# Java Programming

# Unit-1


By:
Dr. Ritu Jain
Associate Professor,
MCA, BCA Department

# Syllabus

| | |
|---|---|
| 1. | **Unit-1: Introduction to Java**<br>• History of Java,<br>• JDK, JRE, JVM,<br>• Features of Java<br>• Programming Concepts<br>  ○ Identifiers, Keywords, Data Types, Variables, Operators, Type Conversion<br>  ○ Control Structure: Selection Statements, Iteration statements, Jump statements<br>  ○ Arrays (1D, 2D array) & Strings (String, String Buffer class)<br>• Java Input / Output Operations. |

created by Dr. Ritu Jain

# Syllabus

| | |
|---|---|
| **2.** | **Unit-2: OOP Constructs:**<br>• Class & Objects<br>  ○ Defining Class with data members and methods,<br>  ○ Creating objects & accessing members of class,<br>• Object Initialization using Constructor,<br>  ○ Types of constructors,<br>  ○ this keyword<br>• Access specifiers<br>  ○ Public, Private and Protected, Default access<br>• Modifiers<br>  ○ Static, Final<br>• Inheritance<br>  ○ Types of Inheritance:<br>  ▪ Single, Multilevel and Hierarchical Inheritance<br>  ○ Super keyword<br>• Polymorphism<br>  ○ Types of Polymorphism<br>  ▪ Method overloading<br>  ▪ Method overriding |

# Syllabus

| | |
|---|---|
| 3. | **Unit-3: Interface, Packages and Exception Handling**<br>• Abstract class<br>  ○ Definition of Abstract class and Implementation<br>• Interface<br>  ○ Definition of Interface Implementing an Interface<br>• Package<br>  ○ Introduction Creating Package<br>  ○ Importing Package / Class from package Package scope<br>• Exception Handling<br>  ○ Introduction to Exception and Exception Handling<br>  ○ Types of Exception,<br>  ○ Exception class hierarchy<br>  ○ Exception handling using try, catch and finally block<br>  ○ Use of throw and throws statements<br>• User defined exception |

# Syllabus

| | |
|---|---|
| **4.** | **Unit-4: Java Input / Output & Multithreading**<br>• Java I/O package<br>  ○ IO class Hierarchy<br>  ○ Byte Stream and Character Stream classes<br>  ○ Buffered Reader and writer classes Classes for file IO operations<br>  ○ PrintWriter class<br>• Multithreading in Java<br>  ○ Introduction to multithreading<br>  ○ Thread Life Cycle<br>  ○ Creating Thread using Thread class or Runnable Interface<br>  ○ Main Thread and Thread Properties<br>  ○ Creating multithreaded application<br>  ○ Thread Synchronization and Communication |

# Syllabus

| | |
|---|---|
| 5. | **Unit-5: Java Collection Framework**<br>• Introduction to collection framework<br>• Collection Interface,<br>• Classes and Iterator Collection,<br>• List interface ArrayList, LinkedList<br>• Set, SortedSet, HashSet, TreeSet classes |

created by Dr. Ritu Jain

# Syllabus

**Unit-6: Java Swing**

- Introduction to Swing class
- Features of Swing class
- Containers: JFrame
- Swing Component Classes
  - JButton
  - JLabel
  - JTextField
  - JComboBox
  - JCheckBox
  - JRadioButton
- Creating Menu using Swing
- Layout Manager: FlowLayout, GridLayout
- Event Delegation Model
  - Listener Interface and Event class: Action Listener and Action Event

# Reference Books


1. Java Complete Reference, Herbert Schildt, TMH
2. Programming with Java A Primer, E. Balagurusamy, TMH
3. Java 6 Programming Black Book , Kogent Solution Inc, dreamTech Pub
4. Core Java 2 Volume – I, Cay S Horstmann, Fary Cornell, Sun Microsystems Press

# Course Objectives

1. To learn about the concepts and principles of java programming.
2. To understand and implement fundamentals of object-oriented programming in Java, including defining classes, invoking methods, using class libraries, etc.
3. To understand and implement Interface, Packages and Exception Handling.
4. To develop application using file handling and describe multithreading.
5. To understand and create various data structure using collection framework.
6. To develop GUI application using Swing programming

# Course Outcomes

After learning the course, the students should be able to:

1. Describe Programming Constructs and features of OOP in Java.
2. Use the OOP programming constructs to write a Java Program.
3. Develop the Java application by applying the programming constructs and features like Interface, Package and Exception Handling.
4. Implement file handling and describe the features of multithreading.
5. Understand and create various data structure using collection framework.
6. Develop the GUI application using Swing.

# Java

- Java is a high level, robust, object-oriented and secure programming language.
- It is intended to let application developers **write once, and run anywhere (WORA),** meaning that compiled Java code can run on all platforms that support Java without the need for recompilation.

# History of Java

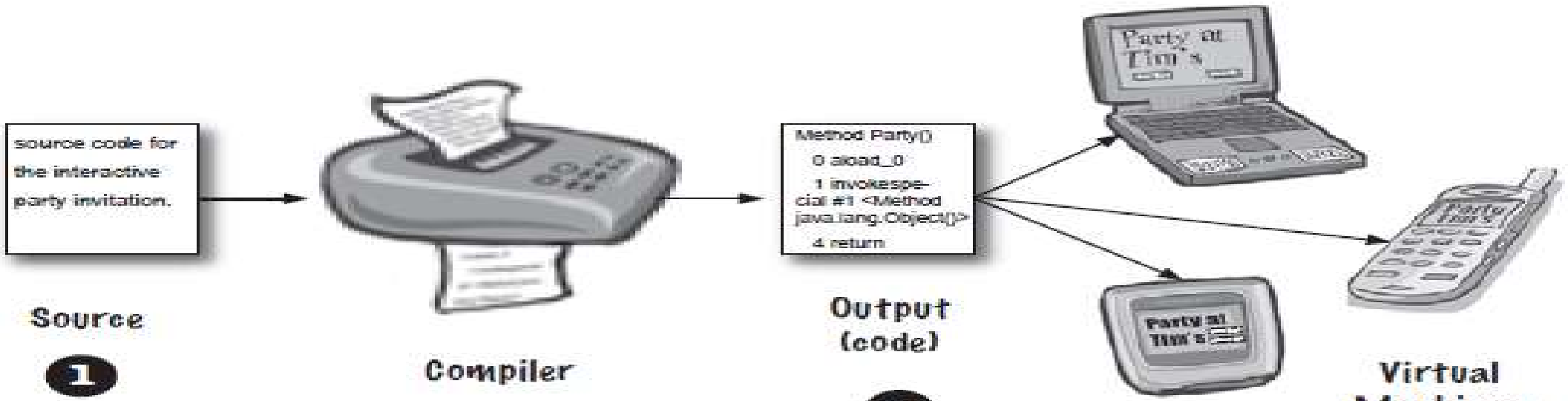
- Java was conceived by James Gosling and his team at Sun Microsystems, Inc. in 1991.
- **James Gosling** is known as the Father of Java.
- Primary motivation was the need for a **platform-independent (that is, architecture-neutral) language** that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- Initially, it was called **Oak** and it was developed as a part of the Green project.
- In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.
- JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language.
- Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, etc.

# Implementation of a Java Application Program

- Implementation of a Java application program involves a following step. They include:
  1. Creating the program
  2. Compiling the program
  3. Running the program

# The Way Java Works

**The goal is to write one application (in this example, an interactive party invitation) and have it work on whatever device your friends have.**




**Source**

1

Create a source document. Use an established protocol (in this case, the Java language).

**Compiler**

2

Run your document through a source code compiler. The compiler checks for errors and won't let you compile until it's satisfied that everything will run correctly.

**Output (code)**

3

The compiler creates a new document, coded into Java ***bytecode***. Any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform-independent.

**Virtual Machines**

4

Your friends don't have a physical Java Machine, but they all have a ***virtual*** Java machine (implemented in software) running inside their electronic gadgets. The virtual machine reads and *runs* the bytecode.

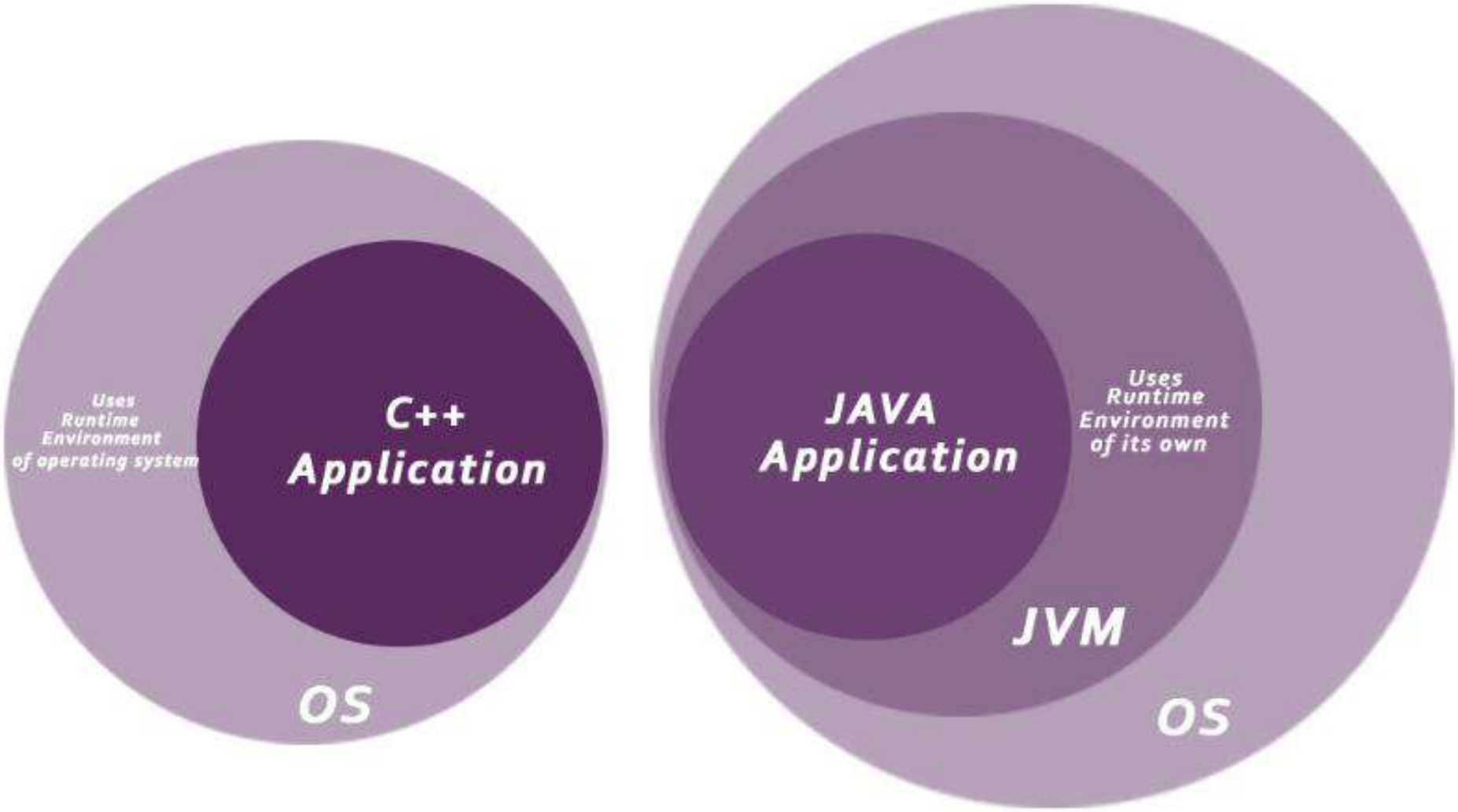
# Java Virtual Machine (JVM)

- There are three execution phases of a program. They are written, compile and run the program.
  - Writing a program is done by a java programmer.
  - The compilation is done by the **JAVAC** compiler which is a Java compiler included in the Java development kit (JDK). It takes the Java program as input and generates bytecode as output.
  - In the Running phase of a program, **JVM** executes the bytecode generated by the compiler.
- Function of Java Virtual Machine is to execute the bytecode produced by the compiler.
- Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a **platform-independent language.**

# Bytecode

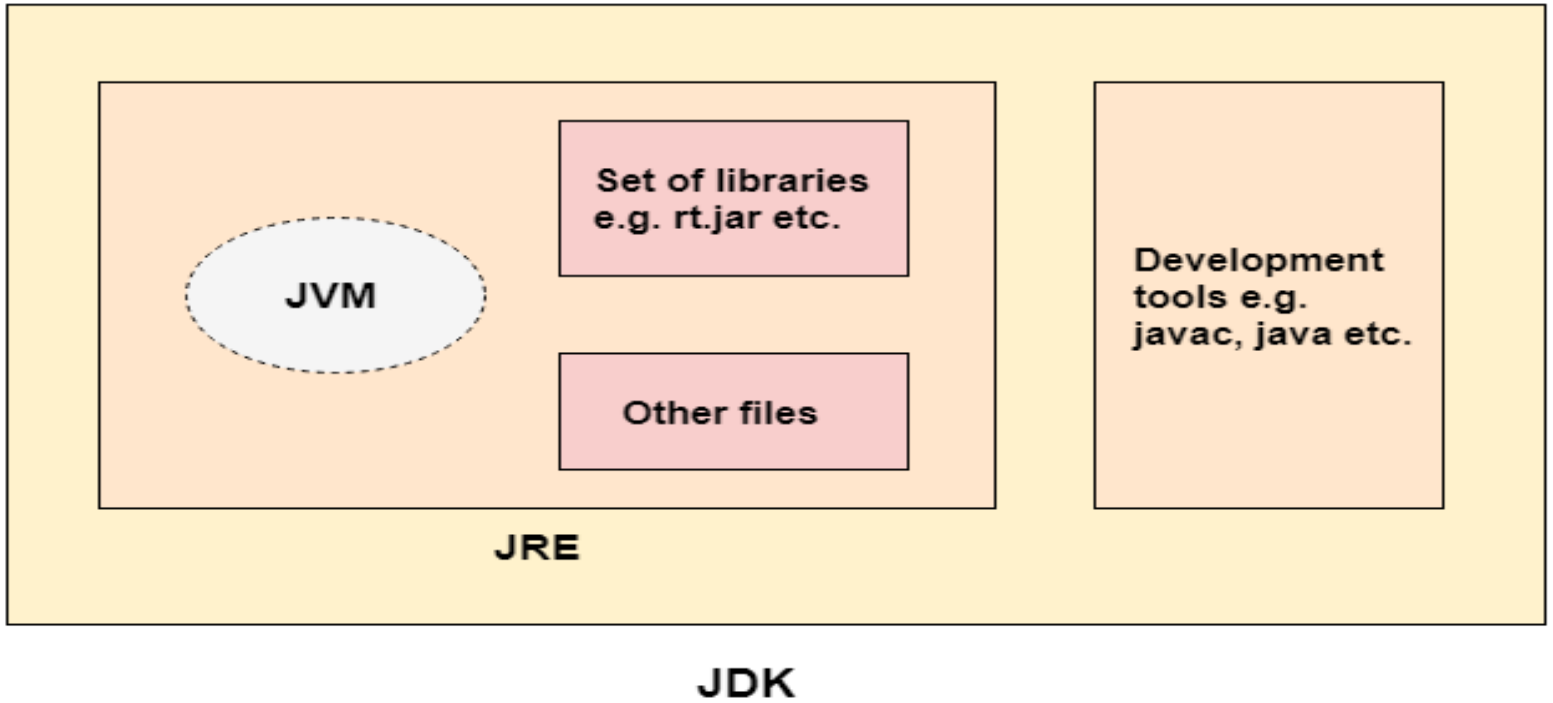
- Java addressed **portability and security** concerns by the fact that the:
  *"Output of Java compiler is not executable code. Rather, it is bytecode."*
- *Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).*
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.
- The fact that Java program is executed by JVM also helps to make it secure. Because execution in under the control of the JVM, it ensures that no side effects outside the system are generated.

# Java Virtual Machine (JVM)



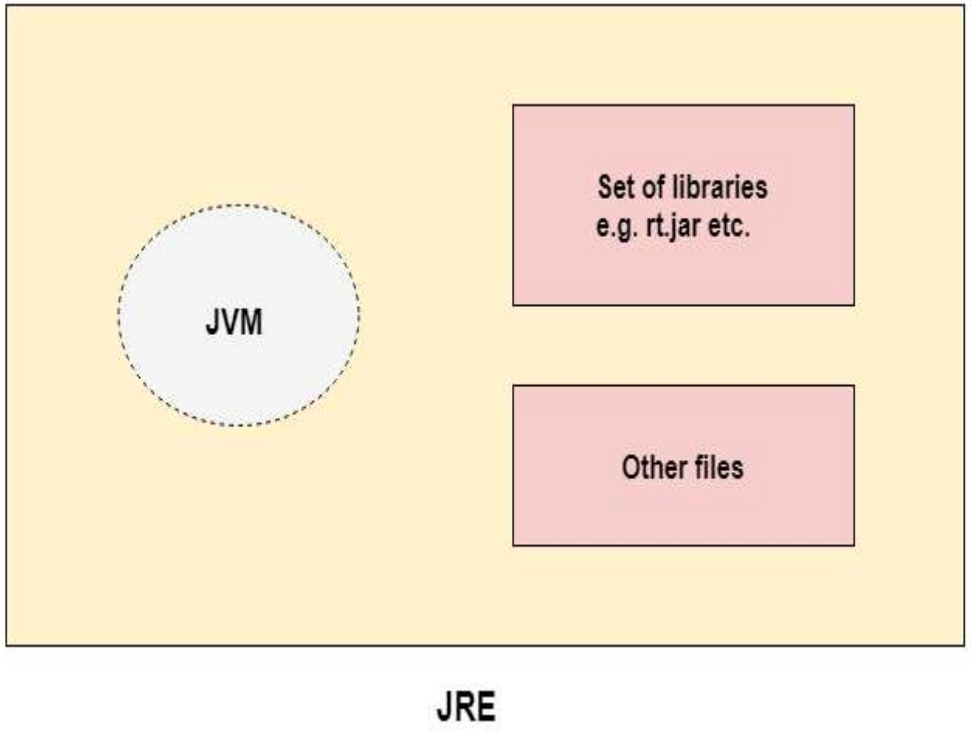
# Java Development Kit (JDK)

- JDK includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc.
- For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.



# Java Runtime Environment (JRE)

- JDK includes JRE. JRE installation on our computers allows the java program to run. It is used to provide the runtime environment.
- It contains
  - a set of libraries
  - other files that JVM uses at runtime.



# Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

- **Standalone Application**: also known as desktop applications or window-based applications. Examples of standalone application are Media player, antivirus, etc.
- **Web Application:** The presence of JSP, web servers, spring, and Hibernate provides feasibility in the web development process.
- **Enterprise Application:** Java is a popular choice for building enterprise-level applications due to its scalability, reliability, and security.
- **Mobile Application:** is used to build applications that run across smartphones and other small-screen devices.

# Features of Java

The Java programming language is characterized by all of the following buzzwords:

- **Simple**
- **Object oriented**
- **Distributed**
- **Interpreted**
- **Robust**
- **Secure**
- **Architecture neutral**
- **Portable**
- **High performance**
- **Multithreaded**
- **Dynamic**

**Some of the features such as, secure, portable have already been discussed.**

# Features of Java

- **Simple**
  - Java was designed to be easy for professional programmer to learn and use effectively.
  - It's simple and easy to learn if you already know the basic concepts of Object Oriented Programming.
  - C++ programmer can move to JAVA with very little effort to learn.
  - It does not have complex features like pointers, operator overloading, multiple inheritances, and explicit memory allocation.

# Features of Java

- **Object Oriented**
  - Java is object oriented language. All program code and data reside within objects and classes.
  - Almost "Everything is an Object" paradigm.
  - Java comes with an extensive set of classes, arranged in packages that can be used in our programs through inheritance.
  - Organizing the program in the terms of a collection of objects is a way of object-oriented programming, each of which represents an instance of the class.
  - The four main concepts of Object-Oriented programming are:
    - Abstraction
    - Encapsulation
    - Inheritance
    - Polymorphism

# Features of Java

- **Distributed**
  - Java is designed for distributed environment of the Internet. In Java, accessing a resource using a URL is not much different from accessing a file.
  - Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

# Features of Java

- **Compiled and Interpreted**
  - Usually a computer language is either compiled or Interpreted. Java combines both this approach and makes it a two-stage system.
  - Compiled : Java enables creation of a cross platform programs by compiling into an intermediate representation called Java Bytecode.
  - Interpreted : Bytecode is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

# Features of Java

- **Robust**
  - Java is developed in such a way that it puts a lot of effort into **checking errors** as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language.
  - Java is a strictly typed language. It checks code both at compile time and runtime.
  - The main features of java that make it robust are garbage collection, exception handling, and memory allocation.

# Features of Java

- **Secure**
  - In java, we don't have pointers, so we cannot access out-of-bound arrays i.e it shows ArrayIndexOutOfBound Exception if we try to do so. That's why several security flaws are not possible in Java.
  - Also, java programs run in an environment that is independent of the OS (operating system) environment which makes java programs more secure.
  - Java achieves this protection by confining a Java program to the java execution environment and not allowing it to access other parts of the computer.

# Features of Java

- **Architecture Neutral**
  - Java language and the Java Virtual Machine has been designed to achieve the goal of "**write once; run anywhere, any time, forever**."
  - Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler.
  - This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa.
  - Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of the bytecode. That is why we call java a platform-independent language.

# Features of Java

- **Portable**
  - As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

# Features of Java

- **High Performance**
  - Java architecture is defined in such a way that it reduces overhead during the runtime and at some times java uses Just In Time (JIT) compiler where the compiler compiles code on-demand basis where it only compiles those methods that are called making applications to execute faster.

# Features of Java

- **Multithreaded**
  - Java supports multithreading.
  - It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU.

# Features of Java

- **Dynamic**
- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- This makes it possible to dynamically link code in a safe and expedient manner.

# Object oriented Paradigms

- **Abstraction:** Abstraction in Java is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.
- **Encapsulation:** *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- **Inheritance** is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.
- **Polymorphism** (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

# A First Simple Program

```
/* This is a simple Java program. Call this file "Example.java". */
class Example
{
        public static void main(String args[])
        {
                System.out.println("This is a simple Java program.");
        }
 }
```

# Entering the Program

- The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**.
- In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. It should also have same case as Java is case-sensitive.

# Compiling and executing the Program

- To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

**javac Example.java**

- The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. The output of **javac** is not code that can be directly executed.
- To actually run the program, you must use **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

**java Example**

# Typical cycle in Java

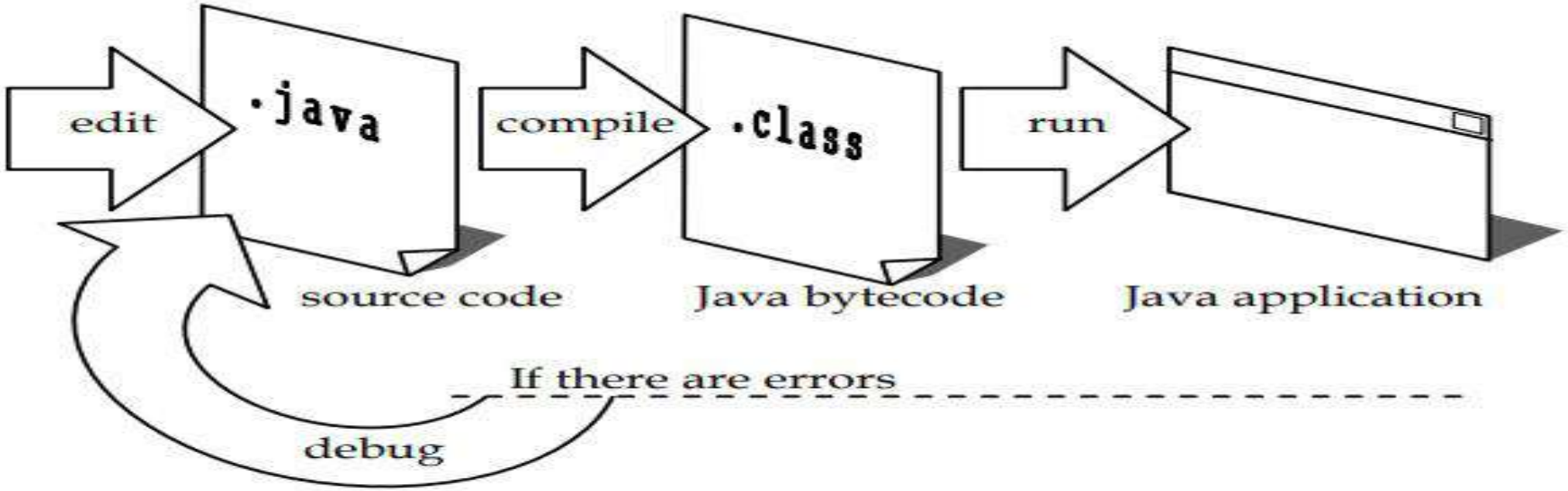
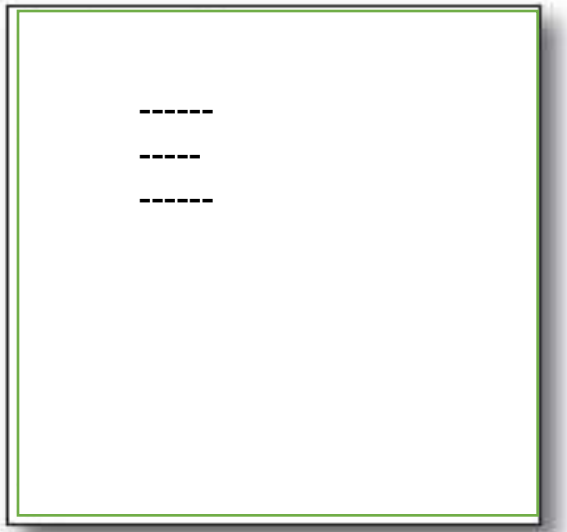



**Figure 13: Typical cycle in Java programming**

# How to execute a Java Program

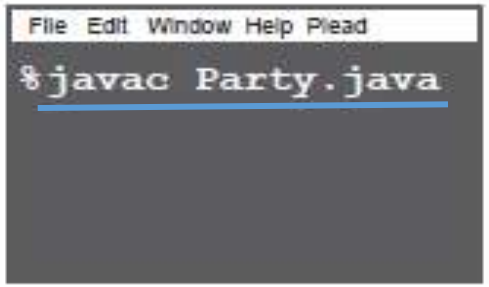



**Source**

1

Type your source code.

Save as: ***Party.java***

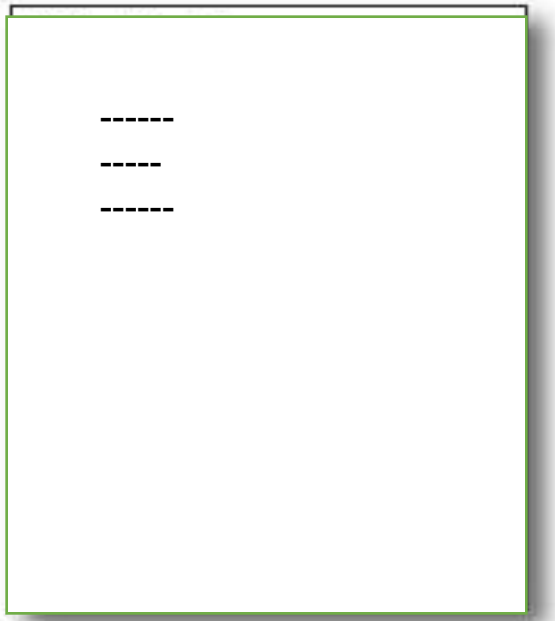



**Compiler**

2

Compile the ***Party.java*** file by running `javac` (the compiler application). If you don't have errors, you'll get a second document named ***Party.class***

The compiler-generated Party.class file is made up of *bytecodes*.




**Output (code)**

3

Compiled code: ***Party.class***


**Virtual Machines**

4

Run the program by starting the Java Virtual Machine (JVM) with the ***Party.class*** file. The JVM translates the *bytecode* into something the underlying platform understands, and runs your program.

# Lexical Issues

- Java programs are a collection of
  - *whitespace*
  - *identifiers*
  - *literals*
  - *separators*
  - *operators*
  - *comments*
  - *keywords*

created by Dr. Ritu Jain

# Java is a free-form language

- You do not need to follow any special indentation rules.
- For instance, the **Example** program could have been written all on one line or in any other strange way you felt like typing it.
- In Java, *whitespace is a space, tab, or newline, or line feed*.

# Identifiers

- Identifiers are used for class names, method names, and variable names.
- An identifier can contain:
  - uppercase letters,
  - Lowercase letters,
  - numbers,
  - Underscore
  - and dollar-sign characters (not intended for general use)
- They must not begin with a number, lest they be confused with a numeric literal.
- Java is case-sensitive, so **VALUE** is a different identifier than **Value**.
- Some examples of valid identifiers are:

  AvgTemp, count, a4, $test, this_is_ok
- Invalid identifier names include these:

  2count, high-temp, Not/ok

# Separators

| Symbol | Name | Purpose |
|---|---|---|
| ( ) | Parentheses | Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types. |
| { } | Braces | Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes. |
| [ ] | Brackets | Used to declare array types. Also used when dereferencing array values. |
| ; | Semicolon | Terminates statements. |
| , | Comma | Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement. |
| . | Period | Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable. |
| :: | Colons | Used to create a method or constructor reference. |
| ... | Ellipses | Indicate a variable-arity parameter. |
| @ | Ampersand | Begins an annotation |

# Java Keywords

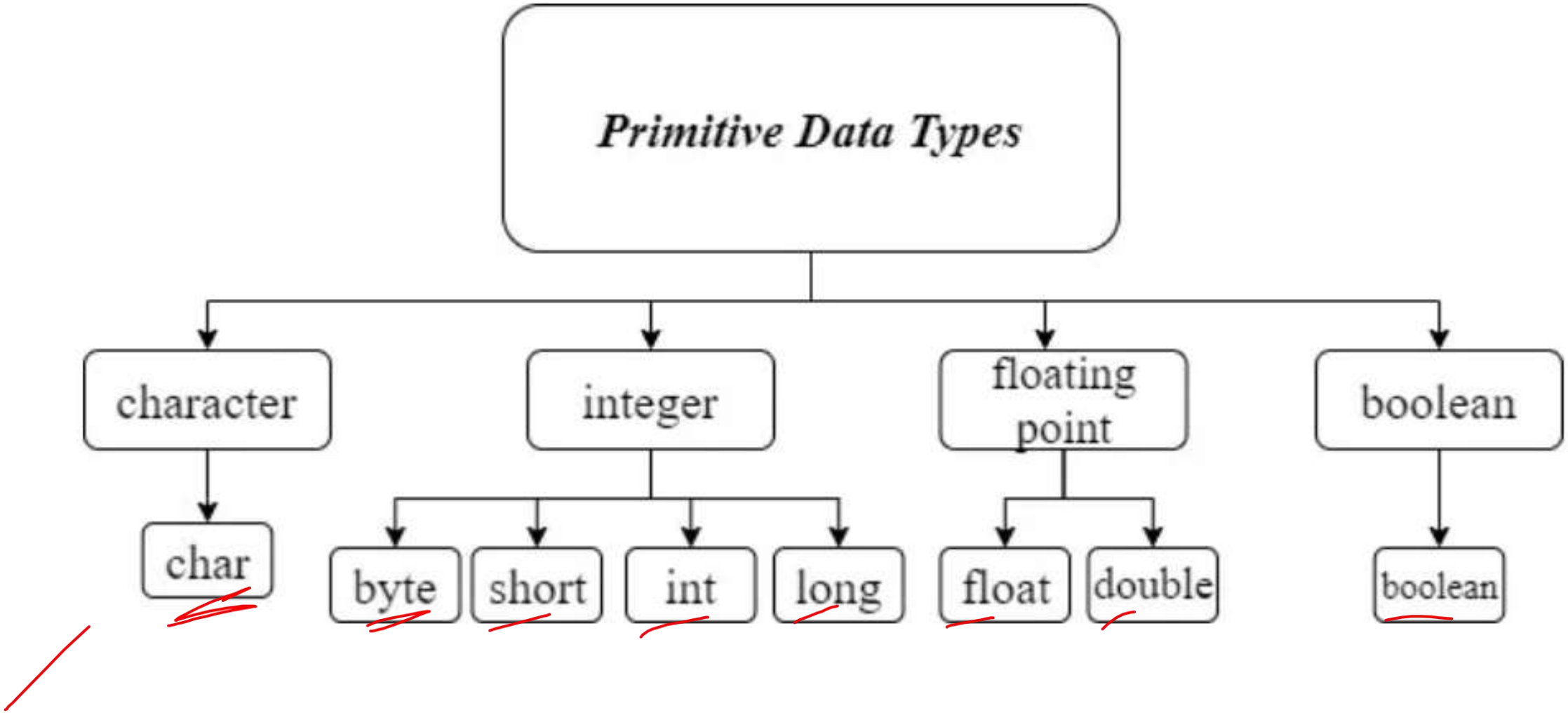
- There are 50 keywords currently defined in the Java language.
- These keywords cannot be used as names for a variable, class, or method.
- The keywords **const** and **goto** are reserved but not used.

| | | | | |
|---|---|---|---|---|
| abstract | continue | for | new | switch |
| assert | default | goto | package | synchronized |
| boolean | do | if | private | this |
| break | double | implements | protected | throw |
| byte | else | import | public | throws |
| case | enum | instanceof | return | transient |
| catch | extends | int | short | try |
| char | final | interface | static | void |
| class | finally | long | strictfp | volatile |
| const | float | native | super | while |

**TABLE 2-1** Java Keywords

# Java Is a Strongly Typed Language

- A part of Java's safety and robustness comes from the fact that it strongly typed.
- Every variable has a type, every expression has a type, and every type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible.
- Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.




created by Dr. Ritu Jain

# Data Types in Java

**Primitive data types:** *Represent single values – not objects*

- *Eight* primitive data types include
  - **Integer**: all integers (byte, short, int, long) are signed
    - byte – 8 bits
    - short – 16 bits
    - int – 32 bits
    - long – 64 bits
  - **Floating point numbers**:
    - float - 32 bits
    - double – 64 bits
  - **Characters**: char – 16 bits
  - **Boolean:** boolean

# Primitive Data Types in Java

| DATA TYPES | SIZE | DEFAULT | EXPLAINATION |
|---|---|---|---|
| boolean | 1 bit | false | Stores true or false values |
| byte | 1 byte/ 8bits | 0 | Stores whole numbers from -128 to 127 |
| short | 2 bytes/ 16bits | 0 | Stores whole numbers from -32,768 to 32,767 |
| int | 4 bytes/ 32bits | 0 | Stores whole numbers from -2,147,483,648 to 2,147,483,647 |
| long | 8 bytes/ 64bits | 0L | Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| float | 4 bytes/ 32bits | 0.0f | Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits |
| double | 8 bytes/ 64bits | 0.0d | Stores fractional numbers. Sufficient for storing 15 decimal digits |
| char | 2 bytes/ 16bits | '\u0000' | Stores a single character/letter or ASCII values |

# Primitive Data Types

- **byte:**
  - To save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer.
  - When you are working with a stream of data from a network or a file.
  - When you are working with a raw binary data.
    - **Example:** byte a = 10; byte b = -20;
- **short:**
  - The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.
  - **Example:** short s = 1000; short r = -5000;

# Primitive Data Types

- **int:**
  - to control loops and to index arrays
  - You might think that using a **byte** or **short** would be more efficient than using an **int** in situations in which the larger range of an **int** is not needed, this may not be the case.
  - The reason is that when **byte** and **short** values are used with binary operator, they are *promoted* to **int** when the expression is evaluated.
- **long:**
  - is useful when an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed.

# Primitive Data Types

- **floating-point types:** There are two kinds of floating-point types for numbers having fractional components:
  - **float** represent single precision numbers
    - Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.
    - **Example:** float hightemp, lowtemp;
  - **double** represent double-precision numbers
    - when you need to *maintain accuracy over many iterative calculations or manipulating large-valued numbers*
    - All math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

created by Dr. Ritu Jain

# Primitive Data Types

- char: Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages.
- For this purpose, it requires 16 bits.
- The range of a **char** is 0 to 65,536.
- There are no negative **char**s.
- The standard set of characters known as ASCII still ranges from 0 to 127 as always.
- In this way, unicode supports global portability.
- Although **char** is designed to hold Unicode characters, it can also be thought of as an integer type on which you can perform arithmetic operations. For example, you can add two characters together, or increment the value of a character variable.

# Primitive Data Types

- **boolean data type**
  - The boolean data type is used to store only two possible values:
    - true
    - false
  - This is the type returned by all relational operators
  - ***Example*:** boolean one = false;

# Literals

- A constant value in Java is created by using a literal representation of it.
- **E.g:** 100, 98.6, 'X' "This is a test“
- Integer Literals: Any whole number value is an integer literal
  - **Decimal literal:** base 10 whole number.
    - Normal decimal value can not have leading zeros.
  - **Octal literal: base 8**
    - denoted by leading zero.
    - Range: 0-7
    - E.g.: 045
  - **Hexadecimal**: base16
    - denoted by leading zero-x (0x or 0X)
    - Range: 0-9, A-F
    - E.g: 0x9, 0XA4
  - **Binary:** base 2
    - Prefix the value with 0b or 0B.
    - E.g: 0b101

# Integer Literals

- **Long literal is expressed by appending l or L at the end of the literal.**
- E.g.: 9223372036854L
- An integer can be assigned to byte, short, or char as long as it is within the range.
- **underscore (_)** can be embedded in an integer, floating, char literal.
  - Doing it makes it easier to read larger integer literal.
  - When the literal is compiled, the underscores are discarded.

E.g.: int x = 123_456_789;
int y = 123__456___789;

Especially useful when encoding such things as telephone numbers, customer ID numbers, part numbers.

# Floating-Point Literals

- represent decimal values with a fractional component.
- They can be expressed in either
  - **Standard notation:** *whole number component . fractional component*
    - E.g.: 2.45, 3.1456
  - **Scientific notation:** floating point notation E a suffix specifies a power of 10 by which the number is to be multiplied.
    - E.g.: 6.022E23, 31459E-05, 2e+100
- Floating-point literals in Java default to **double** precision.
- To specify a **float** literal, you must append an *F* or *f* to the constant.

created by Dr. Ritu Jain

# boolean Literals

- There are only two logical values that a **boolean** value can have, **true** and **false**.
- The values of **true** and **false** do not convert into any numerical representation.
- The **true** literal in Java does not equal 1, nor does the **false** literal equal 0.
- In Java, they can only be assigned to variables declared as **boolean**, or used in expressions with boolean operators.

# character Literals

- Represented by Unicode character set.
- Can be manipulated with integer operations, such as addition, subtraction operators.
- E.g: ‘a’
- For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as ‘\’’ for the single-quote character itself and **‘\n’** for the newline character.

# String Literals

- A sequence of characters between a pair of double quotes.
- E.g: “Hello World”, “two\nlines”, “\”This is in quotes\”“

# Variable

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.

  Type identifier [= value][, identifier [= value]…];

- **Dynamic Initialization:**

  double c = Math.sqrt(a * a + b * b);

# Scope of a variable

- All variables have a scope, which defines their
  - visibility
  - lifetime
- **Visibility:**
  - A scope determines what objects are visible to other parts of your program.
  - Variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
  - Scopes can be nested.
    - The outer scope encloses the inner scope.
    - This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

# Scope of a variable

```
// Demonstrate block scope.
class Scope {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope
     int y = 20; // known only to this block

      // x and y both known here.
      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

# Lifetime of a variable

- Variables are created when their scope is entered, and destroyed when their scope is left.
- This means that a variable will not hold its value once it has gone out of scope.
- Thus, the lifetime of a variable is confined to its scope.

# Lifetime of a variable

```
// Demonstrate lifetime of a variable.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

# Two variables with same names

- Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

```
class ScopeErr
{
        public static void main(String args[])
        {
           int bar = 1;
            {                   // new scope
              int bar = 2; // Compile-time error – bar already defined!
            }
        }
}
```

# Type Conversion

- Type conversion in Java refers to the process of converting data from one data type to another.
- Java supports two main types of conversions:
  - Implicit Type Conversion
  - Explicit Type Conversion

# Automatic Type Conversion

## Java's Automatic Conversions

- When one type of data is assigned to another type of variable, an *automatic type conversion (widening conversion )* will take place if:
  - The two types are compatible.
  - The destination type is larger than the source type.


- For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.
- However, there are no automatic conversions from the numeric types to **char or boolean**.
- **char** and **boolean** are not compatible with each other.
- As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

# Guess the Output?

```
class IntTochar{
        public static void main(String [] args){
                int i=10;
                char c=i;
                System.out.println(c);
        }

}
```

# Narrowing or Explicit Conversion

- If we want to assign a value of larger data type to a smaller data type we have to perform explicit **type casting or narrowing**.
- This is useful for incompatible data types where automatic conversion cannot be done.


- A *cast* is simply an explicit type conversion. It has this general form:

  (*target-type*) *value*

- Here, *target-type* specifies the desired type to convert the specified value to.
- char and number are not compatible with each other.

# Java program to illustrate explicit type conversion

```
//Java program to illustrate explicit type conversion
class Test
{       public static void main(String[] args)
        {
                double d = 100.04;

                long l = (long)d;                       //explicit type casting

                int i = (int)l;                 //explicit type casting
                System.out.println("Double value "+d);
                System.out.println("Long value "+l);       //fractional part lost
                System.out.println("Int value "+i);          //fractional part lost
        }
}
```

# Type promotion in Expressions

- While evaluating expressions, conditions for type promotion are:
  - Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
  - If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

  - What would be the output?
    - byte b = 50;
    - b = b * 2;

# Type promotion in Expressions

```
//Java program to illustrate Type promotion in Expressions
class Test
{
        public static void main(String args[])
        {
                byte b = 42;
                char c = 'a';
                short s = 1024;
                int i = 50000;
                float f = 5.67f;
                double d = .1234;

                double result = (f * b) + (i / c) - (d * s);
                System.out.println("result = " + result);
        }
}
```

# Format Specifiers

%d :  for all integral types

%f: for all floating point types

%c: for characters

%s : for string

%b : for boolean

```
class format{
  public static void main(String args[])
    {
        int i =5;
        double d = 55.5;
        char c = 'a';
        String st = "Hello";
        boolean b_var =false;

       System.out.printf("%d  ",  i);
       System.out.printf(" \n %c  %s", c, st);
       System.out.printf("\n %f  ", d);
       System.out.printf(“%b”, b_var);
   }
}
```

# Operators in java

- **Operator** in java is a symbol that is used to perform operations.
  - Arithmetic Operator
  - Bitwise Operator
  - Relational Operator
  - Boolean Logical Operator
  - Assignment Operator
  - ? Operator

# Arithmetic Operator

| Operator | Result |
|---|---|
| + | Addition |
| – | Subtraction (also unary minus) |
| * | Multiplication |
| / | Division |
| % | Modulus |
| ++ | Increment |
| += | Addition assignment |
| –= | Subtraction assignment |
| *= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulus assignment |
| – – | Decrement |

- The operands of the arithmetic operators must be of a numeric type.
- You can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.
- When the division operator is applied to an integer type, there will be no fractional component attached to the result.

```
// Demonstrate the basic arithmetic operators.
class BasicMath {
public static void main(String args[]) {
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.printf("a = %d, b = %d, c= %d, d = %d,  e= %d“, a, b, c, d, e);
        System.out.println("\nFloating Point Arithmetic");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - a;
        double de = -dd;
        System.out.printf("a = %f, b = %f, c= %f, d = %f,  e= %f“, da, db, dc, dd, de);
        }
}
```

# The Modulus Operator

It can be applied to floating-point types as well as integer types.

```
// Demonstrate the % operator.
class Modulus
{
        public static void main(String args[])
         {
                int x = 42;
                double y = 42.25;
                System.out.println("x mod 10 = " + x % 10);
                System.out.println("y mod 10 = " + y % 10);
        }
}
```

# Compound assignment operators

- Java provides special operators that can be used to combine an arithmetic operation with an assignment.

  a = a + 4; can be rewritten as a += 4;

  a = a % 4; can be rewritten as a %= 4;

  *var = var op expression; can be rewritten as*

  *var op= expression;*

- The compound assignment operators provide two benefits:
  - First, they save you a bit of typing,
  - Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

```
// Demonstrate several assignment operators.
class OpEquals
{
        public static void main(String args[])
        {
                int a = 1;
                int b = 2;
                int c = 3;
                a += 5;
                b *= 4;
                c += a * b;
                c %= 6;
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
        }
}
```

# Increment and Decrement

- The increment operator increases its operand by one.
- The decrement operator decreases its operand by one.
- x = x + 1; can be rewritten as x++;
- x = x - 1 is equivalent to x--

# Increment and Decrement Operator

- In the **prefix form**, the operand is incremented or decremented before the value is obtained for use in the expression.
- In **postfix form**, the previous value is obtained for use in the expression, and then the operand is modified.
- For example: x = 42;

  y = ++x; // y= 43 x = 43
- when written like this,

  x = 42;

  y = x++; // y = 42, x= 43

```
class IncDec {
   public static void main(String args[])
   {
            int a = 1;
            int b = 2;
            int c;
            int d;
            c = ++b;
            d = a++;
            c++;
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("c = " + c);
            System.out.println("d = " + d);
   }
}
```

# Bitwise Operators

- Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**.
- These operators act upon the individual bits of their operands

| Operator | Result |
|---|---|
| ~ | Bitwise unary NOT |
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |
| >> | Shift right |
| >>> | Shift right zero fill |
| << | Shift left |
| &= | Bitwise AND assignment |
| \|= | Bitwise OR assignment |
| ^= | Bitwise exclusive OR assignment |
| >>= | Shift right assignment |
| >>>= | Shift right zero fill assignment |
| <<= | Shift left assignment |

# Bitwise Logical Operator

| A | B | A \| B | A & B | A ^ B | ~A |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

# Bitwise NOT operator (~)

- unary operator
- inverts all the bits of its operand.
- For example, the number 42, which has the following bit pattern:
- 00101010 becomes 11010101 after the NOT operator is applied.

# Bitwise AND operator

- produces a 1 bit if both operands are 1.
- A zero is produced in all other cases.
- Here is an example:

```
  00101010   42
& 00001111   15
  00001010   10
```

# Bitwise OR (|)

- If either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
  00101010   42
| 00001111   15
  00101111   47
```

# Bitwise XOR (^)

- XOR operator, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

```
  00101010   42
^ 00001111   15
  00100101   37
```

```
// Demonstrate the bitwise logical operators.
class BitLogic
{   public static void main(String args[])
    {
      String binary[] = {"0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111","1000", "1001", "1010",
                         "1011", "1100",  "1101", "1110", "1111"};
      int a = 3;                  // 0011 (3) in binary
      int   b = 6;                //  0110  (6) in binary
      int c = a | b;
      int d = a & b;
      int e = a ^ b;
      int f = ~a & 0x0f;
      System.out.println(" a = " + binary[a]);
      System.out.println(" b = " + binary[b]);
      System.out.println(" a|b = " + binary[c]);
      System.out.println(" a&b = " + binary[d]);
      System.out.println(" a^b = " + binary[e]);
      System.out.println(" ~a & 0x0f = " + binary[g]);
    } }
```



O/P:

a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a & 0x0f = 1100

# Left Shift

- The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.

  *Syntax:* *value << num*

- Here, *num* specifies the number of positions to left-shift the value in *value*.

  *E.g.:* byte a =64;

```
int i;
i = a<<2;   // i=256
```

- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

  E.g.: b= (byte)(a<<2);

```
system.out.println("b= "+ b);          //b =0
```

- This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position 31.
- Each left shift doubles the original value (multiply by 2).

# Left Shift

- Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values, as **byte** and **short** values are promoted to **int** when an expression is evaluated.
- Furthermore, the result of such an expression is also an **int**. This means that the outcome of a left shift on a **byte** or **short** value will be an **int**, and the bits shifted left will not be lost until they shift past bit position 31.
-

```
// Left shifting a byte value.
class ByteShift
{
    public static void main(String args[])
    {
            byte a = 64, b;
            int i;
             i = a << 2;
             b = (byte) (a << 2);
             System.out.println("Original value of a: " + a);
             System.out.println("i and b: " + i + " " + b);
    }
}
```

# Right Shift

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.

  *syntax:* *value >> num*

- Here, *num* specifies the number of positions to right-shift the value in *value*.

e.g1:

```
int a = 32;
a = a >> 2; // a now contains 8
```

e.g2:

```
int a = 35;
a = a >> 2; // a still contains 8
```

- Each time you shift a value to the right, it divides that value by two—and discards any remainder.

created by Dr. Ritu Jain

# Right Shift

- When you are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when you shift them right.
- For example, –8 >> 1 is –4, which, in binary, is

11111000 –8

>>1

11111100 –4

# Unsigned Right Shift

- Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

int a = -1;

a = a >>> 24;

Here is the same operation in binary form:

11111111 11111111 11111111 11111111  –1 in binary

>>>24

00000000 00000000 00000000 11111111  255 in binary

# Relational Operators

| Operator | Result |
|---|---|
| == | Equal to |
| != | Not equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |

Note: The outcome of these operations is **boolean** value.

# Relational Operators

- The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.
- Only integer, floating-point, and character operands may be compared to see which is greater or less than the other.
- Any type in Java, including integers, floating-point numbers, characters, and boolean can be compared using the equality test, ==, and the inequality test, **!**=.

# Boolean Logical Operators

| Operator | Result |
|---|---|
| & | Logical AND |
| \| | Logical OR |
| ^ | Logical XOR (exclusive OR) |
| \|\| | Short-circuit OR |
| && | Short-circuit AND |
| ! | Logical unary NOT |
| &= | AND assignment |
| \|= | OR assignment |
| ^= | XOR assignment |
| == | Equal to |
| != | Not equal to |
| ?: | Ternary if-then-else |

- The Boolean logical operators operate on **boolean** operands.
- Result will be **boolean** value.

# Boolean Logical Operators

| A | B | A \| B | A & B | A ^ B | !A |
|---|---|---|---|---|---|
| False | False | False | False | False | True |
| True | False | True | False | True | False |
| False | True | True | False | True | True |
| True | True | True | True | False | False |

created by Dr. Ritu Jain

```
// Demonstrate the boolean logical operators.
class BoolLogic
{
    public static void main(String args[])
    {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" a|b = " + c);
        System.out.println(" a&b = " + d);
        System.out.println(" a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println(" !a = " + g);
    }
}
```

# Short-Circuit Logical Operators

- If you use the || and **&&** forms, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.
- if (denom != 0 && num / denom > 10)
- If this line of code were written using the single **&** version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.
- Use simple Logical operator in the following case:

```
if(c==1 & e++ < 100) d = 100;
```

- Here, using a single **&** ensures that the increment operation will be applied to **e** whether **c** is equal to 1 or not.

# The Assignment Operator

- **The assignment operator is the single equal sign, =.**
- It has this general form:

  var = expression;

- Here, the type of var must be compatible with the type of expression.

**Chain of assignments:**

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

- The = is an operator that yields the value of the right-hand expression.
- The value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**.

# The ? Operator

- Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements.
- *Syntax:*

  *expression1* **?** *expression2* **:** *expression3*

- *expression1 can be any expression that evaluates to a* ***boolean*** *value.*
- If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated.
- Both *expression2* and *expression3* are required to return the same type, which can't be **void**.
- E.g.:

```
ratio = denom == 0 ? 0 : num / denom;
```

# The ? Operator

- ratio = denom == 0 ? 0 : num / denom;
- When Java evaluates this assignment expression, it first looks at the expression to the *left* of the question mark.
- If **denom** equals zero, then the expression *between* the question mark and the colon is evaluated and used as the value of the entire **?** expression.
- If **denom** does not equal zero, then the expression *after* the colon is evaluated and used for the value of the entire **?** expression.
- The result produced by the **?** operator is then assigned to **ratio**.

# The Precedence of Java Operators

| **Highest** | | | | | | |
|---|---|---|---|---|---|---|
| ++(Postfix) | --(Postfix) | | | | | |
| ++(Prefix) | --(Prefix) | ~ | ! | + (Unary) | - (Unary) | (Typecast) |
| * | / | % | | | | |
| + | - | | | | | |
| >> | >>> | << | | | | |
| > | >= | < | <= | instanceof() | | |
| != | == | | | | | |
| & | | | | | | |
| ^ | | | | | | |
| \| | | | | | | |
| && | | | | | | |
| \|\| | | | | | | |
| ?: | | | | | | |
| -> | | | | | | |
| = | Op= | | | | | |
| **Lowest** | | | | | | |

# Operator Precedence

- **Precedence:** Operators with higher precedence are evaluated before operators with relatively lower precedence.
  - Operators in same row are equal in precedence.
- **Associativity:** When operators of equal precedence appear in the same expression, associativity governs the order of evaluation.
  - Associativity for unary operators, ternary operator, and assignment operator is right to left. For all other operators, it is left to right.
- Separators [], (), . have highest precedence.

# Reading Input from User using Scanner class

- To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:
  - Scanner in = **new** Scanner(System.in);

# Methods of Scanner Class

| | | |
|---|---|---|
| boolean | nextBoolean() | It scans the next token of the input into a boolean value and returns that value. |
| byte | nextByte() | It scans the next token of the input as a byte. |
| double | nextDouble() | It scans the next token of the input as a double. |
| float | nextFloat() | It scans the next token of the input as a float. |
| int | nextInt() | It scans the next token of the input as an Int. |
| String | nextLine() | This method can read the input till the end of line. |
| long | nextLong() | It scans the next token of the input as a long. |
| short | nextShort() | It scans the next token of the input as a short. |
| String | next() | This method can read the input only until a space(" ") is encountered. |

# User Input using Scanner class

- **Note:** When you use nextInt(), nextFloat(), or nextDouble(), these methods **do not consume the newline character** (\n) left in the buffer after pressing Enter.
- So when you call sc.nextLine() right after those, it reads the leftover newline and gives you an **empty string**.
- **Solution:** Add an extra sc.nextLine() before reading the line.
- Refer UserInput3.java

# Control Statements

- Control statements are divided into three groups:
  1. Selection ( if and switch)
  2. Iteration (do while, while, for, for each)
  3. Jump ( break and continue)

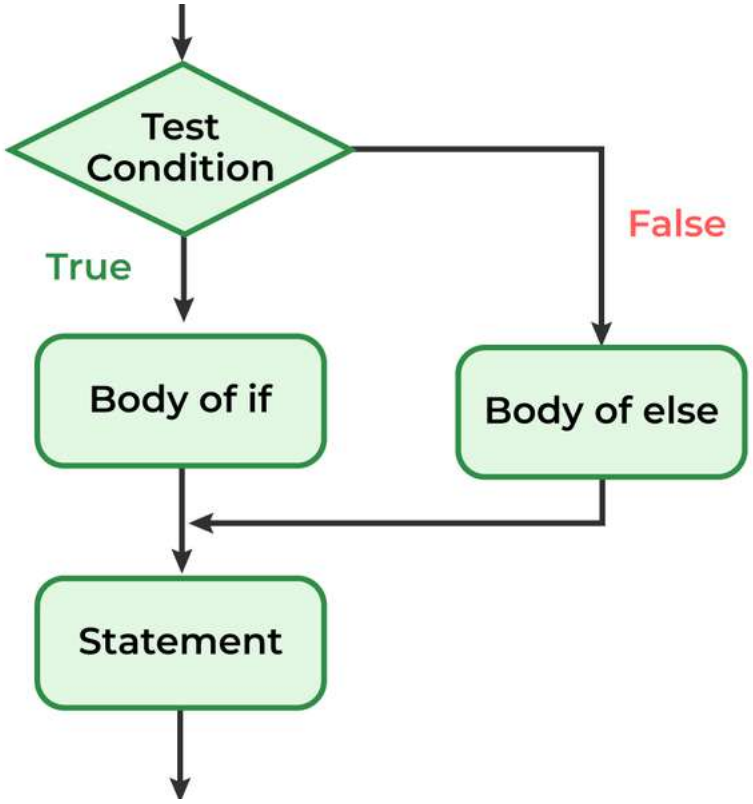
# Java Selection Statements

- Java supports two selection statements:
  - **if**
  - **switch**.
- These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

# if statements

- The **if** statement is Java's conditional branch statement.
  1) if-else
  2) if-else-if
  3) Nested if else

# if else




```
if (condition)
{
        // Executes this block if condition is true
}
 else {
        // Executes this block if condition is false
 }
```

- Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*).
- The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

# Example of if else

```
//Java Program Using if-else to Check if a Number is Positive or Negative
public class IfElseExample {
    public static void main(String[] args) {
        int number = -10;  // You can change this value

        if (number > 0) {
            System.out.println(number + " is positive.");
        } else {
            System.out.println(number + " is negative or zero.");
        }
    }
}
```

# Nested if else

- **Nested if:** refers to having one if statement inside another if statement. If the outer condition is true the inner conditions are checked and executed accordingly.

```
if(condition)
{
    if(condition)
        statement;
}
else
    statement;

statement 2;
```

Nested if

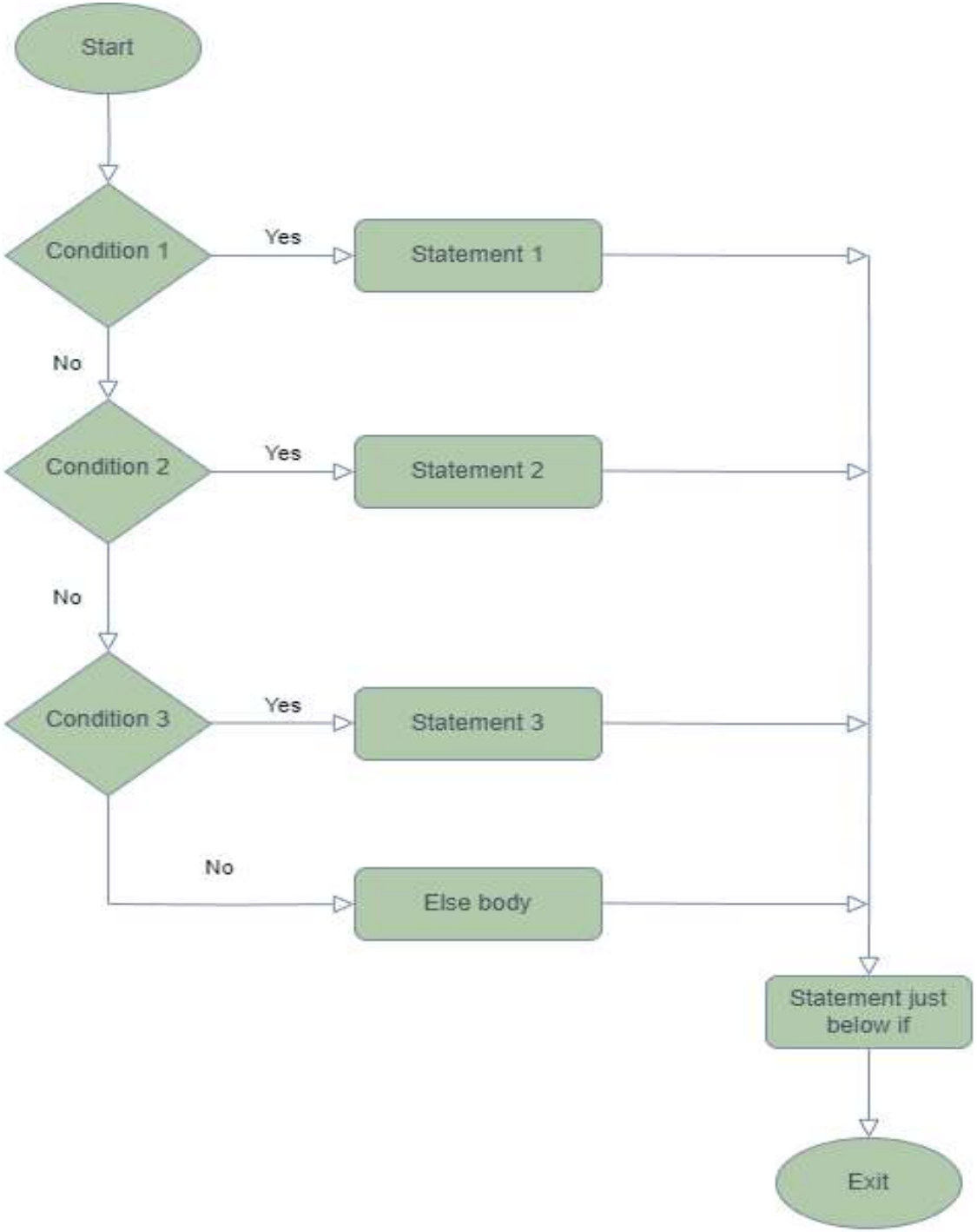
# // Java Program to implement Nested if statement

```
//WAP to check eligibility for donating blood.
public class NestedIf {
    public static void main(String[] args) {
        int age = 25;
        double w = 65.5;
        if (age >= 18) {
            if (w >= 50.0) {
                System.out.println("You are eligible to donate blood.");
            } else {
                System.out.println("You must weigh at least 50 kilograms to donate blood.");
            }
        } else {
            System.out.println("You must be at least 18 years old to donate blood.");
        } //end of else
    } //end of main
} // end of class
```

# if-else-if Ladder

```
if(condition)
        statement;
else if(condition)
        statement;
else if(condition)
        statement;
...
else
        statement;
```

# If else if ladder



# if-else-if Ladder

- The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final **else** statement will be executed.
- The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed.
- If there is no final **else** and all other conditions are **false**, then no action will take place.

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

# switch statement

The **switch** statement provides multiway branching and is often a better alternative to nested **if-else-if** statements. The general form of the switch statement is :

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    ...
    case valueN :
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

**Note:** The expression must be of type **byte**, **short**, **int, char or String**

# Switch

- case statements must be of a type compatible with the expression.
- Each case value must be a literal.
- Duplicate case values not allowed.
- When a match is found, the set of statements following that **case** statement is executed. If none of the values match, then the default statement is executed.
- Note however, that the **default** statement is optional.
- **break** statement: to terminate the statement block
- The **break** statement is optional. If you do not use the **break** statement, execution continues with the next **case**.

created by Dr. Ritu Jain

# switch

- The **switch** statement differs from **if** statement in that, the switch can test only for equality, whereas **if** can evaluate any type of **Boolean** expression.
- No two case constants in the same **switch** can have identical values.
- A switch statement is generally more efficient than a set of nested if statements.

```
// A simple example of the switch.
class SampleSwitch {
  public static void main(String args[]) {
    for(int i=0; i<6; i++)
      switch(i) {
        case 0:
          System.out.println("i is zero.");
          break;
        case 1:
          System.out.println("i is one.");
          break;
        case 2:
          System.out.println("i is two.");
          break;
        case 3:
          System.out.println("i is three.");
          break;
        default:
          System.out.println("i is greater than 3.");
      }
  }
}
```

# Iteration Statements

- Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.
- Java provides three iteration statements:
    1) while
    2) do-while
    3) for

# while loop

- The general form of the **while** is:

```
while(condition) {
 body of the loop
 }
```

- Entry controlled loop
- The body of the **while** can be empty:
  - while(condition);

# Ex: while loop

```
public class WhileEx {
        public static void main(String[] args) {
                int i=1;
                while(i<=10){
                        System.out.println(i);
                        i++;
                }
        }
}
```

# do while loop

The general form of the **do-while** is :

```
do {
body of the loop
} while(condition);
```

- Exit controlled loop
- always executes its body at least once.
- useful in menu selection, where the body of the menu loop should be executed at least once.

# do while Example

```
public class DoWhileEx {
public static void main(String[] args) {
        int i=1;
        do{
                System.out.println(i);
                i++;
            }while(i<=10);
        }
}
```

# for loop

```
for(initialisation; condition; iteration) {
//body of the loop
}
```

initialization—>condition—>Body—>Iteration—>condition

created by Dr. Ritu Jain

# Variations in for loop

- for(;;) {}
- for(*initialisation*; *condition*; *iteration*);
- Comma operator:
  - for(a = 1, b = 6; a<b; a++,b--) {}

# Example of for loop

```
class ForLoop {
        public static void main(String args[ ]) {
                int a, b;
                for(a = 1, b = 4; a<b; a++,b--) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                }
        }
}
```

# Nested for loop

```
public class NestedForEx {
   public static void main(String[] args) {
         for(int i=1;i<=3;i++){
                for(int j=1;j<=3;j++){
                      System.out.println(i+" "+j);
                }
         }
   }
}
```

# for each loop

- The for-each loop is used to traverse array or collections in Java.
- It is easier to use than simple for loop because we don't need to increment value and use subscript notation.
- It works on the basis of elements and not the index.
- It returns element one by one in the defined variable.

# For each loop

- The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*)
*statement-block*

```
// The for-each loop is essentially read-only.
class NoChange {
	public static void main(String args[])
	{
		int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
		for(int x : nums)
		{
			x = x * 10; // no effect on nums
			System.out.print(x + " ");

		}
		System.out.println();
		for(int x : nums)
			System.out.print(x + " ");
		System.out.println();
	}
}
```

```
10 20 30 40 50 60 70 80 90 100
1 2 3 4 5 6 7 8 9 10
```

# Break

- Break can be used to:
  - terminates a statement sequence in a **switch** statement.
  - can be used to exit a loop.
  - can be used as a "civilized" form of goto.

- When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- **break** was not designed to provide the normal means by which a loop is terminated. The loop's conditional expression serves this purpose.
- The **break** statement should be used to cancel a loop only when some sort of special situation occurs.

# Break to exit from a loop

```
// Using break to exit a loop.
class BreakLoop
{
    public static void main(String args[])
    {
       for(int i=0; i<100; i++)
          {
             if(i == 10) break; // terminate loop if i is 10
             System.out.println("i: " + i);
          }
        System.out.println("Loop complete.");
    }
}
```

# break in nested loop

- When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.

```
// Using break with nested loops.
class BreakLoop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

# Labelled Break: Civilized form of goto

- By using labelled **break**, you can break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block.
- Further, you can specify precisely where execution will resume, because this form of **break** works with a label.
- The general form of the labeled **break** statement is shown here:

  break *label*;

- Most often, *label* is the name of a label that identifies a block of code.

# Labelled break

- When this form of **break** executes, control is transferred out of the named block.
- The labeled block must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled **break** statement to exit from a set of nested blocks.
- But you cannot use **break** to transfer control out of a block that does not enclose the **break** statement.
- To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon.
- Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block.

# Labelled Break: Civilized form of goto

```
class Break             // Using break as a civilized form of goto.
{       public static void main(String args[])
        {  boolean t = true;
           first: {
                second: {
                    third: {
                        System.out.println("Before the break.");
                        if(t) break second;     // break out of second block
                            System.out.println("This won't execute");
                        }
                    System.out.println("This won't execute");
                }
            System.out.println("This is after second block.");
            }
        }
}
```

```
// Using break to exit from nested loops
class BreakLoop4 {
  public static void main(String args[]) {
    outer: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break outer; // exit both loops
        System.out.print(j + " ");
      }
      System.out.println("This will not print");
    }
    System.out.println("Loops complete.");
  }
}
```

```
// This program contains an error.
class BreakErr {
  public static void main(String args[]) {

    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }

    for(int j=0; j<100; j++) {
      if(j == 10) break one; // WRONG
      System.out.print(j + " ");
    }
  }
}
```

- In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for** statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

# continue

```
// Demonstrate continue.
class Continue {
  public static void main(String args[]) {
    for(int i=0; i<10; i++) {
      System.out.print(i + " ");
      if (i%2 == 0) continue;
      System.out.println("");
    }
  }
}
```

# Labelled Continue

- As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue.

```
class ContinueLabel
{  public static void main(String args[])
    {
        outer: for (int i=0; i<10; i++)
          {  for(int j=0; j<10; j++)
             {
                if(j > i)
                 {
                    System.out.println();
                    continue outer;
                 }
                System.out.print(" " + (i * j));
             }
          }
        System.out.println();
    }
}
```

# Labelled Continue

**Output of the program ContinueLabel.java:**

0

0 1

0 2 4

0 3 6 9

0 4 8 12 16

0 5 10 15 20 25

0 6 12 18 24 30 36

0 7 14 21 28 35 42 49

0 8 16 24 32 40 48 56 64

0 9 18 27 36 45 54 63 72 81

# return

- The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.
- At any time in a method, the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed.

# Example of return statement

```
public class ReturnExample {
    // Method that adds two numbers and returns the result
    int add(int a, int b) {
        return a + b; // RETURNS the sum to the caller
    }

    public static void main(String[] args) {
        ReturnExample obj = new ReturnExample();
        int result = obj.add(5, 7);
        System.out.println("Sum: " + result); // Output: Sum: 12
    }
}
```

# Array, String, StringBuffer

# Arrays

- An *array* is a finite, ordered collection of homogenous elements.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

# One-Dimensional Arrays

- A *one-dimensional array* is, essentially, a list of like-typed variables.
- *Syntax:* *type array_var*[ ];
  *array_var* = new *type*[*size*];
- Here, *type* determines what type of data the array will hold.
- E.g.:int month_days[]; // **month_days** is set to **null**, it represents an array with no value.
  month_days = new int[12]; //this statement link **month_days** with an actual, physical
  // array of integers with the help of new operator.
- **new** is a special operator that allocates memory.

```
// Demonstrate a one-dimensional array.
class Array {
  public static void main(String args[]) {
    int month_days[];
    month_days = new int[12];
    month_days[0] = 31;
    month_days[1] = 28;
    month_days[2] = 31;
    month_days[3] = 30;
    month_days[4] = 31;
    month_days[5] = 30;
    month_days[6] = 31;
    month_days[7] = 31;
    month_days[8] = 30;
    month_days[9] = 31;
    month_days[10] = 30;
    month_days[11] = 31;
    System.out.println("April has " + month_days[3] + " days.");
  }
}
```

# Initialization of Arrays

- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types.
- An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer.
- There is no need to use **new**.
- Eg: int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

# Example: Initialization of One-Dimensional Arrays

```
// Demonstrate a one-dimensional array.
class AutoArray
{
   public static void main(String args[])
   {
     int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
     System.out.println("April has " + month_days[3] + " days.");
   }
}
```

# Example: One-Dimensional Arrays

```
// Average an array of values.
class Average
{
   public static void main(String args[])
   {
      double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
      double result = 0;
       int i;
       for(i=0; i<5; i++)
           result = result + nums[i];
       System.out.println("Average is " + result / 5);
    }
}
```

# ArrayIndexOutOfBoundsException

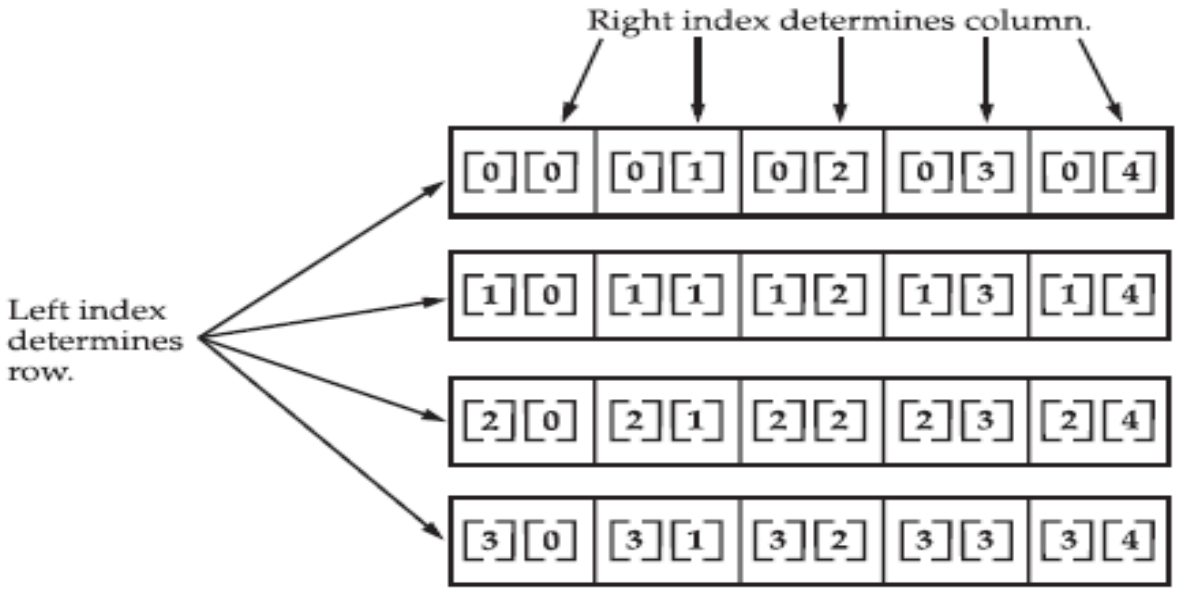
- The Java run-time system will check that all array indexes are in the correct range.
- For example, the runtime system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive.
- If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

# Multidimensional Arrays

- In Java, *multidimensional arrays* are actually arrays of arrays.

```
int twoD[][] = new int[4][5];
```

- This allocates a 4 by 5 array and assigns it to **twoD**.
- Internally this matrix is implemented as an *array* of *arrays* of **int**.




Given: int twoD [ ] [ ] = new int [4] [5] ;

**FIGURE 3-1** A conceptual view of a 4 by 5, two-dimensional array

# Example: 2-D Array

```
// Demonstrate a two-dimensional array.
class TwoDArray
 {   public static void main(String args[])
     {       int twoD[][]= new int[4][5];
             int i, j, k = 0;
             for(i=0; i<4; i++)
                         for(j=0; j<5; j++)
                          {
                                     twoD[i][j] = k;
                                     k++;
                         }
                         for(i=0; i<4; i++)
                         {     for(j=0; j<5; j++)
                                     System.out.print(twoD[i][j] + " ");
                               System.out.println();
                         }
     }
}
```

# Multidimensional Arrays

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (**leftmost**) dimension.
- You can allocate the remaining dimensions separately.
- For example,

```
int twoD[][] = new int[4][];

twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

# Multidimensional Arrays: Jagged Array

- When you allocate dimensions manually, *you do not need to allocate the same number of elements for each dimension*.
- Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.
- If you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.
- In Java, a **Jagged array** is a multidimensional array where each row can have a different number of columns.

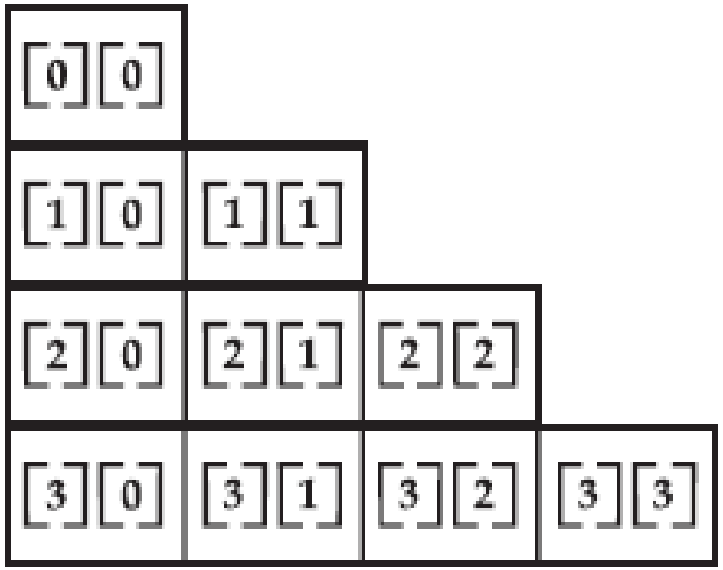
# Initialization of Multi-Dimensional Arrays

- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.
- Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.
class Matrix {
  public static void main(String args[]) {
    double m[][] = {
      { 0*0, 1*0, 2*0, 3*0 },
      { 0*1, 1*1, 2*1, 3*1 },
      { 0*2, 1*2, 2*2, 3*2 },
      { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;

    for(i=0; i<4; i++) {
      for(j=0; j<4; j++)
        System.out.print(m[i][j] + " ");
      System.out.println();
    }
  }
}
```

```
// Manually allocate differing size second dimensions.
class TwoDAgain
{
        public static void main(String args[])
        {
                int twoD[][] = new int[4][];
                twoD[0] = new int[1];
                twoD[1] = new int[2];
                twoD[2] = new int[3];
                twoD[3] = new int[4];
                int i, j, k = 0;
                for(i=0; i<4; i++)
                        for(j=0; j<i+1; j++)
                        {
                                twoD[i][j] = k;
                                k++;
                        }
                for(i=0; i<4; i++)
                {
                        for(j=0; j<i+1; j++)
                                System.out.print(twoD[i][j] + " ");
                        System.out.println();
                }
        }
}
```

- The array created by this program looks like this:




O/P:
0
1 2
3 4 5
6 7 8 9

# Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:

  *type*[ ] *var-name;*

- For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

- The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3; // create three arrays
```

- creates three array variables of type **int**. It is the same as writing

```
int nums[], nums2[], nums3[]; // create three arrays
```



- The alternative declaration form is also useful when specifying an array as a return type for a method.

# String Class

- Every string you create is actually an object of type **java.lang.String** class .
- It is basically an object that represents sequence of char values.
- Indexing starts from 0
- **How to Create a String:**
  - Using String literal:
    - Eg: *String s1 = "Hello";*
  - Using *new* keyword:
    - Eg1: String s2 = new String ("Hello");
    - Eg 2:
      - char[] ch={'j','a','v','a'};  //character array
      - String s3=new String(ch);
    - Eg 3:
      - String s4=new String(s3);

# String Object Creation By new keyword

```
public class StringExample{
public static void main(String args[]){
        String s1="java";
        char ch[]={'s','t','r','i','n','g','s'};
        String s2=new String(ch);
        String s3=new String("example");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
}}
```

# String Class

- String objects are **immutable**; once a String object is created, its contents cannot be altered.
- **Concatenation**: The compiler will convert an operand to its string equivalent whenever one of the operand of the + is an instance of String. Expression is evaluated left to right

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

# String Class

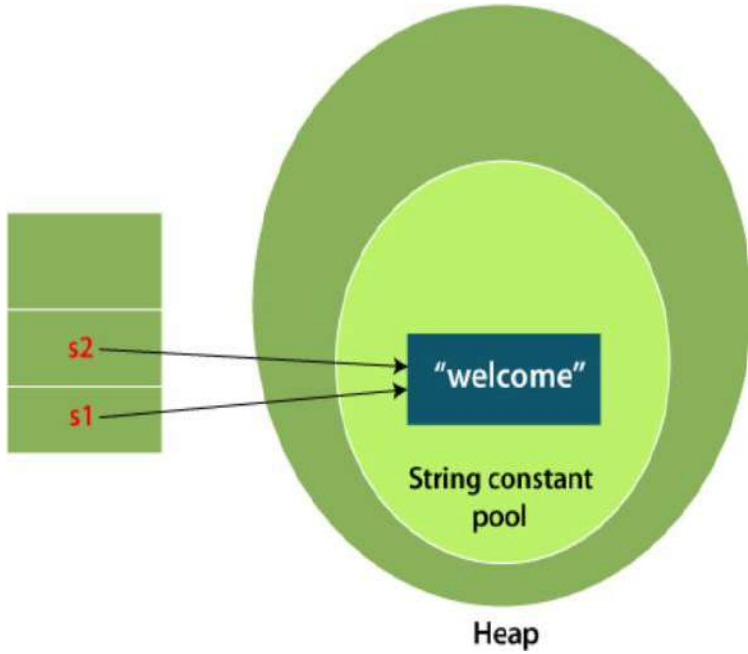
```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

The output produced by this program is shown here:

First String
Second String
First String and Second String

# String Literal

- Each time you create a string literal, JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:
  - String s1="Welcome";
  - String s2="Welcome";//It doesn't create a new instance
- Why Java uses the concept of String literal?
  - To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).
- String s=new String("Hello");
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Hello" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).



# Methods of String class

| Method | Description |
|---|---|
| int length() | It returns number of characters in the string |
| char charAt(int index) | It returns character at the specified index |
| int indexOf(int ch) | It returns the index position for the given char value |
| int indexOf(int ch, int fromIndex) | It returns the index position for the given char value and from index |
| int indexOf(String substring) | It returns the index position for the given substring |
| int indexOf(String substring, int fromIndex) | It returns the index position for the given substring and from index |
| lastIndexOf( ) | Searches for the last occurrence of a character or substring. |
| String substring(int beginIndex) | It returns substring starting from begin index. |
| String substring(int beginIndex, int endIndex) | It returns substring from beginIndex to endIndex-1 |
| boolean contains(CharSequence s) | It returns true or false after matching the sequence of char value. |
| String concat(String s) | concatenates (appends) string s to the end of invoking string object. |

# Methods of String class

| Method | Description |
|---|---|
| boolean equals(Object o) | It checks whether two string objects contain exactly same characters in same order. |
| boolean equalsIgnoreCase(String s) | compares two strings irrespective of the case (lower or upper) of the string |
| boolean isEmpty() | It checks if string is empty. |

# Methods of String class

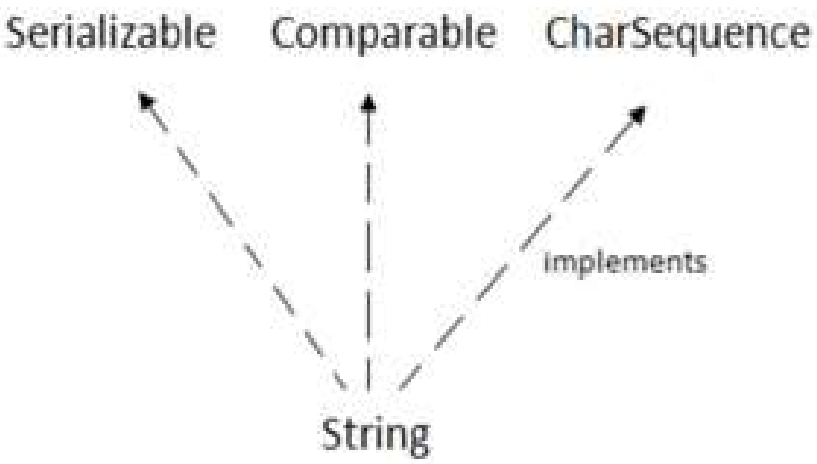
| Method | Description |
|---|---|
| String toLowerCase() | It returns a string in lowercase. |
| String toUpperCase() | It returns a string in uppercase. |
| int compareTo( String anotherString) | Compares two string lexicographically. |
| int compareToIgnoreCase( String anotherString) | Compares two string lexicographically, ignoring case considerations. |
| String trim() | Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle. |
| String replace (char oldChar, char newChar) | Returns new string by replacing all occurrences of *oldChar* with *newChar*. |
| char[] toCharArray(): | Converts this String to a new character array. |
| boolean startsWith(String s) | Return true if string starts with this prefix. |
| boolean endswith(String s) | Return true if string ends with this prefix. |

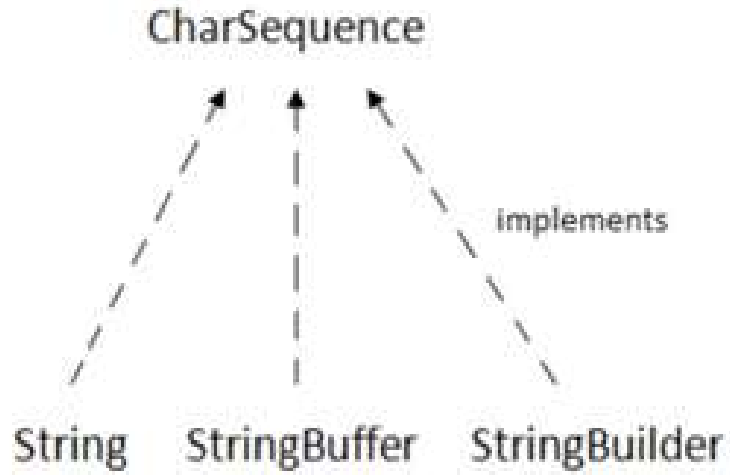
# length vs length() in Java

- **array.length:** length is a final variable applicable for array. With the help of the length variable, we can obtain the size of the array.
- **string.length() :** length() method is a final method which is applicable for string objects. The length() method returns the number of characters present in the string.

# String Class

- The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.
- **CharSequence Interface:** The CharSequence interface is used to represent the sequence of characters.
- **String, StringBuffer and StringBuilder** classes implement it. It means, we can create strings in Java by using these three classes.



# Some Methods of String class (cont'd)

- **int length()**: returns the number of characters in the String.
  - Eg: ”Hello".length();  // returns 5
- **Char charAt(int i)** : Returns the character at ith index.
  - Eg: ”Hello".charAt(3); // returns  ‘l’
- **String substring (int i)**: Return the substring from the ith  index character to end.
  - Eg: ”Hello”.substring(3); // returns “lo”
-

# Some Methods of String class (cont'd)

**String substring (int i, int j)**

Returns the substring from i to j-1 index.

```
"Hello".substring(2, 5); // returns “llo”
```

**5. String concat( String str)**

Concatenates specified string to the end of this string.

```
String s1 = ”Hello”;
String s2 = ”World”;
String output = s1.concat(s2); // returns “HelloWorld”
```

•

# Some Methods of String class (cont'd)

- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

If String s is not present in input string then -1 is returned as the default value.

```
1.  String s = "Learn Share Learn";
    int output = s.indexOf("Share"); // returns 6
2. String s = "Learn Share Learn"
   int output = s.indexOf("Play");  // return -1
```

- **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

```
String s = "Learn Share Learn";
int output = s.indexOf("ea",3);// returns 13
```

# Some Methods of String class (cont'd)

## int lastIndexOf( String s)

Returns the index within the string of the last occurrence of the specified string. If String s is not present in input string then -1 is returned as the default value.

```
1. String s = ”Learn Share Learn”;
      int output = s.lastIndexOf("a"); // returns 14
2. String s = "Learn Share Learn"
      int output = s.indexOf(“Play”);          // return -1
```

## boolean equals( Object otherObj)

Compares this string to the specified object.

```
Boolean out = “world”.equals(“world”); // returns true
Boolean out = “World”.equals(“world”); // returns false
```

•

# Some Methods of String class (cont'd)

**boolean equalsIgnoreCase (String anotherString)**

Compares string to another string, ignoring case considerations.

```
Boolean out= "World".equalsIgnoreCase("world"); // returns true
```

**int compareTo( String anotherString)**

Compares two string lexicographically.

```
int out = s1.compareTo(s2);
// where s1 and s2 are
// strings to be compared
```

This returns difference s1-s2. If :

```
out  <   0    // s1 comes before s2
out  =   0    // s1 and s2 are equal.
out  >   0     // s1 comes after s2.
```

# compareTo() method

- When we compare two strings in which either first or second string is empty, the method returns the length of the string. So, there may be two scenarios:
  - If **first** string is an empty string, the method returns a **negative**
  - If **second** string is an empty string, the method returns a **positive** number that is the length of the first string.

# CompareToExample

```
public class CompareToExample{
public static void main(String args[]){
        String s1="hello";
        String s2="hello";
        String s3="meklo";
        String s4="hemlo";
        String s5="flag";
        System.out.println(s1.compareTo(s2));//0 because both are equal
        System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
        System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
        System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
}}
```

# Example of Empty String in CompareTo()

```
public class CompareToExample2{
public static void main(String args[]){
        String s1="hello";
        String s2="";
        String s3="me";
        System.out.println(s1.compareTo(s2)); // 5
        System.out.println(s2.compareTo(s3));  //-2
}}
```

# Some Methods of String class (cont'd)

**int compareToIgnoreCase( String anotherString)**

Compares two string lexicographically, ignoring case considerations.

```
 int out = s1.compareToIgnoreCase(s2);
// where s1 and s2 are
// strings to be compared
 This returns difference s1-s2. If :
 out < 0  // s1 comes before s2
 out = 0   // s1 and s2 are equal.
 out > 0   // s1 comes after s2.
```

***Note:*** *In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).*

•

# Ex: compareToIgnoreCase( String anotherString)

```
public class CompareToIgnoreCase
{
   public static void main(String argvs[])
   {
      String string1 = "Book";
      String string2 = "book";
      String string3 = "look";
      String string4 = "abc";
      String string5 = "BEEN";

      System.out.println(string1.compareToIgnoreCase(string2));    //0
      System.out.println(string1.compareToIgnoreCase(string3));    //-10
      System.out.println(string1.compareToIgnoreCase(string4));    //1
      System.out.println(string1.compareToIgnoreCase(string5));    //10
   }
}
```

# Some Methods of String class (cont'd)

**String toLowerCase()**

Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;
String word3 = word1.toLowerCase(); // returns “hello"
```

**14. String toUpperCase()**

Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;
String word2 = word1.toUpperCase(); // returns “HELLO”
```

**15. String trim()**

Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;
String word2 = word1.trim(); // returns “Learn Share Learn”
```

•

# Some Methods of String class (cont'd)

- **String replace (char oldChar, char newChar)**
  - Returns new string by replacing all occurrences of *oldChar* with *newChar*.
  - String s1 = “fadaf“;
  - String s2 = s1.replace(‘f’ ,’r’); // return “radar”
  - ***Note:*** *s1 is still fadaf and s2 is radar*
- **boolean contains(String) :**
  - Returns true if string contains the given string
  - Eg: “abc”.contains(“ab”) // return true
- **Limitations of the contains() method**:
  - The contains() method should not be used to search for a character in a string. Doing so results in an error.
  - The contains() method only checks for the presence or absence of a string in another string. It never reveals at which index the searched index is found. Because of these limitations, it is better to use the indexOf() method instead of the contains() method

# Some Methods of String class (cont'd)

- **char[] toCharArray():**
  - Converts this String to a new character array.

```
String s1="hello";
char []ch=s1.toCharArray();   // returns [ 'h', 'e' , 'l' , 'l' , 'o' ]
for(int I=0;i<ch.length;i++)
   System.out.print(ch[i]);
```

# Some Methods of String class (cont'd)

**startsWith( String s) and endsWith( String s):**

- The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:
  - boolean startsWith(String *str*)
  - boolean endsWith(String *str*)
- Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,
- "Foobar".endsWith("bar")

  and
- "Foobar".startsWith("Foo") are both **true**.

# Example: Counting Frequency of a character in a String by Using the charAt() Method

```
public class CharAtExample5 {
    public static void main(String[] args) { String
        str = "Welcome to Java world";
        int count = 0;
        for (int i=0; i<=str.length()-1; i++) {
            if(str.charAt(i) == 'o') {
                count++;
            }
        }
        System.out.println("Frequency of o is: "+count);
    }
}
```

# Some Methods of String class (cont'd)

- **valueOf(): convert data into its string representation**
- **valueOf( )** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. valueOf( ) calls the **toString( )** method on the object.
- Eg:

```
int value=30;
String s1=String.valueOf(value);
System.out.println(s1+10);//concatenating string with 10   O/P: 3010
```

# StringBuffer

- **StringBuffer** supports a modifiable string. StringBuffer is a **mutable sequence of characters** — which means the **content can be changed after it's created**.
- **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences.
- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

# StringBuffer

- It is part of the java.lang package.
- **Why StringBuffer?**
  - In Java, the **String class is immutable**, meaning that every time you modify a string (like concatenation or replacement), a **new object** is created in memory.
    This is **inefficient** when you have to do **many string modifications** (e.g., in loops).
  - To solve this, Java provides **StringBuffer** (and later **StringBuilder**) which allows **modifying strings without creating new objects**.

# Key Points about StringBuffer

| Feature | Description |
| --- | --- |
| **Mutability** | The content of a StringBuffer object can be changed. |
| **Thread-safety** | StringBuffer is **synchronized**, meaning it is **safe to use in multithreaded environments**. |
| **Performance** | Slightly **slower than StringBuilder** because of synchronization overhead. |
| **Package** | Found in java.lang package. |
| **Default Capacity** | 16 characters (it expands automatically if needed). |

# StringBuffer

- **StringBuffer** defines these four constructors:
  - StringBuffer( )
  - StringBuffer(int *size*)
  - StringBuffer(String *str*)
- The default constructor (the one with no parameters) reserves room for 16
- characters without reallocation.
- The second version accepts an integer argument that explicitly sets the size of the buffer.
- The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
- **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place.

# Common Constructors of StringBuffer

- StringBuffer sb1 = new StringBuffer();  // Empty buffer with capacity 16
- StringBuffer sb2 = new StringBuffer("Hello");  // Initialized with "Hello"
- StringBuffer sb3 = new StringBuffer(50);  // Capacity 50

# Common Methods of StringBuffer

| Method | Description | Example |
|---|---|---|
| **append(String str)** | Adds text at the end | sb.append(" World"); |
| **insert(int index, String str)** | Inserts text at given index | sb.insert(5, " Java"); |
| **replace(int start, int end, String str)** | Replaces text between indexes | sb.replace(0, 5, "Hi"); |
| **delete(int start, int end)** | Deletes characters between indexes | sb.delete(5, 10); |
| **reverse()** | Reverses the content | sb.reverse(); |
| **capacity()** | Returns current capacity | sb.capacity(); |
| **length()** | Returns current length | sb.length(); |
| **toString()** | Converts to a normal String | String s = sb.toString(); |

# length( ) and capacity( )

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity can be found through the **capacity( )** method. They have the following general forms:

```
int length( )
int capacity( )
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

```
buffer = Hello
length = 5
capacity = 21
```

# ensureCapacity( )

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**. **ensureCapacity( )** has this general form:

void ensureCapacity(int *minCapacity*)

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

# setLength( )

To set the length of the string within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

void setLength(int *len*)

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength( )** with a value less than the current value returned by **length( )**, then the characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength( )** to shorten a **StringBuffer**.

# charAt( ) and setCharAt( )

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " + sb.charAt(1));

    sb.setCharAt(1, 'i');
    sb.setLength(2);
    System.out.println("buffer after = " + sb);
    System.out.println("charAt(1) after = " + sb.charAt(1));
  }
}
```

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

# getChars( )

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*.

The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

# append( )

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)

First, the string representation of each parameter is obtained. Then, the result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append( )**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

The output of this example is shown here:

```
a = 42!
```

# insert( )

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **String**s, **Object**s, and **CharSequence**s. Like **append( )**, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*)
StringBuffer insert(int *index*, char *ch*)
StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");

    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

```
I like Java!
```

## reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

# delete( ) and deleteCharAt( )

You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*)
StringBuffer deleteCharAt(int *loc*)

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.delete(4, 7);
    System.out.println("After delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

# replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

```
After replace: This was a test.
```

# substring( )

You can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

String substring(int *startIndex*)
String substring(int *startIndex*, int *endIndex*)

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for **String** that were described earlier.

# indexOf() and lastIndexOf()

```
class IndexOfDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("one two one");
    int i;

    i = sb.indexOf("one");
    System.out.println("First index: " + i);

    i = sb.lastIndexOf("one");
    System.out.println("Last index: " + i);
  }
}
```

# Example of StringBuffer

```
public class StringBufferExample {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        sb.append(" World");
        sb.insert(5, ",");
        sb.replace(0, 5, "Hi");
        sb.delete(3, 4);
        sb.reverse();

        System.out.println("Final String: " + sb);
        System.out.println("Length: " + sb.length());
        System.out.println("Capacity: " + sb.capacity());
    }
}
```

**Output:**
Final String: dlroW iH
Length: 8
Capacity: 21

# Real-Life Example: Using StringBuffer Instead of String

**Scenario:**

- Suppose you are developing a **student registration system** for a university. The program needs to **generate registration IDs** for hundreds of students by repeatedly concatenating names, courses, and roll numbers.
- If you use the **String** class, each concatenation will create a **new String object** (since Strings are immutable).
- This causes **high memory usage** and **slower performance**, especially in loops.
- That's where **StringBuffer** is better — because it is **mutable** and can modify the same object without creating new ones.

# Example Program: Student Registration ID Generator

```
public class RegistrationIDGenerator {
    public static void main(String[] args) {
        String[] studentNames = {"Riya", "Siya", "Amit", "Neha"};
        String course = "MCA";

        // Using StringBuffer for efficient string manipulation
        for (int i = 0; i < studentNames.length; i++) {
            StringBuffer regID = new StringBuffer();

            regID.append(course.substring(0, 3).toUpperCase()); // Add course code
            regID.append("-");
            regID.append(studentNames[i].toUpperCase());        // Add name
            regID.append("-");
            regID.append(100 + i);                              // Add roll number

            System.out.println("Generated Registration ID: " + regID);
        }
    }
}
```

**Output:**

Generated Registration ID: MCA-RIYA-100
Generated Registration ID: MCA-SIYA-101
Generated Registration ID: MCA-AMIT-102
Generated Registration ID: MCA-NEHA-103

```
class PalindromeCheck {
    public static void main(String[] args) {
        String str; = "madam";

        // Convert string to lowercase (to ignore case sensitivity)
        String original = str.toLowerCase();

        // Reverse the string using StringBuffer
        String reversed = new
        StringBuffer(original).reverse().toString();

        // Check palindrome
        if(original.equals(reversed)) {
            System.out.println(str + " is a Palindrome");
        } else {
            System.out.println(str + " is NOT a Palindrome");
        }
    }
}
```

**Sample Output:**

madam is a Palindrome

# Thanks!