

Basics in java programming

Variable types and identifier

- To compute the sum and the difference of two numbers say *x* and *y* in a Java program, we must first declare what kind of data will be assigned to them.
- After we assign values to them, we can compute their sum and difference.
- When declaration is made memory locations to store data values for ***x*** and ***y*** are allocated.
- These memory locations are called ***variables***, and *x* and *y* are the names we associate with the memory locations.

- ***Identifiers*** are the names of variables, methods, classes, packages and interfaces.
- They must be composed of only letters, numbers, the underscore, and the dollar sign (\$).
- They cannot contain white spaces. Identifiers may only begin with a letter, the underscore, or the dollar sign.
- A variable cannot begin with a number.
- All variable names are case sensitive.

- There are different types of variables in Java.

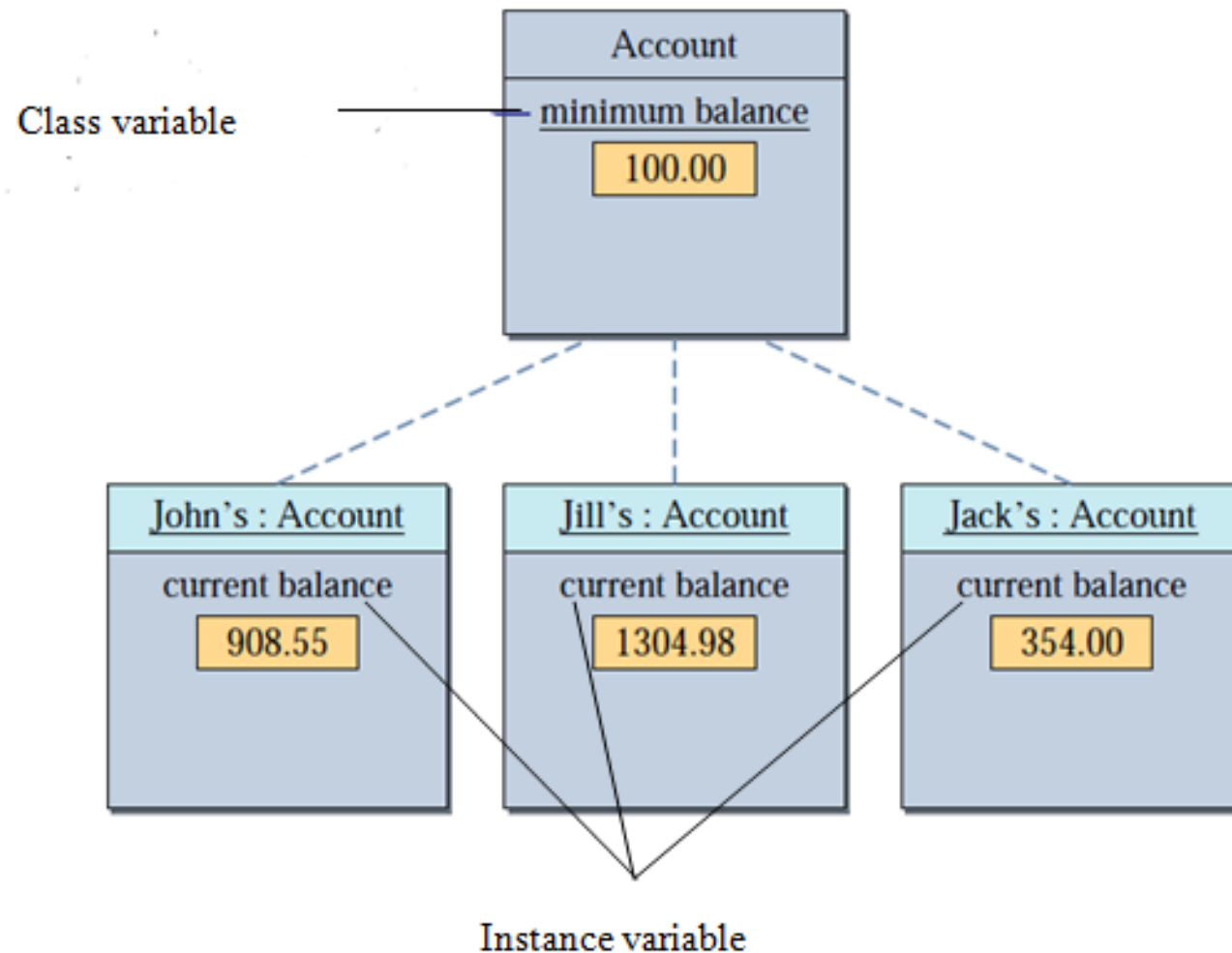
1. Instance Variables (Non-Static Fields)

- Objects store their individual states in “non-static fields”, that is, fields declared without the static keyword.
- Non-static fields are also known as instance variables because their values are unique to each instance of a class.

2. Class Variables (Static Fields)

- A class variable is any field declared with the static modifier;
- this tells the compiler that there is exactly one copy of this variable in existence;
- regardless of how many times the class has been instantiated.

Example on class and instance variable



- In the above diagram current balance is instance variable and every object in the same class will possess same instance variable but the value might be different.
 - i.e. in John's object current balance =908.55, in Jill's object current balance =1304.98, in Jack's object current balance =908.55.
- Minimum balance from this example is class value which means there is only one copy of the variable and all objects can share the variable.

- To appreciate the significance of a class variable, let's see what happens if we represent minimum balance as an instance variable.

<u>John's : Account</u>	<u>Jill's : Account</u>	<u>Jack's : Account</u>
current balance 908.55	current balance 1304.98	current balance 354.00
minimum balance 100.00	minimum balance 100.00	minimum balance 100.00

- The above diagram shows three Account objects having different dollar amounts for the current balance but the same dollar amount for the minimum balance.
- Obviously, this duplication of minimum balance is redundant and wastes space.
- Consider, for example, what happens if the bank raises the minimum balance to \$200.
- If there are 100 Account objects, then all 100 copies of minimum balance must be updated.
- We can avoid this by defining minimum balance as a class variable.

3. Local Variables

- A method stores its temporary state in local variables.
- The syntax for declaring a local variable is similar to declaring a field (for example, `int count = 0;`).
- There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared—between the opening and closing braces of a method.
- As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

- Access modifiers cannot be used for local variables.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.
 - Example: Here, age is a local variable. This is defined inside pupAge() method and its scope is limited to this method only.

```
public class Test{  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
    public static void main(String args[]) {  
        Test test = new Test(); test.pupAge(); }  
}
```

- **Example:** Following example uses age without initializing it, so it would give an error at the time of compilation.

```
public class Test {  
    public void pupAge() {  
        int age; age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
    public static void main(String args[]){  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

Number types, strings, constants

- **Number types**

- There are ***six numerical data types*** in Java: byte, short, int, long, float, and double.
- The data types byte, short, int, and long are for integers; and the data types float and double are for real numbers.
- The data type names byte, short, and others are all reserved words.
- The difference among these six numerical data types is in the range of values they can represent, as shown in Table below.

Data Type	Content	Default Value [†]	Minimum Value	Maximum Value
byte	Integer	0	−128	127
short	Integer	0	−32768	32767
int	Integer	0	−2147483648	2147483647
long	Integer	0	−9223372036854775808	9223372036854775807
float	Real	0.0	−3.40282347E+38 [‡]	3.40282347E+38
double	Real	0.0	−1.79769313486231570E+308	1.79769313486231570E+308

Java numerical data types and their precisions (The character E indicates a number is expressed in scientific notation.)

- A data type with a larger range of values is said to have a ***higher precision***.
 - For example, the data type double has a higher precision than the data type float. The tradeoff for higher precision is memory space—to store a number with higher precision, you need more space.

- We can initialize variable at time of declaration by using assignment operator. The expression for assigning a value to a variable is shown as follows;
 - `<variable> = <expression> ;`
- What we have been calling object names are really variables.
- The only difference between a variable for numbers and a variable for objects is the contents in the memory locations.
- For numbers, a variable contains the numerical value itself; and for objects, a variable contains an address where the object is stored.
- We use an arrow in the diagram to indicate that the content is an address, not the value itself.

Numerical Data

```
int number;
```

```
number = 237;
```

```
number = 35;
```

number



Object

```
Customer customer;
```

```
customer = new Customer();
```

```
customer = new Customer();
```

customer



Content of customer and
number variable after
executing the code inside
the rectangle



```
int number;
```

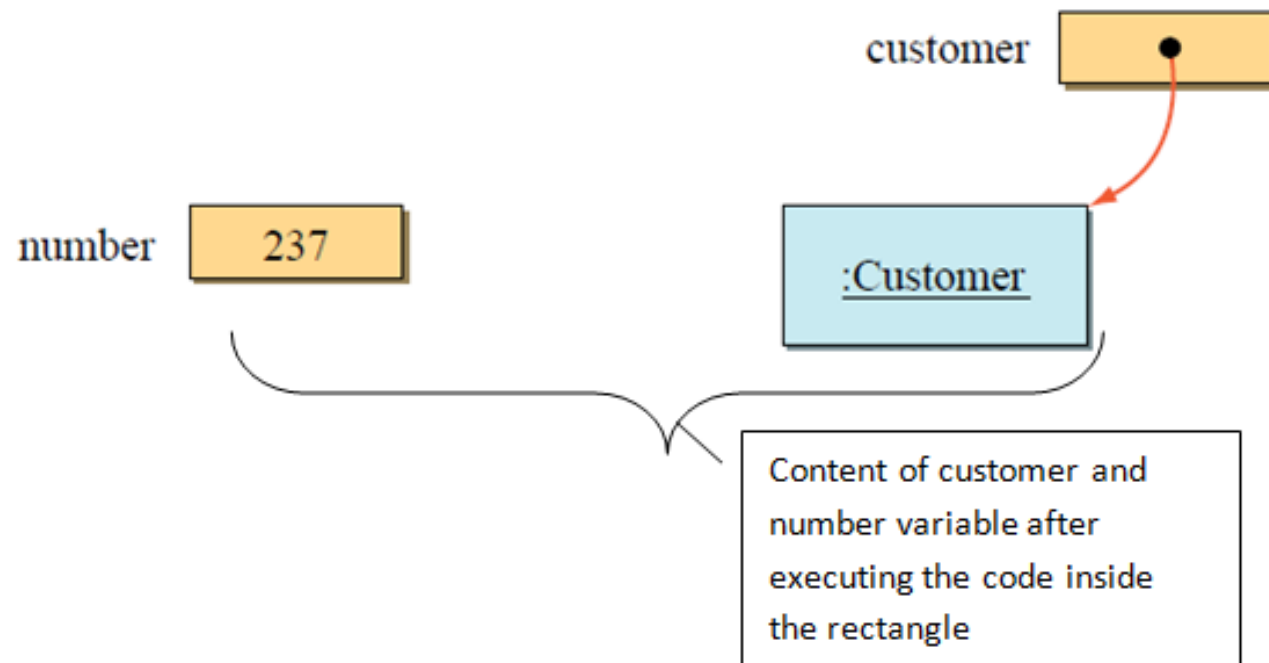
```
number = 237;
```

```
number = 35;
```

```
Customer customer;
```

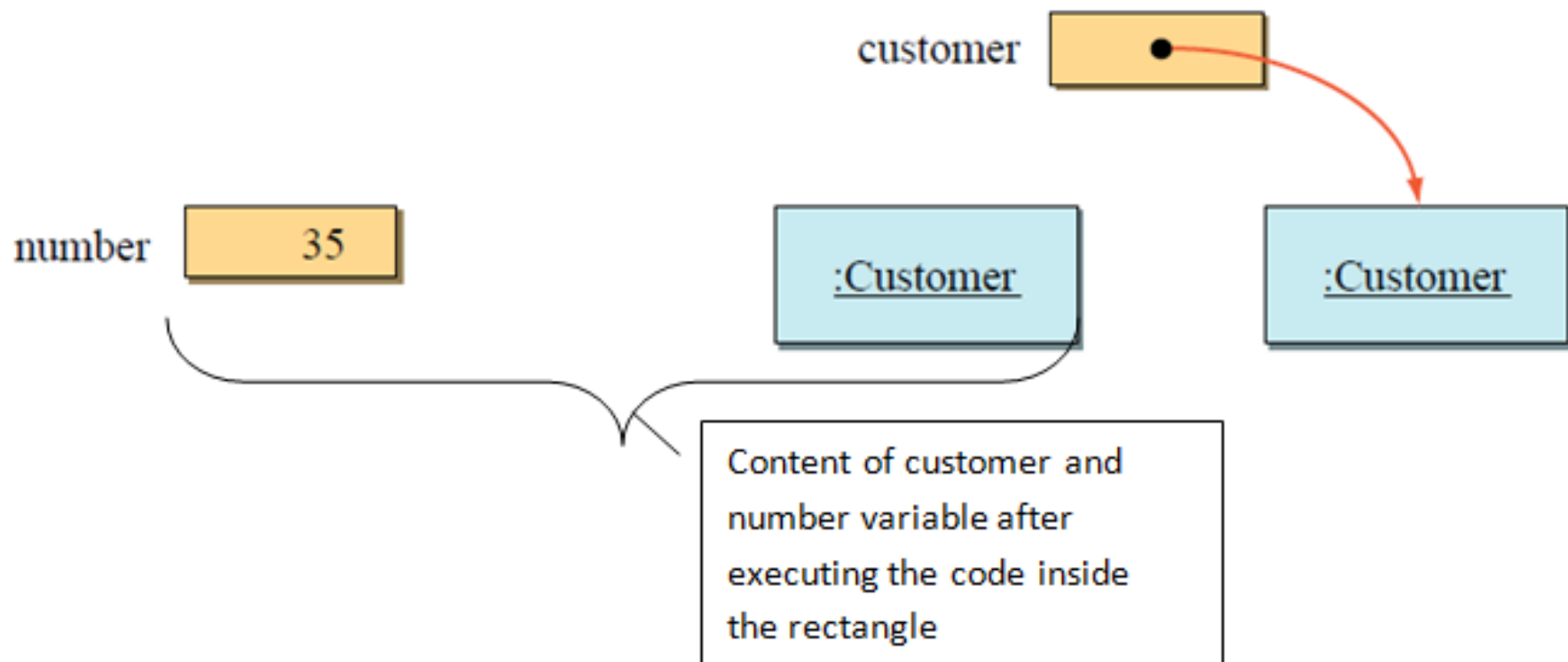
```
customer = new Customer();
```

```
customer = new Customer();
```



```
int number;  
number = 237;  
number = 35;
```

```
Customer customer;  
customer = new Customer();  
customer = new Customer();
```



- The following diagram shows the effects of assigning the content of one variable to another.

Numerical Data

```
int number1, number2;
```

```
number1 = 237;
```

```
number2 = number1;
```

number1



number2



Object

```
Professor alan, turing;
```

```
alan = new Professor();
```

```
turing = alan;
```

alan



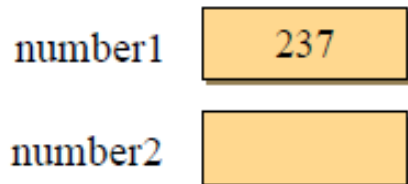
turing



```
int number1, number2;|
```

```
number1 = 237;
```

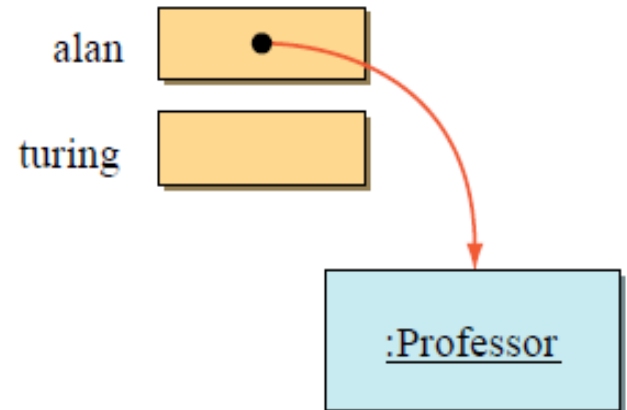
```
number2 = number1|;
```



```
Professor alan, turing;
```

```
alan = new Professor();
```

```
turing = alan|;
```



```
int number1, number2;  
number1 = 237;
```

```
number2 = number1;
```

number1

237

number2

237

```
Professor alan, turing;  
alan = new Professor();
```

```
turing = alan;
```

alan

turing

:Professor

String

- A string is a sequence of characters and in java there is a String class. As String is a class, we can create an instance and give it a name. For example,
 - String name;
 - name = **new** String("hello world");
- Unlike in other classes, the explicit use of new to create an instance is optional for the String class. We can create a new String object, for example, in this way:
 - String name;
 - name = "hello world";
- There are close to 50 methods defined in the String class. We will introduce three of them here: substring, length, and indexOf.

- ***substring()***

- We can extract a substring from a given string by specifying the beginning and ending positions. For example,

```
String text; text = "Espresso";
```

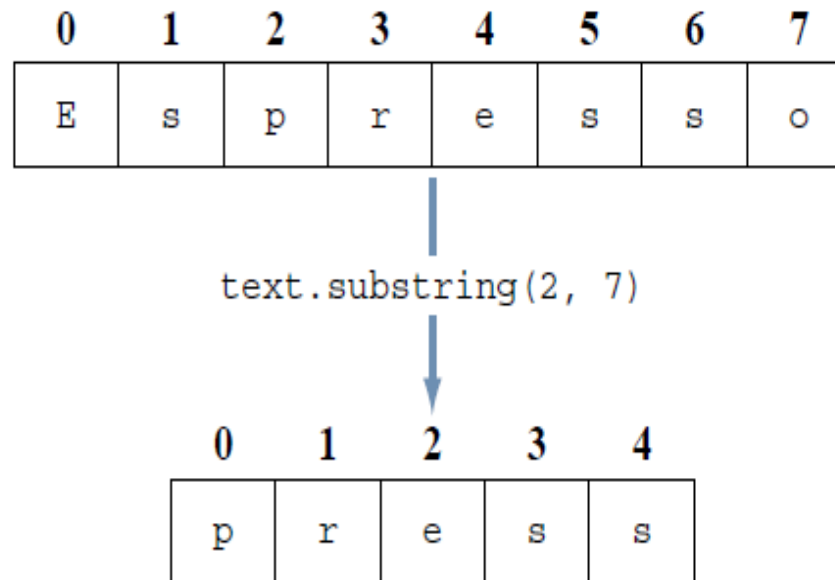
```
System.out.print(text.substring(2, 7));
```



Output:

press

- Individual characters in a String object are indexed from 0.
- The first argument of the substring method specifies the position of the first character, and the second argument specifies the value that is 1 more than the position of the last character.



- An error will result if you pass invalid arguments, such as negative values, the second argument larger than the number of characters in a string, or the first argument larger than the second argument.

- **length()**

- We can find out the number of characters in a String object by using the length method.
- For example, if the name text refers to a string Espresso, then text.length() will return the value 8, because there are eight characters in the string.

- **indexOf()**

- To locate the index position of a substring within another string, we use the indexOf method.
- For example, if the name text refers to a string I Love Java, then `text.indexOf("Love")` will return the value 2, the index position of the first character of the designated string Love.
- If the searched substring is not located in the string, then -1 is returned.

- Notice that the search is done in a case-sensitive manner.
- Thus, `text.indexOf("java")` will return `-1`.
- If there is more than one occurrence of the same substring, the index position of the first character of the first matching substring is returned. Here are some more examples:

text = "I Love Java and Java loves me."

text.indexOf("J") → 7

text.indexOf("love") → 21

text.indexOf("ove") → 3

text.indexOf("ME") → -1

- **String concatenation (+)**

- We can create a new string from two strings by concatenating the two strings.
- We use the plus symbol (+) for string

```
text1 = "Jon";  
text2 = "Java";
```

```
text1 + text2           → "JonJava"
```

```
text1 + " " + text2     → "Jon Java"
```

```
"How are you, " + text1 + "?"
```

```
→ "How are you, Jon?"
```

Constants

- If we want a value to remain fixed, then we use a ***constant***.
- A constant is declared in a manner similar to a variable but constant with the additional reserved word `final`.
- A constant must be assigned a value at the time of its declaration.
- We follow the standard Java convention to name a constant, using only capital letters and underscores here's an example of declaring four constants:


- **final double** PI = 3.14159;
- **final short** FARADAY_CONSTANT = 23060; //
unit is cal/volt
- **final double** CM_PER_INCH = 2.54;
- **final int** MONTHS_IN_YEAR = 12;

Operators and operator's precedence

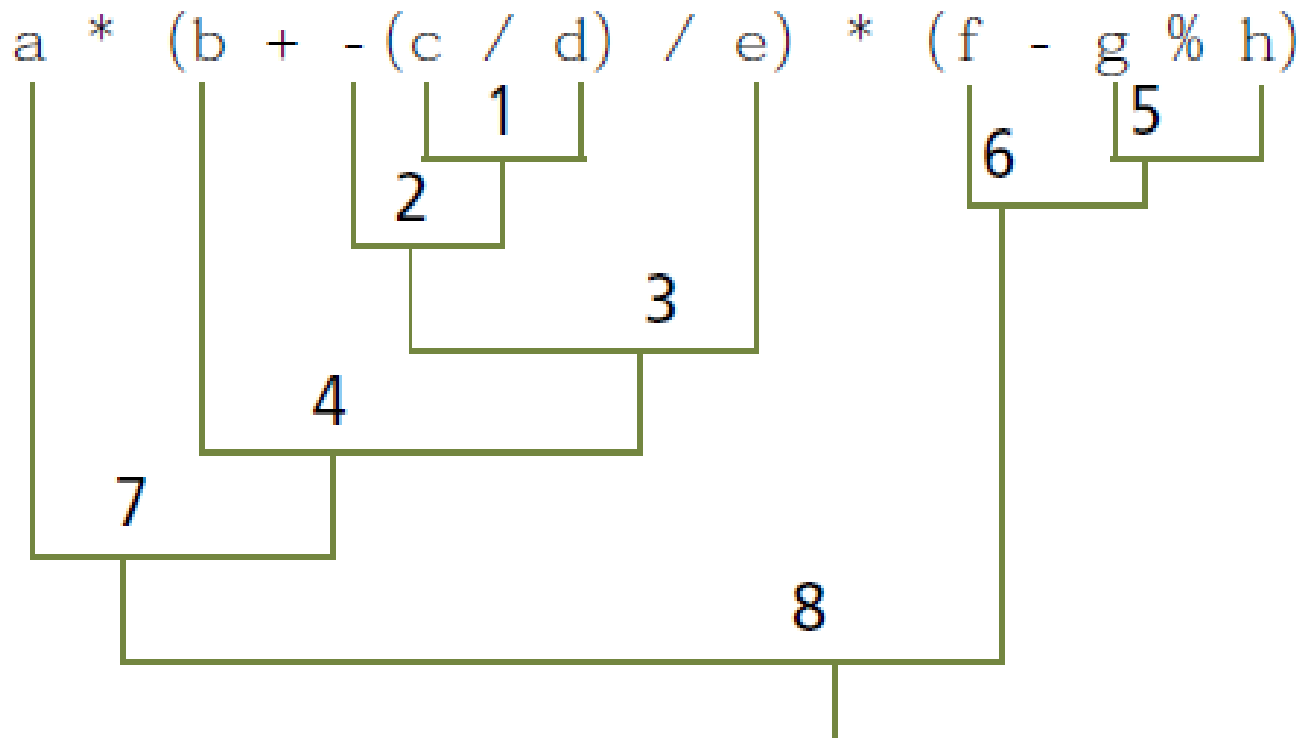
- An expression involving numerical values such as
– 23 + 45
- is called an *arithmetic expression*, because it consists of arithmetic operators and operands.
- An ***arithmetic operator***, such as + in the example, designates numerical computation.
- The following table summarizes the arithmetic operators available in Java.

Operation	Java Operator	Example	Value (x = 10, y = 7, z = 2.5)
Addition	+	x + y	17
Subtraction	-	x - y	3
Multiplication	*	x * y	70
Division	/	x / y	1
		x / z	4.0
Modulo division (remainder)	%	x % y	3

- When two or more operators are present in an expression, we determine the order of evaluation by following the precedence rules.
- The following table shows operator precedence in java.

Order	Group	Operator	Rule
High	Subexpression	()	Subexpressions are evaluated first. If parentheses are nested, the innermost subexpression is evaluated first. If two or more pairs of parentheses are on the same level, then they are evaluated from left to right.
	Unary operator	- , +	Unary minuses and pluses are evaluated second.
	Multiplicative operator	* , / , %	Multiplicative operators are evaluated third. If two or more multiplicative operators are in an expression, then they are evaluated from left to right.
	Additive operator	+ , -	Additive operators are evaluated last. If two or more additive operators are in an expression, then they are evaluated from left to right.
Low			

- The following example illustrates the precedence rules applied to a complex arithmetic expression:



Getting Numerical Input

- To input strings, we use the next method of the Scanner class.
- For the numerical input values, we use an equivalent method that corresponds to the data type of the value we try to input.
- For instance, to input an int value, we use the nextInt method. Here's an example of inputting a person's age:

- Scanner scanner = **new** Scanner(System.in);
- **int** age;
- System.out.print("Enter your age: ");
- age = scanner.nextInt();
- In addition to the int data type, we have five input methods that correspond to the other numerical data types.
- The six input methods for the primitive numerical data types are listed in Table below

Method	Example
<code>nextByte()</code>	<code>byte b = scanner.nextByte();</code>
<code>nextDouble()</code>	<code>double d = scanner.nextDouble();</code>
<code>nextFloat()</code>	<code>float f = scanner.nextFloat();</code>
<code>nextInt()</code>	<code>int i = scanner.nextInt();</code>
<code>nextLong()</code>	<code>long l = scanner.nextLong();</code>
<code>nextShort()</code>	<code>short s = scanner.nextShort();</code>

Type conversion/casting

- When the data types of variables and constants in an arithmetic expression are different data types, then a casting conversion will take place.
- A casting conversion, or typecasting, is a process that converts a value of one data type to another data type.
- Two types of casting conversions in Java are implicit and explicit.

- An implicit conversion called numeric promotion is applied to the operands of an arithmetic operator.
- The promotion is based on the rules stated in Table below.
- This conversion is called promotion because the operand is converted from a lower to a higher precision.

Operator Type	Promotion Rule
Unary	<ol style="list-style-type: none">1. If the operand is of type <code>byte</code> or <code>short</code>, then it is converted to <code>int</code>.2. Otherwise, the operand remains the same type.
Binary	<ol style="list-style-type: none">1. If either operand is of type <code>double</code>, then the other operand is converted to <code>double</code>.2. Otherwise, if either operand is of type <code>float</code>, then the other operand is converted to <code>float</code>.3. Otherwise, if either operand is of type <code>long</code>, then the other operand is converted to <code>long</code>.4. Otherwise, both operands are converted to <code>int</code>.

- Instead of relying on implicit conversion, we can use explicit conversion to convert an operand from one data type to another.
- Explicit conversion is applied to an operand by using a typecast operator.
- For example, to convert the int variable x in the expression
 - $x / 3$
- to float so the result will not be truncated, we apply the typecast operator (float) as (float) x / 3
- The syntax is
 - (<data type>) <expression>

- The typecast operator is a unary operator and has a precedence higher than that of any binary operator.
- You must use parentheses to typecast a subexpression; for example, the expression
– $a + (\text{double}) (x + y * z)$
- will result in the subexpression $x + y * z$ typecast to double.

Decision and repetition statements

Overview of java statements

If statements

- **If-else statement**
- There are two versions of the if statement, called if-then-else and if-then.
- We use an if statement to specify which block of code to execute. A block of code may contain zero or more statements.
- Which block is executed depends on the result of evaluating a test condition, called a Boolean expression. The if-then-else statement follows this general format:

if (<boolean expression>)

<then block>

else

<else block>

- If multiple statements are needed in the <then block> or the <else block>, they must be surrounded by braces { and }.
- The <boolean expression> is a conditional expression that is evaluated to either true or false.
- The six relational operators we can use in conditional expressions are:

`< // less than`

`<= // less than or equal to`

`== // equal to`

`!= // not equal to`

`> // greater than`

`>= // greater than or equal to`

Example 1:

```
Scanner scanner = new Scanner(System.in);  
System.out.print("Enter test score: ");  
int testScore = scanner.nextInt();  
if (testScore < 70)  
    System.out.println("You did not pass");  
else  
    System.out.println("You did pass");
```

Example 2:

`a * a <= c` //true if a * a is less than or equal to c

`x + y != z` //true if x + y is not equal to z

`a == b` //true if a is equal to b

Nested if Statements

- The else blocks of an if statement can contain any statement including another if statement.
- An if statement that contains another if statement in either its then or else block is called a ***nested if*** statement.
- It is better to apply the nested if structure if we have to test conditions in some required order.
- **Example 1** : In the example 1, we printed out the messages You did pass or You did not pass depending on the test score. Let's modify the code to print out three possible messages. If the test score is lower than 70, then we print You did not pass, as before. If the test score is 70 or higher, then we will check the student's age. If the age is less than 10, we will print You did a great job. Otherwise, we will print You did pass, as before.

```
if (testScore >= 70) {  
    if (studentAge < 10) {  
        System.out.println("You did a great job");  
    } else {  
        System.out.println("You did pass");//test score >=  
            70  
    } //and age >= 10  
    } else { //test score < 70  
        System.out.println("You did not pass");  
    }  
}
```

- It is possible to write if tests in different ways to achieve the same result. For example, the preceding code can also be expressed as:

```
if (testScore >= 70 && studentAge < 10) {  
    System.out.println("You did a great job");  
} else {  
    //either testScore < 70 OR studentAge >= 10  
    if (testScore >= 70) {  
        System.out.println("You did pass");  
    } else { System.out.println("You did not pass");}}
```


- **Example 2:** the following piece of code prints out letter grade for the given range of marks.

Test Score	Grade
$90 \leq \text{score}$	A
$80 \leq \text{score} < 90$	B
$70 \leq \text{score} < 80$	C
$60 \leq \text{score} < 70$	D
$\text{score} < 60$	F

```
if (score >= 90)
System.out.println("Your grade is A");
else
if (score >= 80)
System.out.println("Your grade is B");
else
if (score >= 70)
System.out.println("Your grade is C");
else
if (score >= 60)
System.out.println("Your grade is D");
else
System.out.println("Your grade is F");
```

If-else if- else

- If we have many selections and one at a time is going to be true then we can use the if else if-else statement.
- **Example:** example2 can also be written using if- else if- else statement which is more readable.

```
if (score >= 90)
System.out.println("Your grade is A");
else if (score >= 80)
System.out.println("Your grade is B");
else if (score >= 70)
System.out.println("Your grade is C");
else if (score >= 60)
System.out.println("Your grade is D");
else
System.out.println("Your grade is F");
```

- Boolean expressions can contain conditional and Boolean operators.
- A Boolean operator, also called a logical operator, takes Boolean values as its operands and returns a Boolean value.
- Three Boolean operators are AND, OR, and NOT. In Java, the symbols `&&`, `||`, and `!` represent the AND, OR, and NOT operators, respectively.

Switch statement

- Another Java statement that implements a selection control flow is the switch ***statement***.
- The syntax for the switch statement is

```
switch ( <integer expression> ) {  
    <case label 1> : <case body 1>  
    ...  
    <case label n> : <case body n>  
}
```

- The <case label i> has the form case <integer constant> or default and <case body i> is a sequence of zero or more statements.
- Notice that <case body i> is not surrounded by left and right braces.
- The <constant> can be either a named or literal constant.
- Each case in the sample switch statement is terminated with the break statement.
- The break statement causes execution to continue from the statement following this switch statement, skipping the remaining portion of the switch statement.
- The following example illustrates how the break statement works:

- We may include a default case that will always be executed if there is no matching case. For example, we can add a default case to print out an error message if any invalid value for ranking is entered.

When this code is
executed, the output

is

1

2

3

```
//Assume necessary declaration and object creation are  
done  
selection = 1;  
switch (selection) {  
case 0: System.out.println(0);  
case 1: System.out.println(1);  
case 2: System.out.println(2);  
case 3: System.out.println(3);  
}
```


For loop

- The general format of the for statement is
for (<initialization>; <boolean expression>; <update>)
 <statement>
- The following code computes the sum of the first 100 positive integers:
 int i, sum = 0;
 for (i = 1; i <= 100; i++) {
 sum += i; //equivalent to sum = sum + i}

While loop

- The statement follows the general format
while (<boolean expression>)
 <statement>
- where <statement> is either a <single statement> or a <compound statement>.

- **Example 1:** The following example computes the sum of the first 100 positive integers using while loop

```
int sum = 0, number = 1;  
while (number <= 100) {  
    sum = sum + number;  
    number = number + 1;  
}
```

Do-while loop

- The while statement is characterized as a ***pretest loop*** because, the test is done before execution of the loop body.
- Because it is a pretest loop, the loop body may not be executed at all.
- The do-while is a repetition statement that is characterized as a ***posttest loop***.
- With a posttest loop statement, the loop body is executed at least once.
- The general format for the do-while ***statement*** is

do

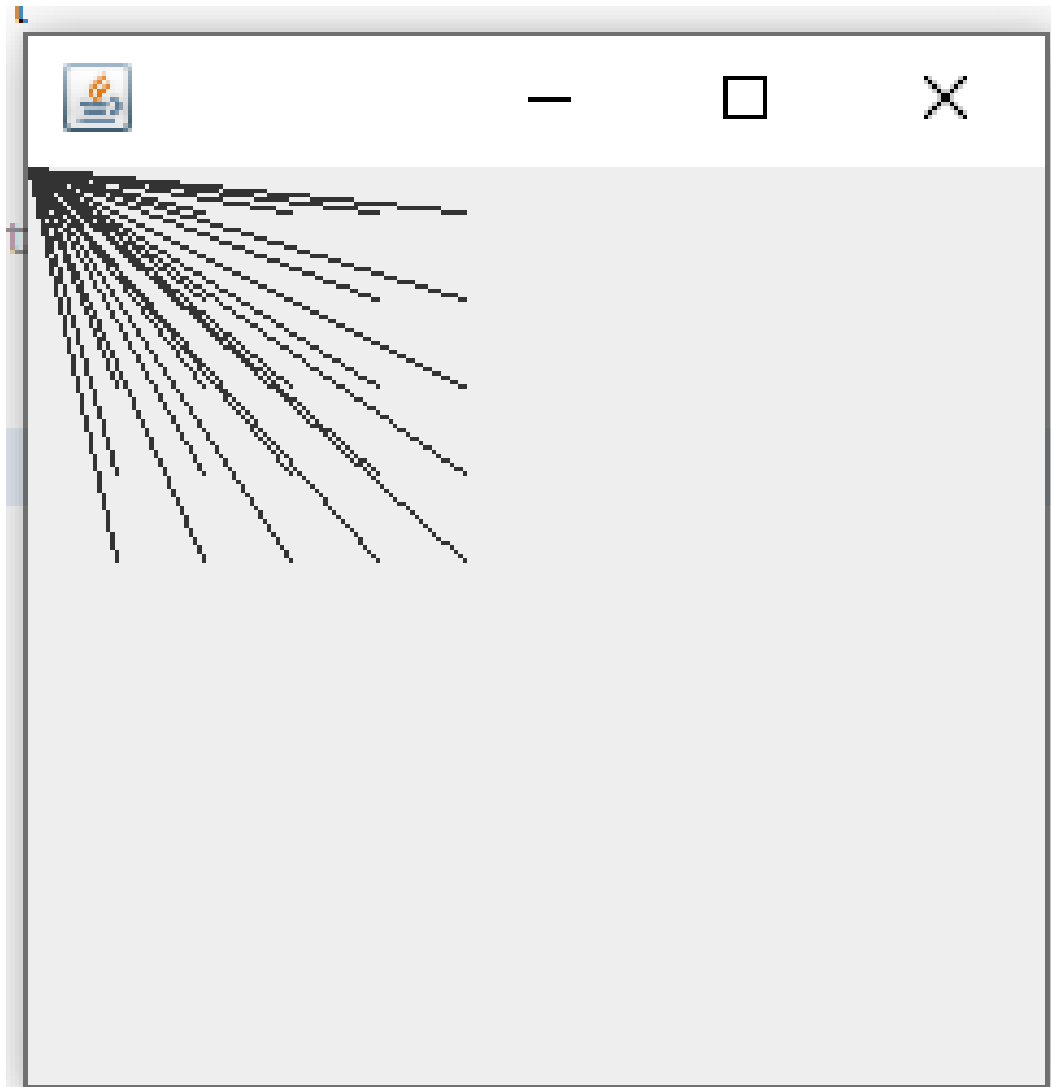
<statement>

while (<boolean expression>) .

- The <statement> is executed until the <boolean expression> becomes false. Remember that <statement> is either a <single statement> or a <compound statement>.
- **Example:** the following program computes the sum of odd integers entered by the user. We will stop the loop when the sentinel value 0 is entered, an even integer is entered, or the sum becomes larger than 1000.

```
sum = 0;
do {
System.out.print("Enter integer: ");
num = scanner.nextInt();
if (num == 0) { //sentinel
System.out.print("Sum = " + sum);
} else if (num % 2 == 0) //invalid data
System.out.print("Error: even number was entered");
} else {
sum += num;
if (sum > 1000) { //pass the threshold
System.out.print("Sum became larger than 1000");
}}
} while ( !(num % 2 == 0 || num == 0 || sum > 1000) );
```

Example: line drawing using for loop



```
import java.awt.Graphics;
import javax.swing.*;
import javax.swing.JPanel;
public class DrawPanel extends
JPanel {
    @Override
    public void paintComponent(
Graphics g ){
        super.paintComponent( g );

        for(int i=100;i>0;i-=20){
            for(int j=10;j<100;j+=20){
                g.drawLine(0,0,i,j);
            }
        }
    }
}
```

```
public static void main( String
args[] )
{
    // create a panel that contains our
    drawing
    DrawPanel panel = new
    DrawPanel();
    JFrame application = new JFrame();

    application.setDefaultCloseOperatio
n( JFrame.EXIT_ON_CLOSE );
    application.add( panel ); // add the
    panel to the frame
    application.setSize( 250, 250 ); // set
    the size of the frame
    application.setVisible( true ); //
    }
}
```