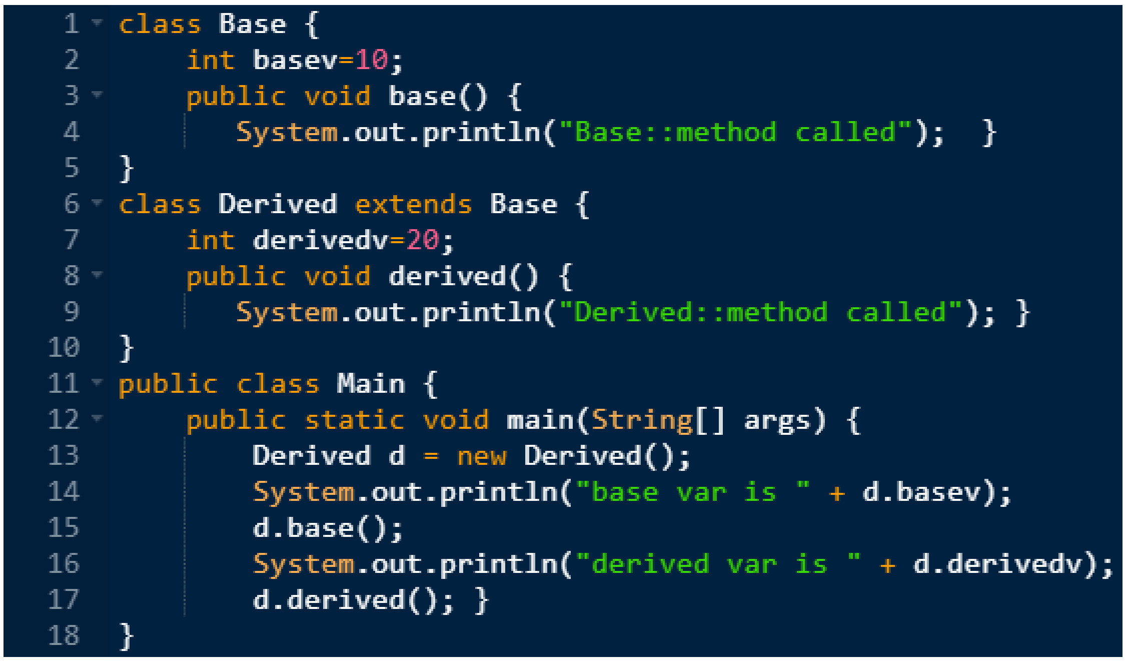
Describe Inheritance and its types with suitable example.

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



- When a class inherits another class, it is known as a *single inheritance*.
- When there is a chain of inheritance, it is known as *multilevel inheritance*.
- When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

1. **Single Inheritance**

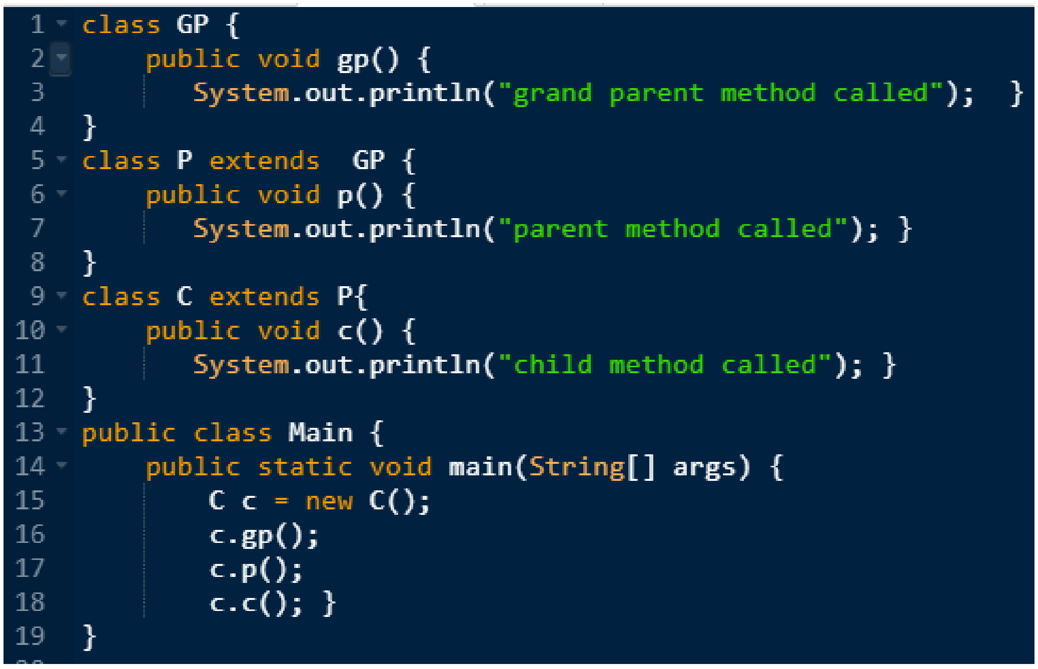


```java
class Base {
     int basev=10;
     public void base() {
        System.out.println("Base::method called");   }
}
class Derived extends Base {
     int derivedv=20;
     public void derived() {
        System.out.println("Derived::method called"); }
}
public class Main {
     public static void main(String[] args) {
         Derived d = new Derived();
         System.out.println("base var is " + d.basev);
         d.base();
         System.out.println("derived var is " + d.derivedv);
         d.derived(); }
}
```

```
Base var is 10
Base::method called
Derived var is 20
Derived::method called
```

NOTE:
Multiple inheritance is not supported in java through class but interface.
Why use inheritance in java:
- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

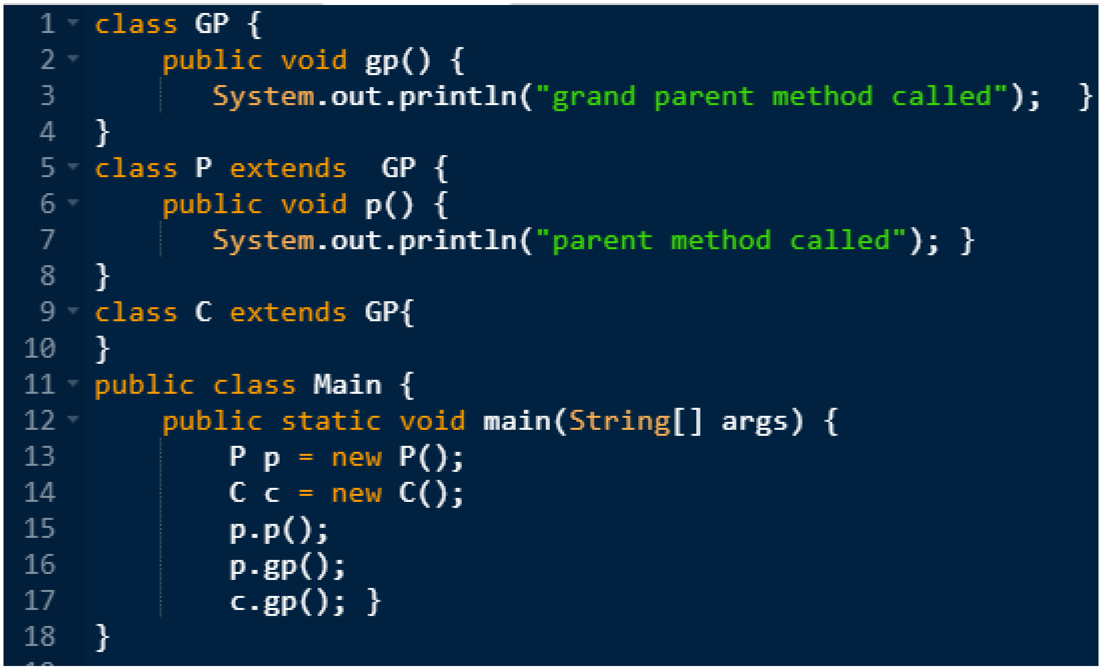
2. **Multilevel Inheritance**



```java
class GP {
     public void gp() {
        System.out.println("grand parent method called");   }
}
class P extends  GP {
     public void p() {
        System.out.println("parent method called"); }
}
class C extends P{
     public void c() {
        System.out.println("child method called"); }
}
public class Main {
     public static void main(String[] args) {
         C c = new C();
         c.gp();
         c.p();
         c.c(); }
}
```

```
grand parent method called
parent method called
child method called
```

3. Hierarchical Inheritance



```java
class GP {
    public void gp() {
       System.out.println("grand parent method called");  }
}
class P extends  GP {
    public void p() {
       System.out.println("parent method called"); }
}
class C extends GP{
}
public class Main {
    public static void main(String[] args) {
        P p = new P();
        C c = new C();
        p.p();
        p.gp();
        c.gp(); }
}
```

```
parent method called
grand parent method called
grand parent method called
```

❖ **Important facts about inheritance in Java**

1. **Default superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.

2. **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because **Java does not support** **multiple inheritance** with classes. Although with interfaces, multiple inheritance is supported by java.

3. **Inheriting Constructors:** A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

4. **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods(like getters and setters) for accessing its private fields, these can also be used by the subclass.

NOTE : What all can be done in a Subclass?
In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- We can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- We can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus overriding it (as in example above, *toString()* method is overridden).
- We can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- We can declare new methods in the subclass that are not in the superclass.
- We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

Explain Super keyword with the help of example.

The keyword **super** *refers to the superclass and can be used to invoke the superclass's methods and constructors.*

- The super keyword in Java is used to refer immediate parent class instance variable.
- *super* can be used to invoke immediate parent class method.
- *super()* can be used to invoke immediate parent class constructor.

1. super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Fruit{
      String color="yellow";
}
class Apple extends Fruit{
      String color="red";
      void printColor(){
      System.out.println(color);//prints color of Apple class
      System.out.println(super.color);//prints color of Fruit class
}
}
class TestSuper1{
      public static void main(String args[]){
            Apple a =new Apple();
            a.printColor();
      }
}
```

Output:

```
red
yellow
```

- In the above example, Fruit and Apple both classes have a common property color.
- If we print color property, it will print the color of current class by default.
- To access the parent property, we need to use super keyword.

2. **super can be used to invoke parent class method**

- This is used when we want to call parent class method.
- So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.

```
class Person{
    void message(){
        System.out.println("This is person class");
    }
}

class Student extends Person{
    void message(){
        System.out.println("This is student class");
    }
    void display(){
      message(); // will invoke or call current class message() method
      super.message(); // will invoke or call parent class message() method
    }
}
class TestSuper2{
    public static void main(String args[]){
         Student s = new Student();
         s.display();  // calling display() of Student
    }
}
```

Output:

```
This is student class
This is person class
```

- In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

3. **super is used to invoke parent class constructor.**
   super keyword can also be used to access the parent class constructor.

```
class Person{
    Person(){
        System.out.println("Person class Constructor");
    }
}

class Student extends Person{
    Student(){
        super();      // invoke or call parent class constructor
        System.out.println("Student class Constructor");
    }
}

class TestSuper3{
    public static void main(String[] args){
        Student s = new Student();
    }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

- In the above example we have called the superclass constructor using keyword ‘super’ via subclass constructor.
- Call to super() must be first statement in Derived(Student) Class constructor.

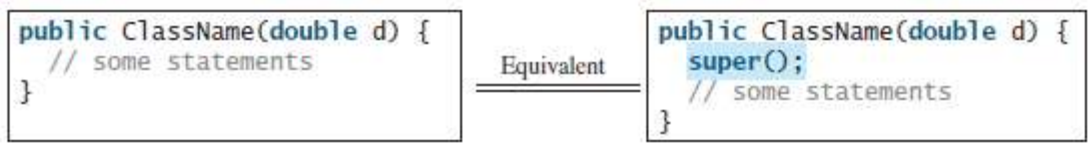
Explain Constructor Chaining

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.
Constructor chaining can be done in two ways:

- **Within same class**: It can be done using ***this()*** keyword for constructors in same class
- **From base class:** by using ***super()*** keyword to call constructor from the base class.

- A constructor may invoke an overloaded constructor or its superclass constructor.
- If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor.

For example:



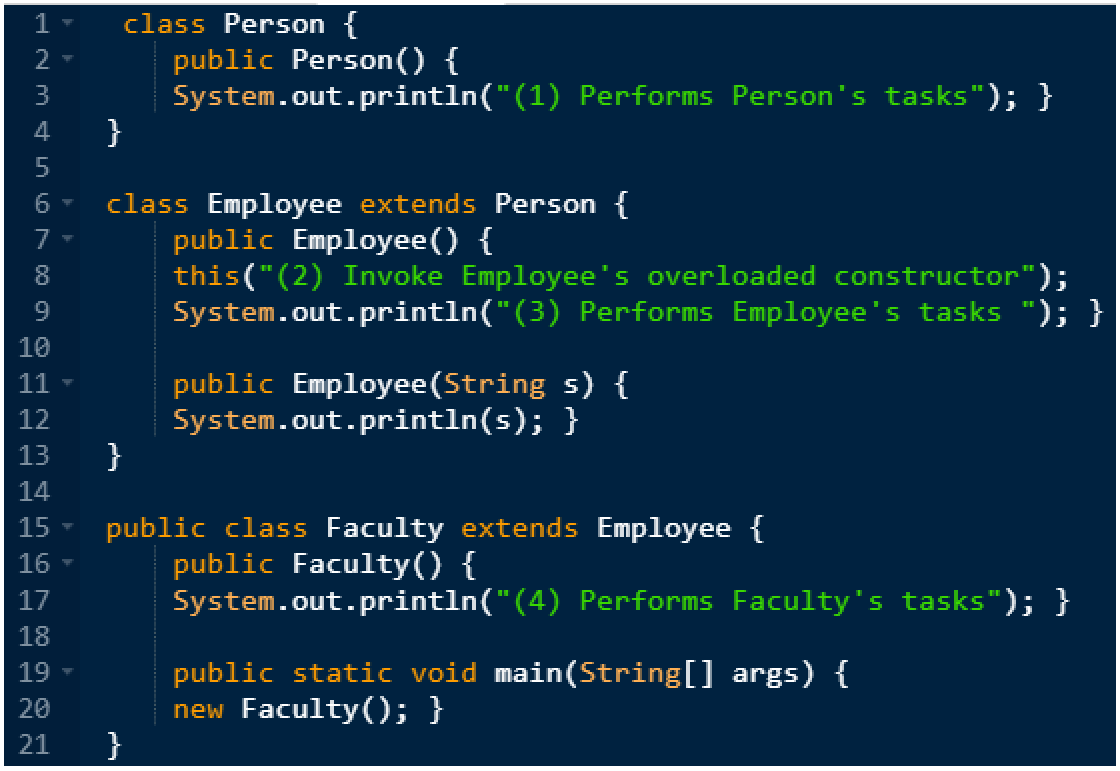
```
public ClassName(double d) {
  // some statements
}
```

```
public ClassName(double d) {
  super();
  // some statements
}
```

- When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks.
- If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks.
- This process continues until the last constructor along the inheritance hierarchy is called.
- This is called ***constructor chaining.***
- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain.

Why do we need constructor chaining ?
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

Program: Write a program to demonstrate constructor chaining



```java
 class Person {
    public Person() {
    System.out.println("(1) Performs Person's tasks"); }
}

class Employee extends Person {
    public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Performs Employee's tasks "); }

    public Employee(String s) {
    System.out.println(s); }
}

public class Faculty extends Employee {
    public Faculty() {
    System.out.println("(4) Performs Faculty's tasks"); }

    public static void main(String[] args) {
    new Faculty(); }
}
```

```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```

Caution
If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors.

5.7 overriding and overloading methods

Explain Method Overloading and Overriding.

Method Overloading

- Method Overloading is when a class has **many methods with the same name but different parameters.**
- If we have to perform only one operation, having same name of the methods increases the **readability** of the program
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs.
- As a result, we use method overloading to easily interpret the program.

Advantage of method overloading

- Method overloading increases the **readability** of the program.

Different ways to overload the method

- There are two ways to overload the method in java
    1. By changing number of arguments
    2. By changing the data type

1. Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

```
public class Adder {

    int add(int a,int b){
        return a+b;
    }
    int add(int a,int b,int c){
        return a+b+c;
    }
     public static void main(String[] args) {

        Adder a=new Adder();
        System.out.println(a.add(10, 10));
        System.out.println(a.add(10, 10, 10));
    }
}
```

Output:

```
20
30
```

2. Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type

```
public class Adder {

    int add(int a,int b){
        return a+b;
    }
    double add(double a,double b){
        return a+b;
    }
    public static void main(String[] args) {

        Adder a=new Adder();
        System.out.println(a.add(10, 10));
        System.out.println(a.add(10.2, 10.5));
    }
}
```

Output:

```
20
20.7
```

Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

- In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

Example of method overriding

```
class Parent {
    void show(){
        System.out.println("Parent's show()");
    }
}
class Child extends Parent {
 @Override
    void show(){
        System.out.println("Child's show()");
    }
}

class OverridingExample {
  public static void main(String[] args){
        Parent obj1 = new Parent();
        obj1.show();

        Parent obj2 = new Child();
        obj2.show();
        Child obj3 = new Child();
        Obj3.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
Child's show()
```

- The version of a method that is executed will be determined by the object that is used to invoke it.
- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

NOTE

Overriding

■ An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden.

■ Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax **SuperClassName.staticMethodName**.

Overloading means to define multiple methods with the same name but different signatures.
Overriding means to provide a new implementation for a method in the subclass.

■ Overridden methods are in different classes related by inheritance; overloaded methods can be either in the same class or different classes related by inheritance.

■ Overridden methods have the same signature and return type; overloaded methods have the same name but a different parameter list.

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place**@Override** before the method in the subclass.

Compare method overloading with method overriding.

- A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1.	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2.	Method overloading is performed within class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3.	In case of method Overloading, Parameter must be different.	In case of method overriding, parameter must be same.
4.	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
5.	In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding.

5.8 Polymorphism and dynamic binding

Polymorphism means that a variable of a supertype can refer to a subtype object.

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

A Type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism (Dynamic Binding/ Dynamic Method Dispatch)

1. **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

   Operator Overloading: Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands.

2. **Runtime polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

   Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

What is dynamic method dispatch? Explain with suitable example.

- It is also known as runtime polymorphism or dynamic method dispatch.
- We can achieve dynamic polymorphism by using the method overriding.
- In this process, an overridden method is called through a reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

Properties of Dynamic Polymorphism

- It decides which method is to execute at runtime.
- It can be achieved through dynamic binding.
- It happens between different classes.
- It is required where a subclass object is assigned to a super-class object for dynamic polymorphism.
- Inheritance involved in dynamic polymorphism.

Example of Dynamic Polymorphism

- We have assigned the child class object to the parent class reference.
- So, in order to determine which method would be called, the type of the object would be decided by the JVM at run-time.
- It is the type of object that determines which version of the method would be called (not the type of reference).

```
class Parent{
      public void display(){
            System.out.println("Overridden Method");}
}
public class Chlild extends Parent{
      public void display(){
            System.out.println("Overriding Method");
      }
      public static void main(String args[]){
      Parent p = new Child();
      p.display();
      }
}
```

Output:

```
Overriding Method
```

5.9 Casting objects and instanceof operator

One object reference can be typecast into another object reference. This is called casting object.

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.

Explain casting objects

- In java the object can also be typecasted like the datatypes.
- Parent and Child objects are two types of objects.
- So, there are two types of typecasting possible for an object, i.e., Parent to Child and Child to Parent or can say Upcasting and Downcasting.

Upcasting:

- Upcasting is a type of object typecasting in which a child object is typecasted to a parent class object.
- By using the Upcasting, we can easily access the variables and methods of the parent class to the child class.
- Here, we don't access all the variables and the method.
- We access only some specified variables and methods of the child class.
- Upcasting is also known as Generalization and Widening.

```
Parent obj1 = (Parent) new Child();
```

Downcasting

- Downasting is a type of object typecasting in which a parent object is typecasted to a child class object.
- Downcasting is used when we need to develop a code that accesses behaviors of the child class.

```
Parent p = new Child();
// Performing Downcasting Explicitly
        Child c = (Child)p;
```

Explain instanceof operator.

- The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).
- The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false.
- Let's see the simple example of instance operator where it tests the current class.

Example-1

```
class Parent{
 public static void main(String args[]){
 Parent p=new Parent();
 System.out.println(p instanceof Parent);//true
 }
}
```

Output:

```
true
```

Example-2
```
class Parent{}
class Child extends Parent{
 public static void main(String args[]){
     Child c=new Child();
     System.out.println(c instanceof Parent);//true
 }
}
```
Output:
```
true
```

Example-3
- instanceof in java with a variable that have null value
- If we apply instanceof operator with a variable that have null value, it returns false.
- Let's see the example given below where we apply instanceof operator with the variable that have null value.

```
class Child2{
 public static void main(String args[]){
  Child2 c=null;
  System.out.println(c instanceof Child2);//false
 }
}
```

Output:
```
false
```

EXTRA:
A parent reference referring to a Child is an instance of Child
```
Parent cobj = new Child();
System.out.print( (cobj instanceof Child) ); //true
```
Application of instanceof keyword
When we do typecast, it is always a good idea to check if the typecasting is valid or not. instanceof helps us here. We can always first check for validity using instancef, then do typecasting.
```
Parent par = cobj;
// Using instanceof to make sure that par is a valid reference before typecasting
if (par instanceof Child) {
  System.out.println("Value accessed through " + "parent reference with
typecasting is " +  ((Child)par).value);
}
```

5.10 The ArrayList class and its methods

Explain ArrayList and its methods.

- The `ArrayList` class is a resizable array, which can be found in the `java.util` package.
- The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one).

- While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different
- Java new **generic collection** allows you to have only one type of object in a collection. Now it is **type safe** so typecasting is not required at runtime.
- Let's see the old non-generic example of creating java collection.

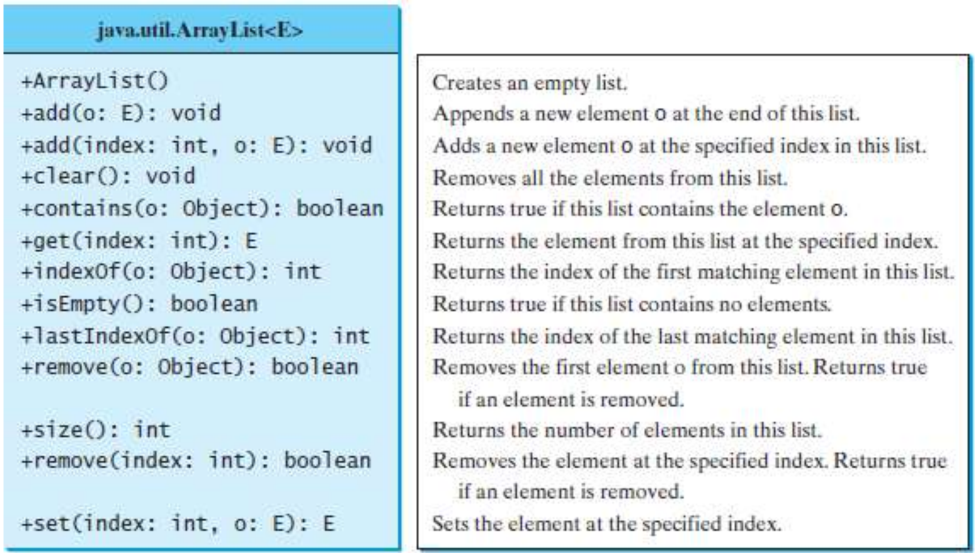
```
ArrayList al=new ArrayList();//creating old non-generic arraylist
```

- Let's see the new generic example of creating java collection.

```
ArrayList<String> al=new ArrayList<String>();//creating new generic arraylist
```

- In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

An **ArrayList** stores an unlimited number of objects.

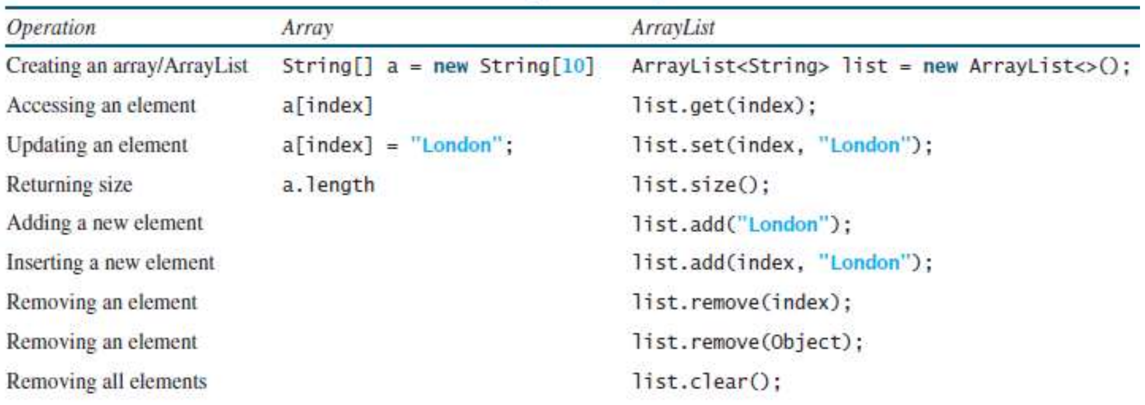


java.util.ArrayList<E>	
+ArrayList()	Creates an empty list.
+add(o: E): void	Appends a new element o at the end of this list.
+add(index: int, o: E): void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int): E	Returns the element from this list at the specified index.
+indexOf(o: Object): int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object): int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the first element o from this list. Returns true if an element is removed.
+size(): int	Returns the number of elements in this list.
+remove(index: int): boolean	Removes the element at the specified index. Returns true if an element is removed.
+set(index: int, o: E): E	Sets the element at the specified index.

- **ArrayList** is known as a generic class with a generic type **E**.
- You can specify a concrete type to replace **E** when creating an **ArrayList**.
- For example, the following statement creates an **ArrayList** and assigns its reference to variable **cities**.
- This **ArrayList** object can be used to store strings.

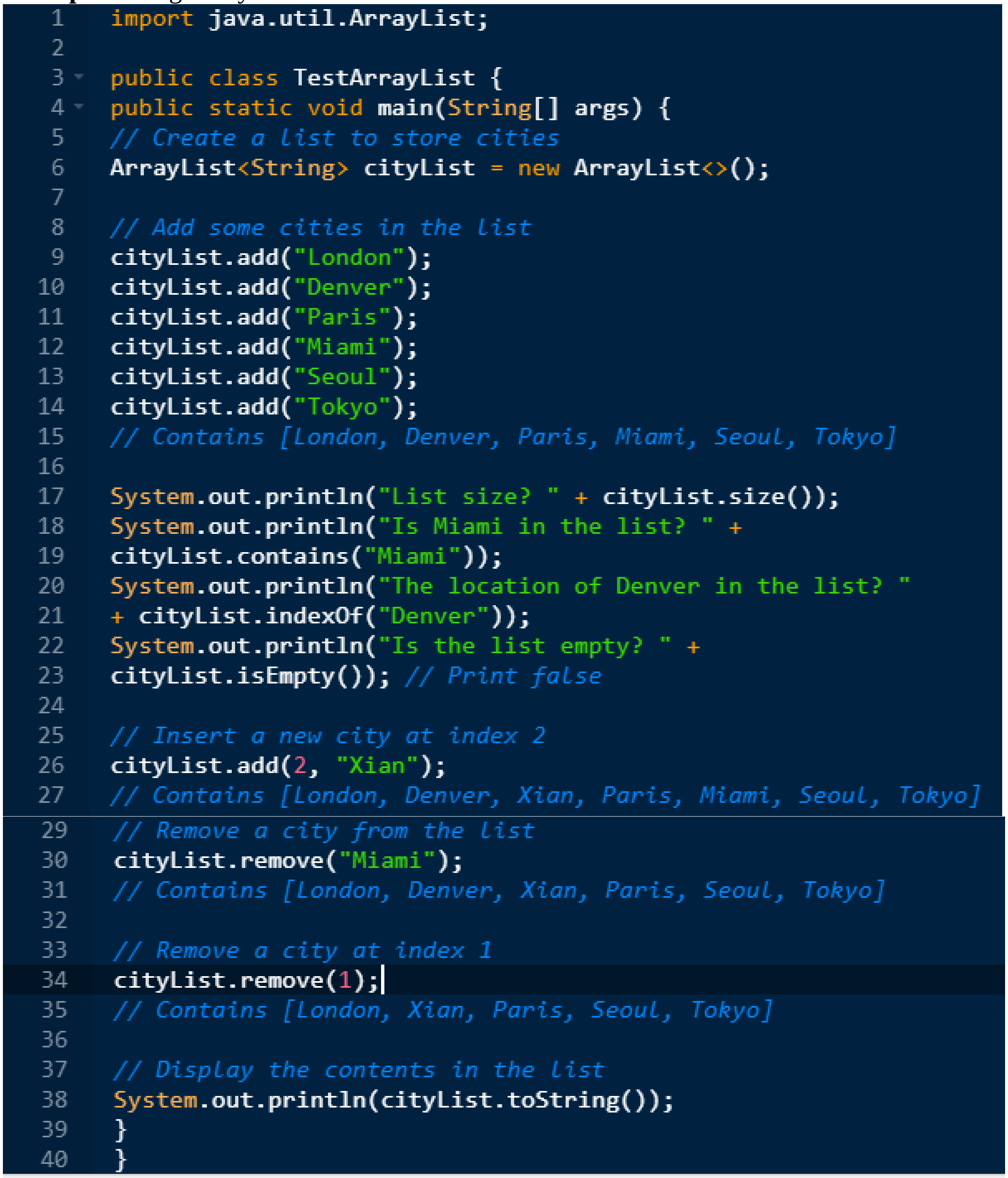
```
ArrayList<String> cities = new ArrayList<String>();
```

TABLE 11.1 Differences and Similarities between Arrays and ArrayList



Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

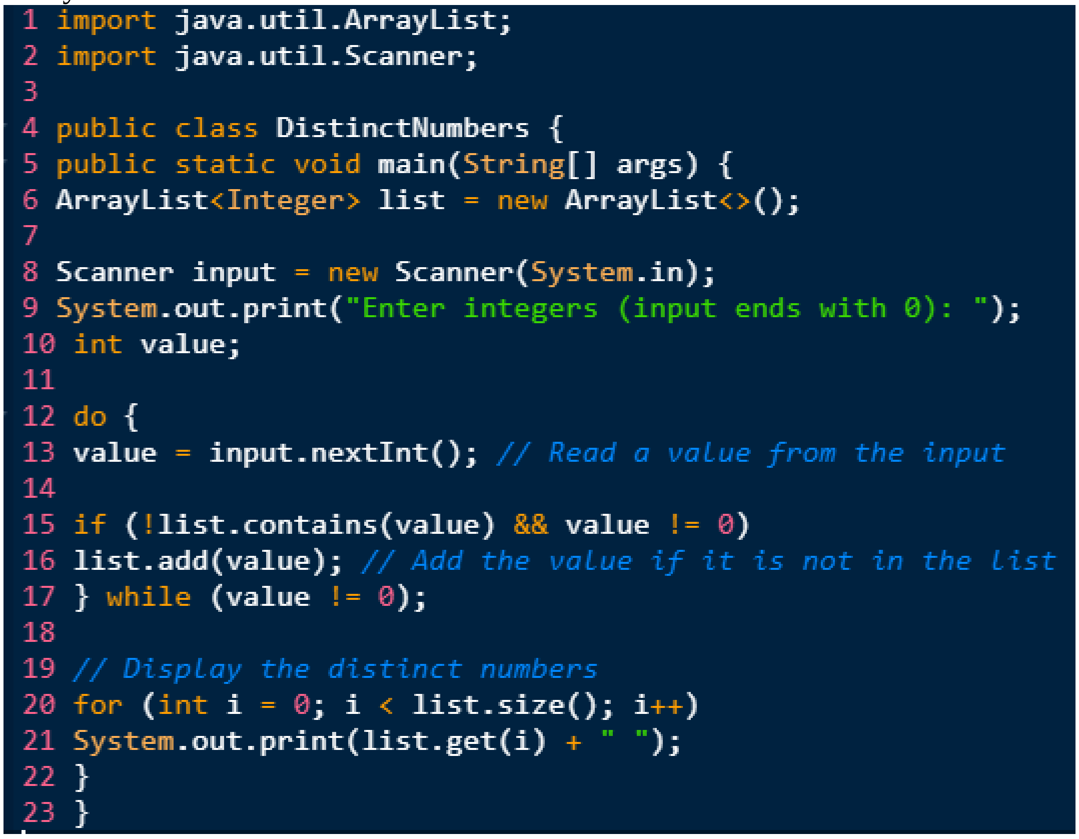
Example- Using ArrayList class methods



```java
import java.util.ArrayList;

public class TestArrayList {
public static void main(String[] args) {
// Create a list to store cities
ArrayList<String> cityList = new ArrayList<>();

// Add some cities in the list
cityList.add("London");
cityList.add("Denver");
cityList.add("Paris");
cityList.add("Miami");
cityList.add("Seoul");
cityList.add("Tokyo");
// Contains [London, Denver, Paris, Miami, Seoul, Tokyo]

System.out.println("List size? " + cityList.size());
System.out.println("Is Miami in the list? " +
cityList.contains("Miami"));
System.out.println("The location of Denver in the list? "
+ cityList.indexOf("Denver"));
System.out.println("Is the list empty? " +
cityList.isEmpty()); // Print false

// Insert a new city at index 2
cityList.add(2, "Xian");
// Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
// Remove a city from the list
cityList.remove("Miami");
// Contains [London, Denver, Xian, Paris, Seoul, Tokyo]

// Remove a city at index 1
cityList.remove(1);
// Contains [London, Xian, Paris, Seoul, Tokyo]

// Display the contents in the list
System.out.println(cityList.toString());
}
}
```

```
List size? 6
Is Miami in the list? true
The location of Denver in the list? 1
Is the list empty? false
[London, Xian, Paris, Seoul, Tokyo]
```

Program: Take numbers from user until 0 is entered by user and print only distinct numbers using ArrayList.



```java
import java.util.ArrayList;
import java.util.Scanner;

public class DistinctNumbers {
public static void main(String[] args) {
ArrayList<Integer> list = new ArrayList<>();

Scanner input = new Scanner(System.in);
System.out.print("Enter integers (input ends with 0): ");
int value;

do {
value = input.nextInt(); // Read a value from the input

if (!list.contains(value) && value != 0)
list.add(value); // Add the value if it is not in the list
} while (value != 0);

// Display the distinct numbers
for (int i = 0; i < list.size(); i++)
System.out.print(list.get(i) + " ");
}
}
```

```
Enter numbers (input ends with 0): 1 2 3 2 1 6 3 4 5 4 5 1 2 3 0
The distinct numbers are: 1 2 3 6 4 5
```

5.11 The protected data and methods.

The protected data and methods

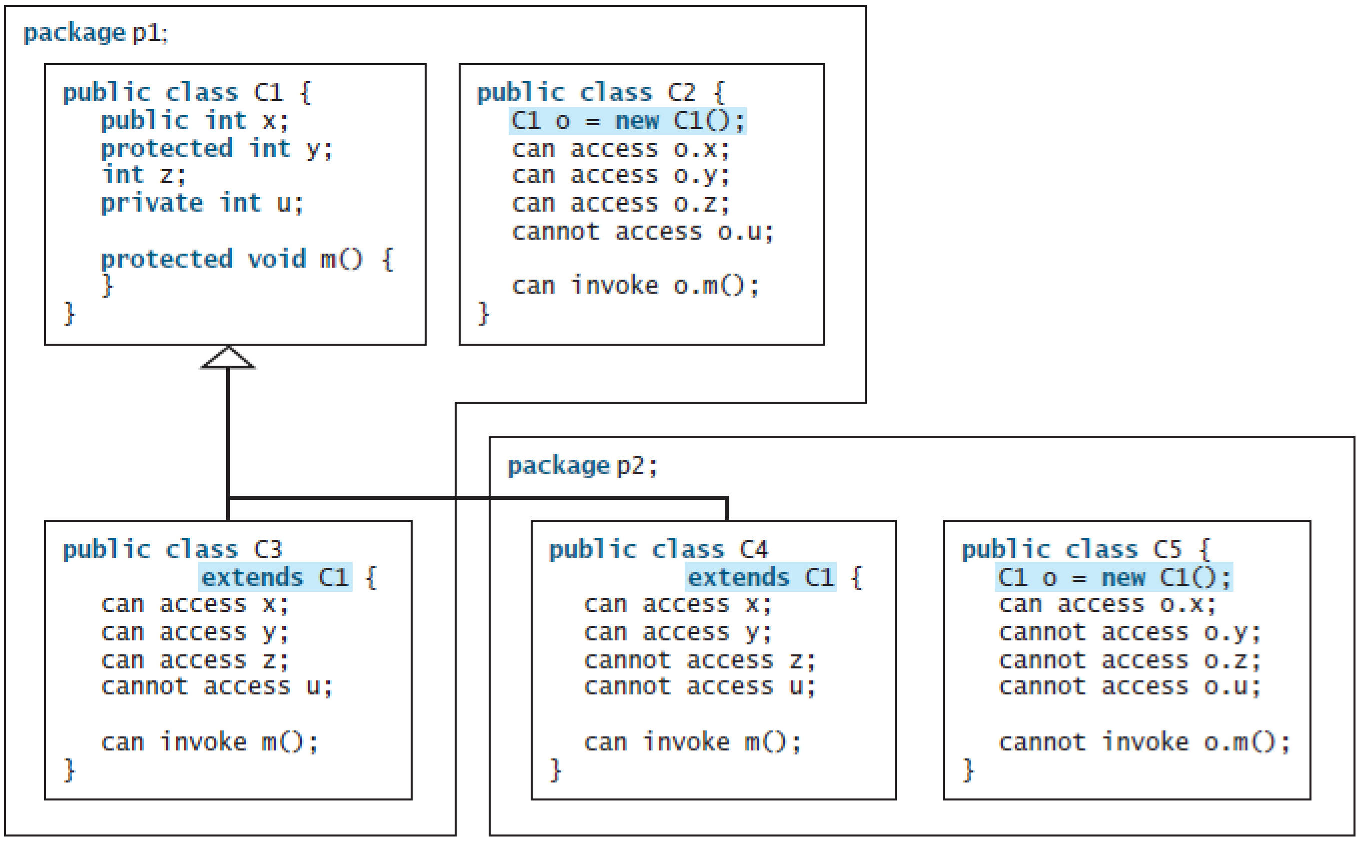
A protected member of a class can be accessed from a subclass.

- The **private** and **protected** modifiers can be used only for members of the class.
- The **public** modifier and the default modifier (i.e., no modifier) can be used on members of the class as well as on the class.
- A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.

NOTE
A subclass may override a protected method defined in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

Example: Testing all the cases of accessibility (visibility modifiers)
Package p1 => class C1, class C2, class C3 extends C1
Package p2 => class C4 extends C1, class C5

Protected members of C1 can be accessed by both subclasses C3 (same package) and C4 (different package)



```
public class C1 {
   public int x;
   protected int y;
   int z;
   private int u;

   protected void m() {
   }
}
```

```
public class C2 {
   C1 o = new C1();
   can access o.x;
   can access o.y;
   can access o.z;
   cannot access o.u;

   can invoke o.m();
}
```

```
public class C3
         extends C1 {
   can access x;
   can access y;
   can access z;
   cannot access u;

   can invoke m();
}
```

```
public class C4
         extends C1 {
   can access x;
   can access y;
   cannot access z;
   cannot access u;

   can invoke m();
}
```

```
public class C5 {
   C1 o = new C1();
   can access o.x;
   cannot access o.y;
   cannot access o.z;
   cannot access o.u;

   cannot invoke o.m();
}
```

❖ Preventing Extending and Overriding

Neither a final class nor a final method can be extended. A final data field is a constant.

You may occasionally want to prevent classes from being extended. In such cases, use the **final** modifier to indicate that a class is final and cannot be a parent class. The **Math** class is a final class. The **String**, **StringBuilder**, and **StringBuffer** classes are also final classes. For example, the following class **A** is final and cannot be extended:

```
public final class A {
// Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its subclasses. For example, the following method **m** is final and cannot be overridden:

```
public class Test {
// Data fields, constructors, and methods omitted
public final void m() {
// Do something
}
}
```

The modifiers **public**, **protected**, **private**, **static**, **abstract**, and **final** are used on classes and class members (data and methods), except that the **final** modifier can also be used on local variables in a method. A **final** local variable is a constant inside a method.