

Constructors

A *constructor* is a special method that has the *same method name as the class name*. That is, the constructor of the class Circle is called Circle(). In the above Circle class, we define three overloaded versions of constructor Circle(...). A constructor is used to *construct* and *initialize* all the member variables. To construct a new instance of a class, you need to use a special "new" operator followed by a call to one of the constructors. For example,

```
Circle c1 = new Circle();           // use 1st constructor
Circle c2 = new Circle(2.0);       // use 2nd constructor
Circle c3 = new Circle(3.0, "red"); // use 3rd constructor
```

A constructor method is different from an ordinary method in the following aspects:

- The *name* of the constructor method must be the same as the classname. By classname's convention, it begins with an uppercase (instead of lowercase for ordinary methods).
- Constructor has *no return type* in its method heading. It implicitly returns void. No return statement is allowed inside the constructor's body.
- Constructor can only be invoked via the "new" operator. It can only be used *once* to initialize the instance constructed. Once an instance is constructed, you cannot call the constructor anymore.
- Constructors are not inherited (to be explained later). Every class shall define its own constructors.

Default Constructor: A constructor with no parameter is called the *default constructor*. It initializes the member variables to their default values. For example, the Circle() in the above example initialize member variables radius and color to their default values.

2.10 Revisit Method Overloading

Method overloading means that the *same method name* can have *different implementations* (versions). However, the different implementations must be distinguishable by their parameter list (either the number of parameters, or the type of parameters, or their order).

Example: The method average() has 3 versions, with different parameter lists. The caller can invoke the chosen version by supplying the matching arguments.

```
1/**
2 * Example to illustrate "Method Overloading"
3 */
4public class MethodOverloadingTest {
5    public static int average(int n1, int n2) {           // version 1
6        System.out.println("Run version 1");
7        return (n1+n2)/2;
8    }
9    public static double average(double n1, double n2) { // version 2
10        System.out.println("Run version 2");
11        return (n1+n2)/2;
12    }
13    public static int average(int n1, int n2, int n3) {  // version 3
14        System.out.println("Run version 3");
15        return (n1+n2+n3)/3;
16    }
17
18    public static void main(String[] args) {
19        System.out.println(average(1, 2));
20        //Run version 1
21        //1
```

```

22  System.out.println(average(1.0, 2.0));
23  //Run version 2
24  //1.5
25  System.out.println(average(1, 2, 3));
26  //Run version 3
27  //2
28  System.out.println(average(1.0, 2));
29  //Run version 2 (int 2 implicitly casted to double 2.0)
30  //1.5
31
32  //average(1, 2, 3, 4);
33  //compilation error: no suitable method found for average(int,int,int,int)
34  }
35}

```

Overloading Circle Class' Constructor

Constructor, like an ordinary method, can also be overloaded. The above Circle class has three overloaded versions of constructors differentiated by their parameter list, as followed:

```

Circle() // the default constructor
Circle(double r)
Circle(double r, String c)

```

Depending on the actual argument list used when invoking the method, the matching constructor will be invoked. If your argument list does not match any one of the methods, you will get a compilation error.

Note: C language does not support method overloading. You need to use different method names for each of the variations. C++, Java, C# support method overloading.

2.11 The Access Control Modifiers: public/private

An *access control modifier* can be used to *control the visibility* of a class, or a member variable or a member method within a class. We begin with the following two access control modifiers:

1. **public:** The class/variable/method is accessible and available to ALL the other objects in the system.
2. **private:** The class/variable/method is accessible and available *within this class only*.

For example, in the above Circle definition, the member variable radius is declared private. As the result, radius is accessible inside the Circle class, but NOT in the TestCircle class. In other words, you cannot use "c1.radius" to refer to c1's radius in TestCircle.

- Try inserting the statement "System.out.println(c1.radius)" in TestCircle and observe the error message (error: radius has private access in Circle).
- Try changing radius to public in the Circle class, and re-run the above statement.

On the other hand, the method getRadius() is declared public in the Circle class. Hence, it can be invoked in the TestCircle class, e.g., c1.getRadius().

UML Notation: In UML class diagram, public members are denoted with a "+"; while private members with a "-".

More access control modifiers will be discussed later.

2.12 Information Hiding and Encapsulation

A class encapsulates the name, static attributes and dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelf for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.

Member variables of a class are typically hidden from the outside world (i.e., the other classes), with private access control modifier. Access to the member variables are provided via public accessor methods, e.g., `getRadius()` and `getColor()`.

This follows the principle of *information hiding*. That is, objects communicate with each others using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

Rule of Thumb: Do not make any variables public, unless you have a good reason.

2.13 The public Getters/Setters for private Variables

To allow other classes to *read* the value of a private variable say `xxx`, we provide a *get method* (or *getter* or *accessor method*) called `getXxx()`. A get method needs not expose the data in raw format. It can process the data and limit the view of the data others will see. The getters shall not modify the variable.

To allow other classes to *modify* the value of a private variable say `xxx`, we provide a *set method* (or *setter* or *mutator method*) called `setXxx()`. A set method could provide data validation (such as range checking), or transform the raw data into the internal representation.

For example, in our Circle class, the variables `radius` and `color` are declared private. That is to say, they are only accessible within the Circle class and not visible in any other classes, including the `TestCircle` class. You cannot access the private variables `radius` and `color` from the `TestCircle` class directly - via say `c1.radius` or `c1.color`. The Circle class provides two public accessor methods, namely, `getRadius()` and `getColor()`. These methods are declared public. The class `TestCircle` can invoke these public accessor methods to retrieve the `radius` and `color` of a Circle object, via say `c1.getRadius()` and `c1.getColor()`.

There is no way you can change the `radius` or `color` of a Circle object, after it is constructed in the `TestCircle` class. You cannot issue statements such as `c1.radius = 5.0` to change the `radius` of instance `c1`, as `radius` is declared as private in the Circle class and is not visible to other classes including `TestCircle`.

If the designer of the Circle class permits the change the `radius` and `color` after a Circle object is constructed, he has to provide the appropriate set methods (or setters or mutator methods), e.g.,

```
// Setter for color
public void setColor(String newColor) {
    color = newColor;
}

// Setter for radius
public void setRadius(double newRadius) {
    radius = newRadius;
}
```

With proper implementation of *information hiding*, the designer of a class has full control of what the user of the class can and cannot do.

2.14 Keyword "this"

You can use keyword "this" to refer to *this* instance inside a class definition.

One of the main usage of keyword `this` is to resolve ambiguity.

```
public class Circle {
    double radius;           // member variable called "radius"
    public Circle(double radius) { // method's parameter also called "radius"
        this.radius = radius;
        // "radius = radius" does not make sense!
        // "this.radius" refers to this instance's member variable
        // "radius" resolved to the method's parameter.
    }
    ...
}
```

```
}
```

In the above codes, there are two identifiers called radius - a member variable of the class and the method's parameter. This causes naming conflict. To avoid the naming conflict, you could name the method's argument *r* instead of radius. However, radius is more approximate and meaningful in this context. Java provides a keyword called *this* to resolve this naming conflict. "*this.radius*" refers to the member variable; while "*radius*" resolves to the method's argument.

Using the keyword "*this*", the constructor, getter and setter methods for a private variable called *xxx* of type *T* are as follows:

```
public class Ccc {
    // A private variable named xxx of the type T
    private T xxx;

    // Constructor
    public Ccc(T xxx) {
        this.xxx = xxx;
    }

    // A getter for variable xxx of type T receives no argument and return a value of type T
    public T getXxx() {
        return xxx; // or "return this.xxx" for clarity
    }

    // A setter for variable xxx of type T receives a parameter of type T and return void
    public void setXxx(T xxx) {
        this.xxx = xxx;
    }
}
```

For a boolean variable *xxx*, the getter shall be named *isXxx()* or *hasXxx()*, which is more meaningful than *getXxx()*. The setter remains as *setXxx()*.

```
// private boolean variable
private boolean xxx;

// getter
public boolean isXxx() {
    return xxx; // or "return this.xxx" for clarity
}

// setter
public void setXxx(boolean xxx) {
    this.xxx = xxx;
}
```

More on "*this*"

- *this.varName* refers to *varName* of *this* instance; *this.methodName(...)* invokes *methodName(...)* of *this* instance.
- In a constructor, we can use *this(...)* to call *another* constructor of *this* class.
- Inside a method, we can use the statement "*return this*" to return *this* instance to the caller.

2.15 Method toString()

Every well-designed Java class shall have a public method called *toString()* that returns a string description of *this* instance. You can invoke the *toString()* method explicitly by calling *anInstanceName.toString()*, or implicitly via *println()* or String concatenation operator '+'. That is, running *println(anInstance)* invokes the *toString()* method of that instance implicitly.

For example, include the following toString() method in our Circle class:

```
// Return a String description of this instance
public String toString() {
    return "Circle[radius=" + radius + ",color=" + color + "]";
}
```

In your TestCircle class, you can get a description of a Circle instance via:

```
Circle c4 = new Circle();
System.out.println(c4.toString()); // Explicitly calling toString()
//Circle[radius=1.0,color=red]
System.out.println(c4);           // Implicit call to c4.toString()
//Circle[radius=1.0,color=red]
System.out.println("c4 is: " + c4); // '+' invokes toString() to get a String before concatenation
//Circle[radius=1.0,color=red]
```

The signature of toString() is:

```
public String toString() { ..... }
```

2.16 Constants (final)

Constants are variables defined with the modifier final. A final variable can only be assigned once and its value cannot be modified once assigned. For example,

```
public final double X_REFERENCE = 1.234;
```

```
private final int MAX_ID = 9999;
```

```
MAX_ID = 10000; //compilation error: cannot assign a value to final variable MAX_ID
```

```
// You need to initialize a final member variable during declaration
```

```
private final int SIZE; //compilation error: variable SIZE might not have been initialized
```

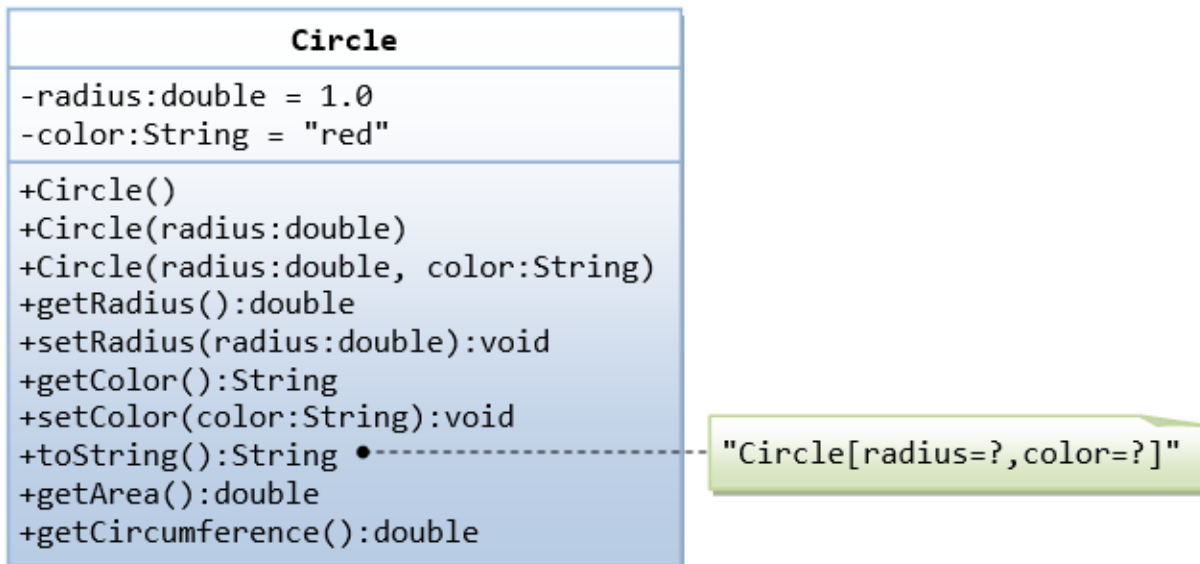
Constant Naming Convention: A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '_', for examples, X_REFERENCE, MAX_INTEGER and MIN_VALUE.

Advanced notes for final:

1. A final primitive variable cannot be re-assigned a new value.
2. A final instance cannot be re-assigned a new object.
3. A final class cannot be sub-classed (or extended).
4. A final method cannot be overridden.

2.17 Putting Them Together in the Finalized Circle Class

We shall include constructors, getters, setters, toString(), and use the keyword "this". The class diagram for the final Circle class is as follows:



Circle.java

```

1/**
2 * The Circle class models a circle with a radius and color.
3 */
4public class Circle { // Save as "Circle.java"
5 // The public constants
6 public static final double DEFAULT_RADIUS = 1.0;
7 public static final String DEFAULT_COLOR = "red";
8
9 // The private instance variables
10 private double radius;
11 private String color;
12
13 // The (overloaded) constructors
14 /** Constructs a Circle with default radius and color */
15 public Circle() { // 1st (default) Constructor
16     this.radius = DEFAULT_RADIUS;
17     this.color = DEFAULT_COLOR;
18 }
19 /** Constructs a Circle with the given radius and default color */
20 public Circle(double radius) { // 2nd Constructor
21     this.radius = radius;
22     this.color = DEFAULT_COLOR;
23 }
24 /** Constructs a Circle with the given radius and color */
25 public Circle(double radius, String color) { // 3rd Constructor
26     this.radius = radius;
27     this.color = color;
28 }
29
30 /** Returns the radius - the public getter for private variable radius. */
31 public double getRadius() {
32     return this.radius;
33 }

```

```
34 /** Sets the radius - the public setter for private variable radius */
35 public void setRadius(double radius) {
36     this.radius = radius;
37 }
38 /** Returns the color - the public getter for private variable color */
39 public String getColor() {
40     return this.color;
41 }
42 /** Sets the color - the public setter for private variable color */
43 public void setColor(String color) {
44     this.color = color;
45 }
46
47 /** Returns a self-descriptive string for this Circle instance */
48 public String toString() {
49     return "Circle[radius=" + radius + ", color=" + color + "]";
50 }
51
52 /** Returns the area of this Circle */
53 public double getArea() {
54     return radius * radius * Math.PI;
55 }
56
57 /** Returns the circumference of this Circle */
58 public double getCircumference() {
59     return 2.0 * radius * Math.PI;
60 }
61 }
```

A Test Driver for the Circle Class (TestCircle.java)

```
/**
 * A Test Driver for the Circle class
 */
public class TestCircle {
    public static void main(String[] args) {
        // Test all constructors and toString()
        Circle c1 = new Circle(1.1, "blue");
        System.out.println(c1); // implicitly run toString()
        //Circle[radius=1.1, color=blue]
        Circle c2 = new Circle(2.2);
        System.out.println(c2);
        //Circle[radius=2.2, color=red]
        Circle c3 = new Circle();
        System.out.println(c3);
        //Circle[radius=1.0, color=red]

        // Test Setters and Getters
        c1.setRadius(3.3);
        c1.setColor("green");
        System.out.println(c1); // use toString() to inspect the modified instance
        //Circle[radius=3.3, color=green]
        System.out.println("The radius is: " + c1.getRadius());
        //The radius is: 3.3
    }
}
```



```
System.out.println("The color is: " + c1.getColor());
//The color is: green

// Test getArea() and getCircumference()
System.out.printf("The area is: %.2f%n", c1.getArea());
//The area is: 34.21
System.out.printf("The circumference is: %.2f%n", c1.getCircumference());
//The circumference is: 20.73
}
```

Autoboxing and Unboxing

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes. For example, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called *unboxing*.

Here is the simplest example of autoboxing:

```
Character ch = 'a';
```

The rest of the examples in this section use generics. If you are not yet familiar with the syntax of generics, see the [Generics \(Updated\)](#) lesson.

Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

Although you add the int values as primitive types, rather than Integer objects, to li, the code compiles. Because li is a list of Integer objects, not a list of int values, you may wonder why the Java compiler does not issue a compile-time error. The compiler does not generate an error because it creates an Integer object from i and adds the object to li. Thus, the compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

Converting a primitive value (an int, for example) into an object of the corresponding wrapper class (Integer) is called autoboxing. The Java compiler applies autoboxing when a primitive value is:

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

Consider the following method:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i: li)
        if (i % 2 == 0)
```



```
        sum += i;
    return sum;
}
```

Because the remainder (%) and unary plus (+) operators do not apply to Integer objects, you may wonder why the Java compiler compiles the method without issuing any errors. The compiler does not generate an error because it invokes the intValue method to convert an Integer to an int at runtime:

```
public static int sumEven(List<Integer> li) {
    int sum = 0;
    for (Integer i : li)
        if (i.intValue() % 2 == 0)
            sum += i.intValue();
    return sum;
}
```

Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value is called unboxing. The Java compiler applies unboxing when an object of a wrapper class is:

- Passed as a parameter to a method that expects a value of the corresponding primitive type.
- Assigned to a variable of the corresponding primitive type.

The [Unboxing](#) example shows how this works:

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

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416); // Pi is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

The program prints the following:

```
absolute value of -8 = 8
pi = 3.1416
```

Autoboxing and unboxing lets developers write cleaner code, making it easier to read. The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:

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

few advantages of autoboxing and unboxing in order to get why we are using it.

- Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
- The technique lets us use primitive types and Wrapper class objects interchangeably and we do not need to perform any typecasting explicitly.

Example 1:

- Java

```
// Java program to illustrate the Concept  
// of Autoboxing and Unboxing
```

```
// Importing required classes  
import java.io.*;
```

```
// Main class  
class GFG {
```

```
    // Main driver method  
    public static void main(String[] args)  
    {
```

```
        // Creating an Integer Object  
        // with custom value say it be 10  
        Integer i = new Integer(10);
```

```
        // Unboxing the Object
```

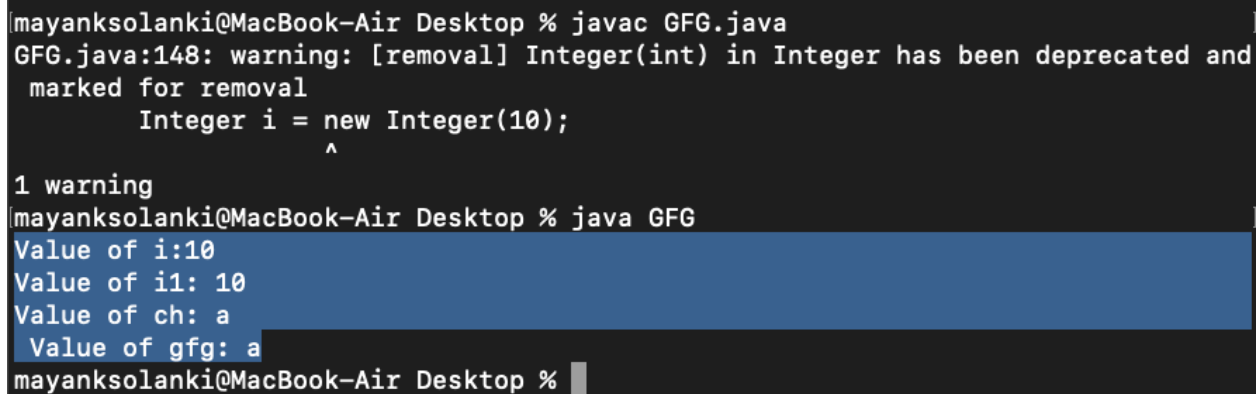
```
int i1 = i;

// Print statements
System.out.println("Value of i:" + i);
System.out.println("Value of i1: " + i1);

// Autoboxing of character
Character gfg = 'a';

// Auto-unboxing of Character
char ch = gfg;

// Print statements
System.out.println("Value of ch: " + ch);
System.out.println(" Value of gfg: " + gfg);
}
}
```

Output:

```
mayanksolanki@MacBook-Air Desktop % javac GFG.java
GFG.java:148:warning: [removal] Integer(int) in Integer has been deprecated and
marked for removal
    Integer i = new Integer(10);
                   ^
1 warning
mayanksolanki@MacBook-Air Desktop % java GFG
Value of i:10
Value of i1: 10
Value of ch: a
Value of gfg: a
mayanksolanki@MacBook-Air Desktop %
```

Let's understand how the compiler did autoboxing and unboxing in the example of Collections in Java using [generics](#).

Example 2:

- Java

// Java Program to Illustrate Autoboxing

// Importing required classes

```
import java.io.*;
import java.util.*;
```

// Main class

```
class GFG {
```

// Main driver method

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

// Creating an empty ArrayList of integer type

```
ArrayList<Integer> al = new ArrayList<Integer>();

// Adding the int primitives type values
// using add() method
// Autoboxing
al.add(1);
al.add(2);
al.add(24);

// Printing the ArrayList elements
System.out.println("ArrayList: " + al);
}
```

Output

ArrayList: [1, 2, 24]

Output explanation:

In the above example, we have created a list of elements of the Integer type. We are adding int primitive type values instead of Integer Object and the code is successfully compiled. It does not generate a compile-time error as the Java compiler creates an Integer wrapper Object from primitive int i and adds it to the list.

Example 3:

- Java

// Java Program to Illustrate Autoboxing

// Importing required classes

```
import java.io.*;
import java.util.*;
```

// Main class

```
class GFG {
```

// Main driver method

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

```
// Creating an empty ArrayList of integer type
List<Integer> list = new ArrayList<Integer>();
```

```
// Adding the int primitives type values by
// converting them into Integer wrapper object
for (int i = 0; i < 10; i++)
```

```
    System.out.println(
        list.add(Integer.valueOf(i)));
```

```
    }
}
```

Output

true

true

true

true

true

true

true

true

true

true

Another example of auto and unboxing is to find the sum of odd numbers in a list. An important point in the program is that the operators remainder (%) and unary plus (+=) operators do not apply to Integer objects. But still, code compiles successfully because the unboxing of Integer Object to primitive int value is taking place by invoking intValue() method at runtime.

Example 4:

- Java

```
// Java Program to Illustrate Find Sum of Odd Numbers
// using Autoboxing and Unboxing
```

```
// Importing required classes
```

```
import java.io.*;
```

```
import java.util.*;
```

```
// Main class
```

```
class GFG {
```

```
    // Method 1
```

```
    // To sum odd numbers
```

```
    public static int sumOfOddNumber(List<Integer> list)
```

```
    {
```

```
        // Initially setting sum to zero
```

```
        int sum = 0;
```

```
        for (Integer i : list) {
```

```
            // Unboxing of i automatically
```

```
            if (i % 2 != 0)
```

```
                sum += i;
```

```
            // Unboxing of i is done automatically
```

```
            // using intValue implicitly
```

```
            if (i.intValue() % 2 != 0)
```

```
                sum += i.intValue();
```

```
        }
```

```
// Returning the odd sum
return sum;
}

// Method 2
// Main driver method
public static void main(String[] args)
{

    // Creating an empty ArrayList of integer type
    List<Integer> list = new ArrayList<Integer>();

    // Adding the int primitives type values to List
    for (int i = 0; i < 10; i++)
        list.add(i);

    // Getting sum of all odd numbers in List
    int sumOdd = sumOfOddNumber(list);

    // Printing sum of odd numbers
    System.out.println("Sum of odd numbers = "
        + sumOdd);
}
}
```

Output

Sum of odd numbers = 50

Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –

Assume integer variable A holds 10 and variable B holds 20, then –

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21

-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19
----------------	--------------------------------------	--------------

The Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –

Show Examples

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

$a \wedge b = 0011\ 0001$

$\sim a = 1100\ 0011$

The following table lists the bitwise operators –

Assume integer variable A holds 60 and variable B holds 13 then –

Show Examples

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Show Examples

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

The Assignment Operators

Following are the assignment operators supported by Java language –

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

Miscellaneous Operators

There are few other operators supported by Java Language.

Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Following is an example –

Example

```
public class Test {  
  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
  
        b = (a == 10) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

This will produce the following result –

Output

```
Value of b is : 30  
Value of b is : 20
```

instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example –

Example

```
public class Test {  
  
    public static void main(String args[]) {  
  
        String name = "James";  
  
        // following will return true since name is type of String  
        boolean result = name instanceof String;  
        System.out.println( result );  
    }  
}
```

This will produce the following result –

Output

```
true
```

This operator will still return true, if the object being compared is the assignment compatible with the type on the right. Following is one more example –

Example

```
class Vehicle {}  
  
public class Car extends Vehicle {  
  
    public static void main(String args[]) {
```

```

Vehicle a = new Car();
boolean result = a instanceof Car;
System.out.println( result );
}
}

```

This will produce the following result –

Output

```
true
```

Precedence of Java Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, $x = 7 + 3 * 2$; here x is assigned 13, not 20 because operator $*$ has higher precedence than $+$, so it first gets multiplied with $3 * 2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression --expression +expression -expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR		Left to right

Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= = <<= >>= >>>=	Right to left