PS02CMCA51 Object Oriented Programming using Java

Dr. J. V. Smart

Table of Contents

- JDBC
- Generics
- Enumeration
- Inner Classes
- Anonymous Classes
- Class Loaders and the Class Path
- The Java Collections Framework
- Serialization and Deserialization
- Multithreading
    - Multithreading in Java
- Network Programming
- Functional Programming and Lambda Expressions

Syllabus

Syllabus with effect from the Academic Year 2021-2022

Course Code	PS02CMCA51	Title of the Course	OBJECT ORIENTED PROGRAMMING USING JAVA
Total Credits of the Course	4	Hours per Week	4

Course Objectives:	1. To learn computer programming using the Java programming language and the Java Platform, Standard Edition (Java SE). 2. To learn the fundamentals of object-oriented programming. 3. Learning to write object-oriented programs in Java. 4. Knowledge of important features of the Java SE platform. 5. Learning to develop graphical and database programs using Java.

Course Content

Unit	Description	Weightage* (%)
1.	**Introduction to Java** - The Java programming language: history, evolution, features - Introduction to the Java programming environment, JDK, JRE - Introduction to the IDE - Data types and wrapper classes, operators - Control structures - String handling - Basic Input-output	25
2.	**Introduction to Object-oriented Programming** - Basic concepts of object-oriented programming - Classes, instances, methods - Static and non-static members - Packages - Inheritance and polymorphism, method overriding - Final and abstract classes, abstract methods - Interfaces - Generics, enumeration - Inner classes and anonymous classes - Class loaders, class path	25
3.	**More Features of the Java Platform** - Exception handling - Input-output and file handling - The collections framework and handling classes in it - Introduction to the java.util package - Multithreading - Introduction to network programming - Introduction to lambda expressions and serialization	25
4.	**Developing Graphical Programs and Database Access**	25

Unit	Description	Weightage* (%)
	- An introduction to graphics in Java - Brief introduction to AWT - The Swing library - Writing graphical programs using Swing - Using various Swing components - Managing layout using Swing - Event handling using Swing - Introduction to JDBC - Different types of JDBC drivers - Programming database applications using JDBC	

Teaching-Learning Methodology	Blended learning approach incorporating traditional classroom teaching as well as online / ICT-based teaching practices

Evaluation Pattern

Sr. No.	Details of the Evaluation	Weightage
1.	Internal Written / Practical Examination (As per CBCS R.6.8.3)	15%
2.	Internal Continuous Assessment in the form of Practical, Viva-voce, Quizzes, Seminars, Assignments, Attendance (As per CBCS R.6.8.3)	15%
3.	University Examination	70%

Course Outcomes: Having completed this course, the learner will be able to	
1.	develop computer programs using the Java programming language and the Java SE platform.
2.	gain an understanding of fundamental object-oriented programming concepts.
3.	develop object-oriented software in Java.

Course Outcomes: Having completed this course, the learner will be able to	
4.	display knowledge of multithreading, file handling and network programming in Java.
5.	develop GUI programs in Java.
6.	have knowledge of database access in Java using JDBC.

Suggested References:

Sr. No.	References
1.	Schildt H. : Java: The Complete Reference, 9th Edition, McGraw-Hill Education, 2017.
2.	Deitel P., Deitel, H. : Java: How to Program: Early Objects, 11th Edition, Pearson Education, 2018.
3.	Rao, R. N.: Core Java: An Integrated Approach, New Edition, Dreamtech Press, 2008.
4.	Horstmann C. : Core Java Volume I – Fundamentals, 11th Edition, Prentice Hall, 2018.
5.	Horstmann C. : Core Java, Volume II – Advanced Features, 11th Edition, Prentice Hall, 2018.

On-line resources to be used if available as reference material	
1.	Java SE API Documentation.
2.	The Java™ Tutorials.

Introduction

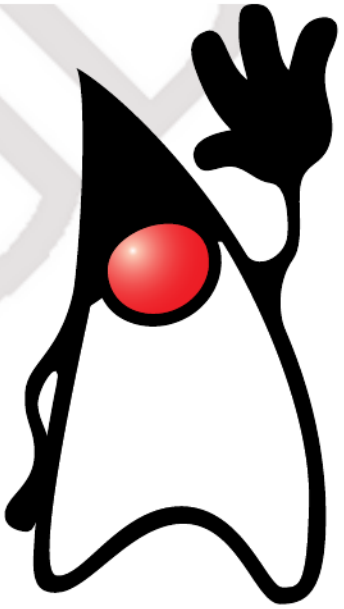
- What is Java?
  - A programming language
  - A software platform

History

James Gosling

- James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991
- The language was initially called Oak after an oak tree that stood outside Gosling's office
- Later the project was named Green; then it was renamed Java, from *Java coffee*
- That is why the logo of the Java programming language is a cup of steaming coffee
- Sun Microsystems released the first public implementation as Java 1.0 in 1996

The Java Logo

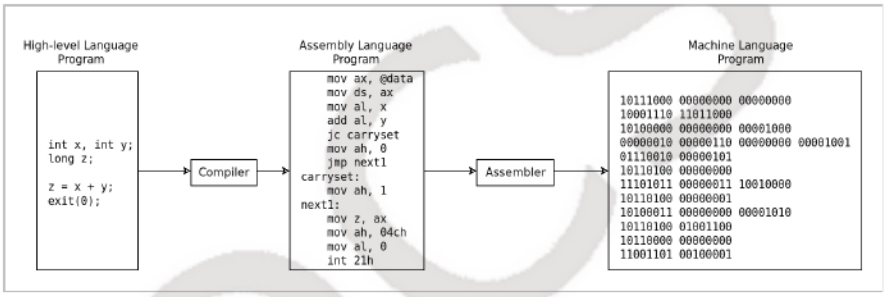


Duke - The Java Mascot

- The Java language was originally designed for embedded systems and to overcome lack of features like garbage collection, security, networking and multithreading as a standard part of the C++ library
- Though Java did not succeed initially as a programming language for embedded systems, it gained widespread adoption for distributing software over the Internet (in the form of Java applets). It was soon incorporated in major browsers
- Originally, Java was developed at Sun Microsystems. Later, Sun Microsystems was purchased by Oracle Corporation. After Oracle transferred Java EE to the Eclipse Foundation, it is now an Eclipse project with the name Jakarta EE.
- According to various developer surveys, Java is one of the most popular programming languages in the world
- The latest version of Java SE is Java 14

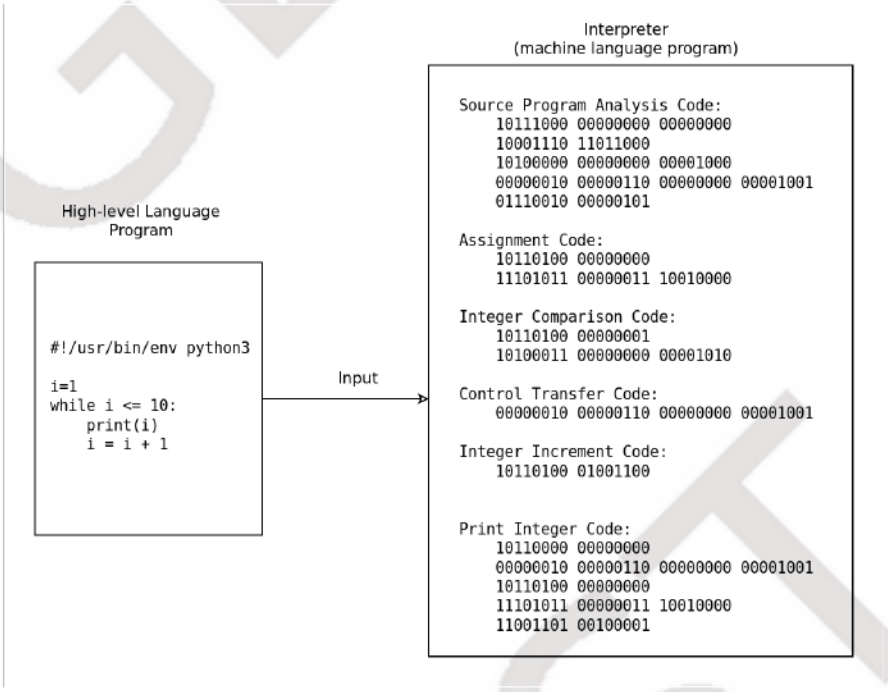
Program Execution Models

Compilation



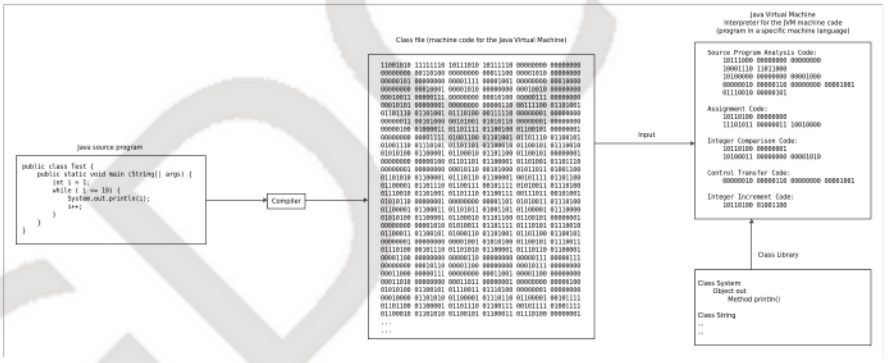
Program Execution through Compilation

Interpretation



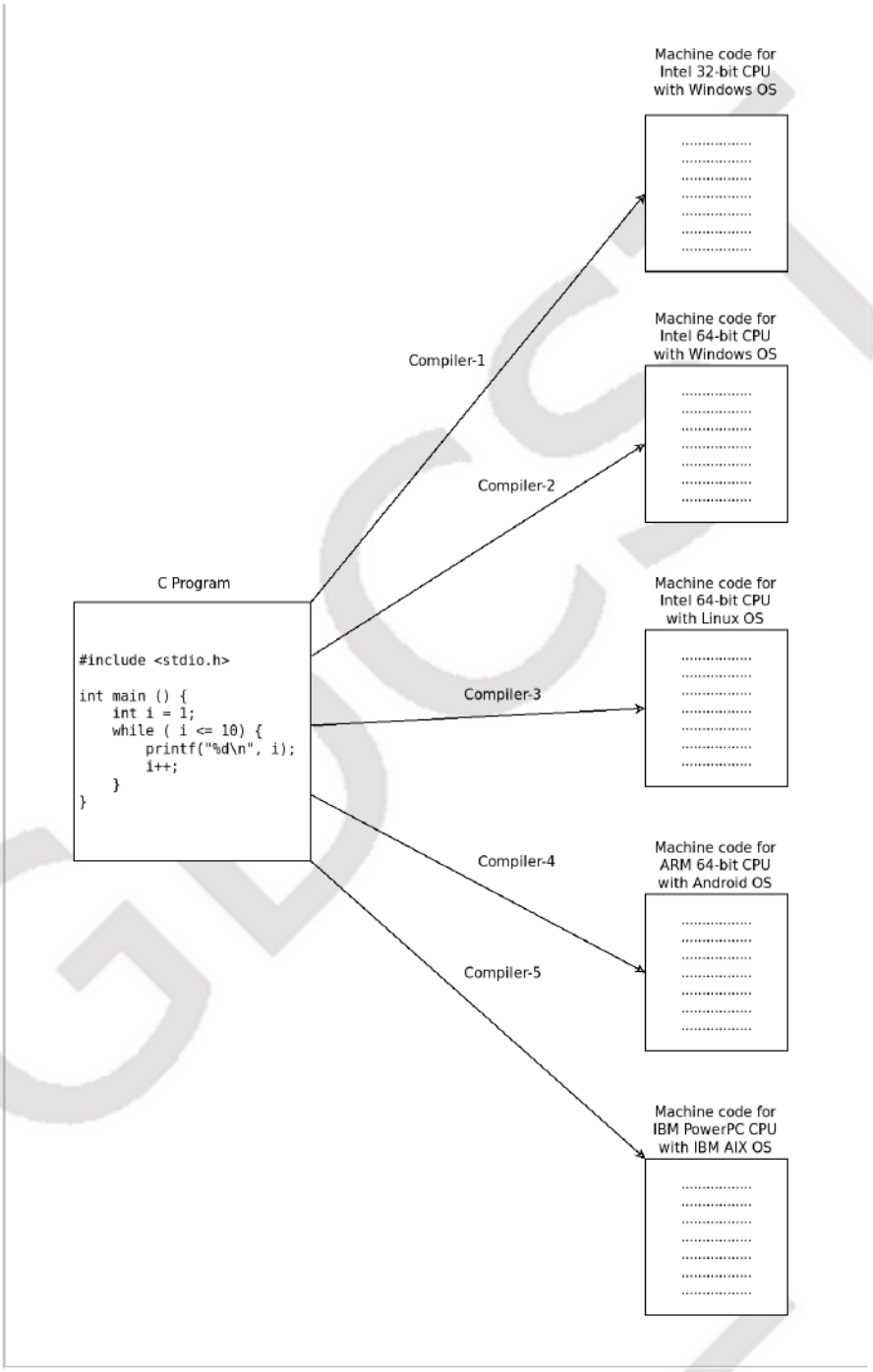
Program Execution through Interpretation

Hybrid Model

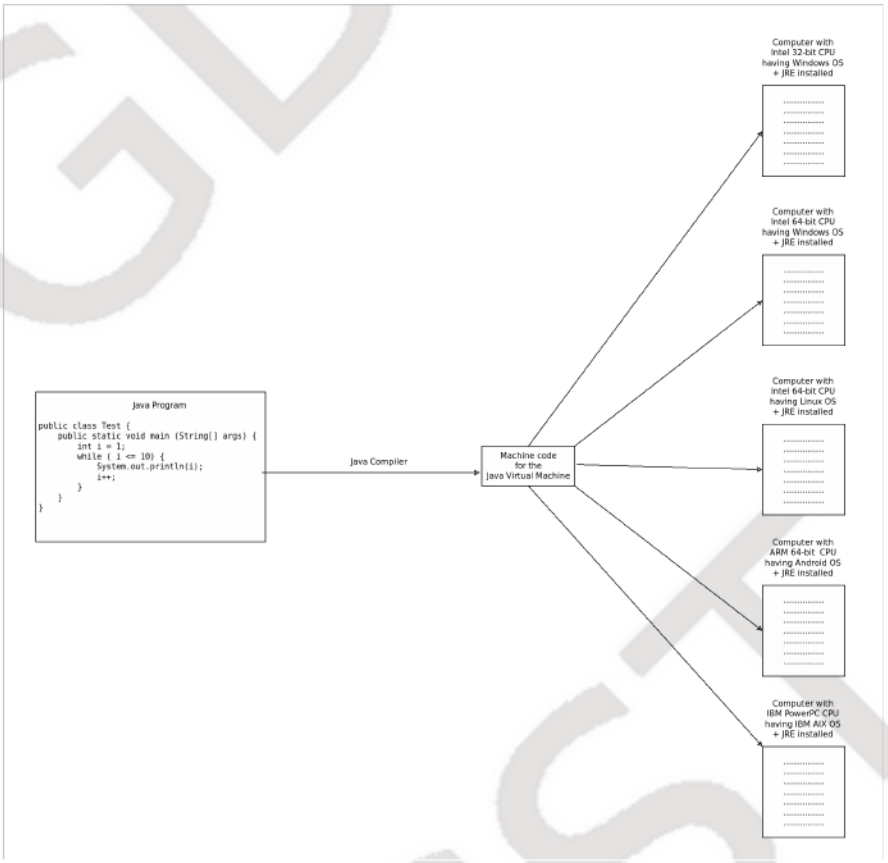


Program Execution using Hybrid Model

Write Once, Run Anywhere (WORA)



Execution of a C Program



Execution of a Java Program

Key Characteristics of The Java Programming Language

- Write Once, Run Anywhere (WORA). Java programs, once compiled, can run on any platform having a JRE (Java Runtime Environment) without recompilation
- Statically-typed language with checks on stack usage, array indexing
- Uses references in place of explicit pointers
- Built-in security, garbage collection, multithreading, networking, etc.
- Large class library providing various features
- Dynamic class loading
- Designed for international use from beginning
- Automatic documentation generator (Javadoc)

Key Terminology

- Java programming language
- Java platform

- WORA (Write Once, Run Anywhere)
- Java Runtime Environment (JRE)
    - Java bytecode interpreter (JVM)
    - API class libraries
    - Some tools
- Java Development Kit (JDK)
    - JRE
    - Java compiler
    - Some additional tools
- Java API (Application Programming Interface)
- Java bytecode
- JVM (Java Virtual Machine)
- Memory leak
- Garbage collection

The Java Platform

Advantages

- **Platform-independence:** The Java platform is designed with the goal of running the same compiled *binary* programs *unmodified* on *any* platform
- It is an object-oriented programming language. Support for several functional programming features has also been added in newer versions
- Memory management is easy and less error-prone due to garbage collection
- Checks on stack usage, array indexing, etc.
- Built-in security, garbage collection, multithreading, networking, etc.
- A single binary executable can run on any platform that has a JRE for it
- Does not use explicit pointers, making programming easy and safer
- The standard library of the Java platform provides a lot of functionality
- Supports multiple programming paradigms including object-oriented programming and functional programming
- Built-in support for internationalization and localization

Disadvantages

- **Performance Issues:** The Java programs are compiled to Java bytecode, which is machine code for an abstract hypothetical machine (called the Java Virtual Machine) that did not exist when Java was created. Later on, some processors with hardware support for executing Java bytecode have been created, but they have not been very successful. Since the commonly used CPUs cannot understand Java bytecode, the Java bytecode is interpreted by an interpreter, which is also called the Java Virtual Machine (JVM). The JVM is a software emulator for the hypothetical Java virtual machine. Even though interpreting bytecode is comparatively faster than interpreting source code directly, it is not as fast as direct execution by the CPU. Use of several techniques, including profiling, partial JIT (Just-In-Time) compilation and aggressive optimizations means that the performance of Java is comparable to that of languages with native compilation. Still performance concerns remain in some types of applications. Garbage collection also incurs considerable and arbitrary performance penalty, though the algorithms have been substantially improved over a period of time
- No automatic update

Java Editions

- **Java Card** for smart cards, ATM cards, SIM cards, etc.
- **Java Platform, Micro Edition** (Java ME, earlier J2ME) - for embedded and mobile devices
- **Java Platform, Standard Edition** (Java SE, earlier J2SE) - for desktops and workstations
- **Jakarta EE** (earlier Java Platform, Enterprise Edition / Java EE, still earlier J2EE) - for large enterprise applications

Implementations

- OpenJDK
- Oracle (Sun) JDK

Other JVM Languages

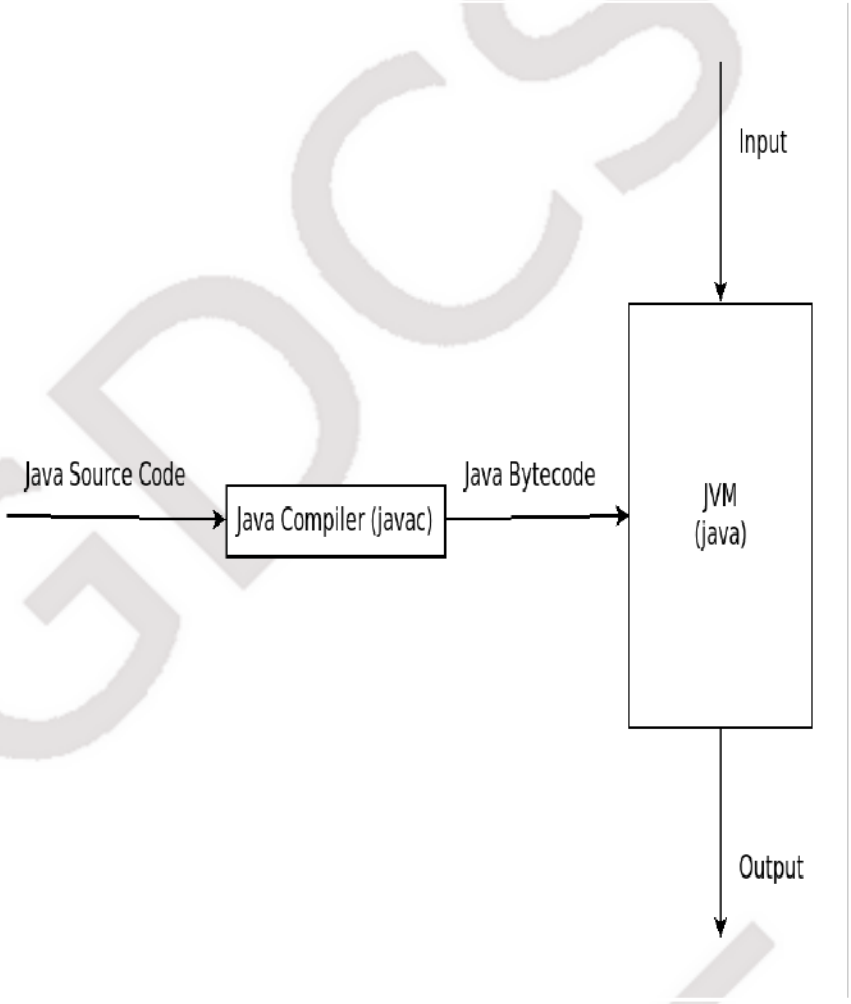
- Scala

- Clojure
- Groovy
- Kotlin

License

- Originally, the Java implementation by Sun Microsystems was freely available, but licensed under a proprietary license
- Much of the Sun JDK code is now made available as open source software
- OpenJDK is a completely free and open source implementation
- OpenJDK is now the official reference implementation

Java Development Cycle



The Java Development Cycle

- **Using any plain text editor:** Edit → Compile → Run
- **Using an IDE:** NetBeans, Eclipse, IntelliJ IDEA, Oracle JDeveloper, etc.

Fundamentals of the Java Programming Language

Language Basics

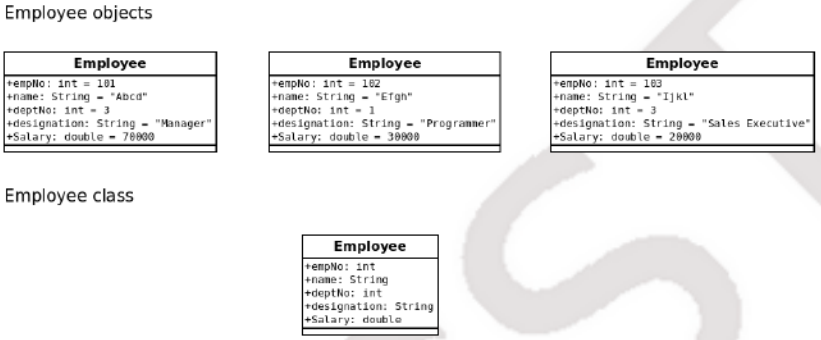
- Java is a statically-typed, free-form, case-sensitive language
- Character set
  - Unicode with UTF-16 encoding
  - Most characters (those in the Basic Multilingual Plane or BMP) are encoded using 16-bit (2-byte) `char` values, but the supplementary characters are represented as a pair of 16-bit `char` values, called surrogate pair
- Data types
  - Primitive Types
    - byte (1-byte *signed* integer) (-2^7 to 2^7-1 or -128 to 127)
    - short (2-byte *signed* integer) (-2^{15} to $2^{15}-1$ or -32768 to 32767)
    - int (4-byte *signed* integer) (-2^{31} to $2^{31}-1$ or -2_147_483_648 to 2_147_483_647)
    - long (8-byte *signed* integer) (-2^{63} to $2^{63}-1$ or -9223372036854775808 to 9223372036854775807)
    - float (4-byte IEEE 754 single-precision floating point number)
    - double (8-byte IEEE 754 double-precision floating point number)
    - char (2-byte single Unicode code point (character) in UTF-16 encoding)
    - boolean
  - Reference Types
    - All variables of class type

- UnicodeExample1.java program

Programming paradigms

- Unstructured programming
- Structured programming (Pascal, C, PHP, Python, etc.)
- Object-oriented programming
- Functional programming

Objects and classes



Objects and Classes

- An object is a computer representation of some real world entity
- A class is a blueprint for creating objects
- A class is a user-defined data type
- An object consists of data and operations on those data
- Classes and objects have members. There are two types of members:
  - **Fields** A field is a data member that holds a value of any data type
  - **Methods** A method is a member function that represents an operation on the class / object
- Members of a class / object (including fields and methods) can be accessed by their name from within the same class / object. To access them from another class / object, the `class.member` or `object.member` notation is used. `.` is the dot (member) operator in Java

Student objects

```
{
    PID                 "MG20001"
    name                "Abcd"
    phone               9876543210
    email               abcd@xyz.com
    address             abcdefg
    programme           MCA
    currentSemester     II
}

{
    PID                 "MG20002"
    name                "Efgh"
    phone               8765432109
    email               efgh@xyz.com
    address             qwerty
```

```
    programme           MCA
    currentSemester     II
}

{
    PID                 "MG20003"
    name                "Ijkl"
    phone               7654321098
    email               ijkl@xyz.com
    address             abcdefg
    programme           MCA
    currentSemester     II
}
```

Student class

```
class Student {
    String PID,
    String name,
    long phone,
    String email,
    String address,
    String programme,
    byte currentSemester
}
```

```
// StudentsResult1.java program
// Without object-orientation (procedural programming)
// Data and operations on the data are scattered throughout the
        project
public class StudentsResult1
{
    static String name;
    static int[] marks;

    static void calculatePercentage() {
        float total = 0;
        for (int i = 0; i < 6; i++) {
            total += marks[i];
        }
        float percentage = total / 6.0f;
        System.out.println("name=" + name + " percentage=" +
        percentage);
    }

    public static void main (String[] args)
    {
        name = "Abc";
        marks = new int[6];
```

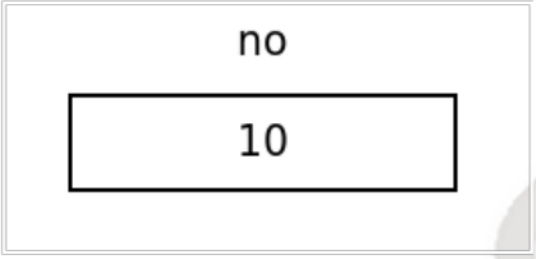
```
        marks[0]=45; marks[1]=54; marks[2]=61; marks[3]=65;
        marks[4]=71;  marks[5]=81;
        calculatePercentage(); // Correct result

        name = "Xyz";
        marks = new int[7];
        marks[0]=45; marks[1]=54; marks[2]=61; marks[3]=65;
        marks[4]=71;  marks[5]=81; marks[6]=93;
        calculatePercentage(); // WRONG result
    }

}
```

```
// StudentsResult2.java program
// With object-orientation
// Data and operations on the data are in the same place
// (bundled in a single class)
class Student
{
    String name;
    final int SUBJECT_COUNT;
    int[] marks;

    Student(String name, final int SUBJECT_COUNT, int[] marks)
    {
        this.name = name;
        this.SUBJECT_COUNT = SUBJECT_COUNT;
        this.marks = marks;
    }

    void calculatePercentage() {
        float total = 0;
        for (int i = 0; i < SUBJECT_COUNT; i++) {
            total += marks[i];
        }
        float percentage = (float)total / SUBJECT_COUNT;
        System.out.println("name=" + name + " percentage=" +
        percentage);
    }

}

public class StudentsResult2
{

    public static void main (String[] args)
    {
        String name;
        int[] marks;

        name = "Abc";
```

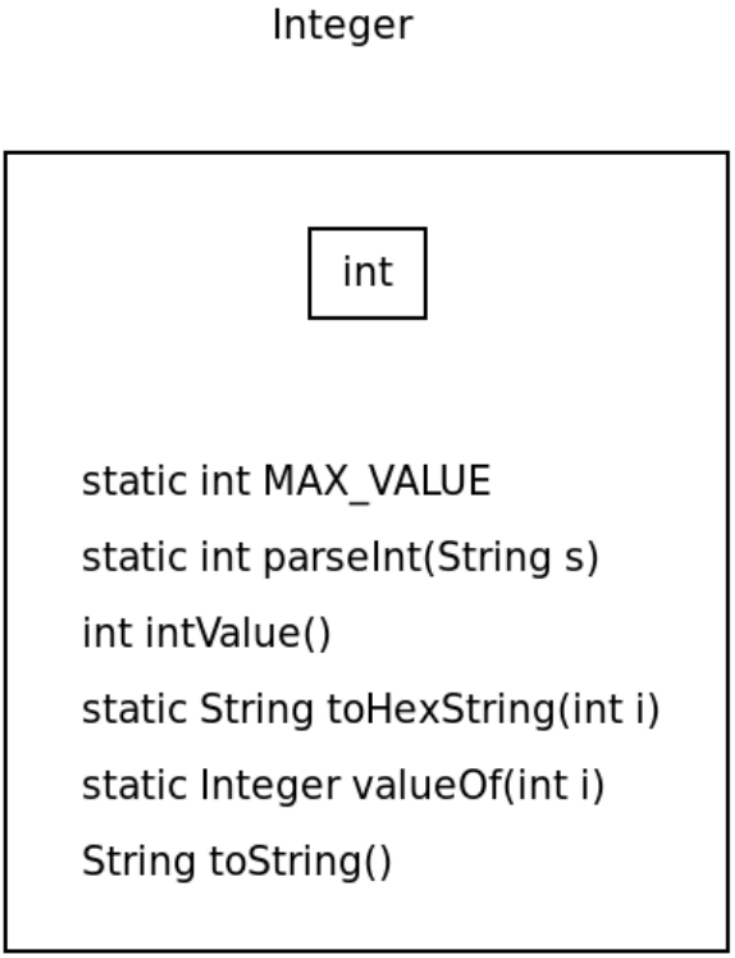
```
        marks = new int[6];
        marks[0]=45; marks[1]=54; marks[2]=61; marks[3]=65;
         marks[4]=71;  marks[5]=81;
        Student student1 = new Student(name, 6, marks);
        student1.calculatePercentage(); // Correct result

        name = "Xyz";
        marks = new int[7];
        marks[0]=45; marks[1]=54; marks[2]=61; marks[3]=65;
         marks[4]=71;  marks[5]=81; marks[6]=93;
        Student student2 = new Student(name, 7, marks);
        student2.calculatePercentage(); // Correct result
    }
}
```

Some important classes from the Java API

- java.lang.Object
- java.lang.String
- Wrapper classes
  - java.lang.Byte
  - java.lang.Short
  - java.lang.Integer
  - java.lang.Long
  - java.lang.Float
  - java.lang.Double
  - java.lang.Character
  - java.lang.Boolean



Primitive Type int



Wrapper Class Integer

```
// WrapperClassExample01.java program
public class WrapperClassExample01
{
    public static void main (String[] args)
    {
        int no = 10;
        long no2 = 4000000001;
        System.out.println(Integer.MAX_VALUE);
        if (no2 > Integer.MAX_VALUE) {
            System.out.println(no2 + " cannot fit in int");
        } else {
            System.out.println(no2 + " can fit in int");
        }

        int i1 = 10;
        float f1 = 12.34f;
        Integer integer1;
        Float float1;

        // Conversion from primitive type to wrapper class type
        integer1 = Integer.valueOf(i1);
```

```
        float1 = Float.valueOf(f1);

        // Automatic boxing
        integer1 = i1;
        float1 = f1;

        // Conversion from wrapper class type to primitive type
        i1 = integer1.intValue();
        f1 = float1.floatValue();

        // Automatic unboxing
        i1 = integer1;
        f1 = float1;

    }
}
```

Java Naming Conventions

- Java uses the camel case 🐫 naming convention
    - All internal words in an identifier name begin with an uppercase letter (except the first one). E.g. convertFromCsvToHtml()
    - The name of a class always begins with an uppercase letter
    - The name of a constant should be entirely in uppercase letters, with words separated by _ (underscore) (e.g. MAX_VALUE)
    - The names of all other identifiers begin with a small letter
    - All identifier names (except those for constants) follow camel case notation
    - All keywords are entirely in lowercase letters
    - Names of all primitive types start with a lowercase letter
    - Names of all other data types start with an uppercase letter
    - Package names always consist of only lowercase letters and . (period character)

Data items

A data item can have several attributes. Some common attributes of a data item are name, data type and value. When we assign value to a data item for the first time, it is known as initialization of that data item. A data item must be initialized before its value can be used.

Class and object members

Classes and objects can have two types of members: fields and methods.

Variables, constants and literals

- **Variables** have a name, a data type and a mutable value (a value that can be changed). The variable may be accessed in other parts of the code within its scope by its name. The value of a variable may be changed

```
// Example of a variable
int x = 10; // variable name: x, data type: int, mutable
        (changeable) value: 10
 ...
 ...
x = 20; // The value of a variable can be changed
```

- **Constants** have a name, data type and an immutable value (a value that cannot be changed). The constant may be accessed in other parts of the code within its scope by its name. The value of a constant cannot be changed. A constant may be assigned a value only during initialization. In Java, the `final` keyword is used to declare a constant

```
// Example of a constant
final int x = 10; // constant name: x, data type: int, immutable
        (unchangeable) value: 10
 ...
 ...
x = 20; // ERROR: variable might already have been assigned
        // The value of a constant CANNOT be changed
```

- **Literals** are values embedded in the program source (i.e. code). They have a data type and an immutable value, but no name. The literal cannot be accessed in other parts of the code because it has no name. The value of a literal cannot be changed

```
int x = 10; // 10 is a literal here. It has no name.
            // data type: int, immutable (unchangeable) value: 10
 ...
 ...
x = 20;
System.out.println(x);
String name = "Abcd";   // "Abcd" is a literal here. It has no
        name.
                        // data type: String, immutable
        (unchangeable) value: "Abcd"
```

- **Using literals of different types**
  - int: 1, 123, +123, -123, 2147483647, 2_147_483_647 (since JDK 1.7)
  - long: 1l, -123l, 400_000_000_000l
  - double: 1.0, 123.45, +234.4567, -123.797, 2147483647.0, 1.0d, 1d, 123.45d, 123d
  - float: 1.0f, 123.45f, +234.4567f, -123.797f, 2147483647.0f, 1f, 123f
  - char: 'a', 'b', 'n', '\n', '\t', '\\', '꠹', '\uXXXX' ('\u0a95' = 'ક', '\u20b9' = '₹')
  - boolean: true, false
  - java.lang.String: "abcd", "xyz", "a", "", "Price=₹ 123.45", "Price=\u20b9 123.45" (static final objects of class java.lang.String)
  - Array literals: A literal of an array of any type can be created by enclosing values of that type in curly braces `{ }`. For example, `{ 10, 20, 30 } // array of ints`, `{ 12.34, 23.456 } // array of doubles`, `{ "Abcd", "Defg", "Hijk", Lmno" } // array of strings`

Block structure and local variables / constants

```python
# block-structure.py program
print('Enter an integer:', end=' ')
no = int(input())
for i in range(no):
    print(i)
print('*****')
for i in range(no):
    print(i)
    print('----------------')
print('===== The End =====')
```

```java
// BlockStructure.java program
import java.util.Scanner;

public class BlockStructure
{
    public static void main (String[] args)
    {
        System.out.print("Enter an integer: ");
        Scanner scanner = new Scanner(System.in);
        int no = scanner.nextInt();
        for (int i = 0; i < no; i++)
            System.out.println(i);
        System.out.println("*****");
        for (int i = 0; i < no; i++)
```

```
        {
            System.out.println(i);
            System.out.println("------------");
        }
        System.out.println("===== The End =====");
    }
}
```

```
{ // Block level 1 start...
    int i = 10;
    ...
    ...
    { // Block level 2 start...
        int j = 20;
        ...
        ...
        { // Block level 3 start...
            int k = 30;
            ...
            ...
            // int i; // NOT ALLOWED
            ...
        } // Block level 3 end...
        ...
        ...
    } // Block level 2 end...
    ...
    ...
    String str;
    ...
    ...
} // Block level 1 end...
 ...
 ...
```

Comments

- Single line comments: `// ...`
- Multiline comments: `/*  ...  */`
- Javadoc special comments `/**  ...  */`

Operators

- Types of operators
    - **Unary operators** A unary operator takes only one operand. E.g. `i++`

- **Unary prefix operators** A unary operator that comes before its operand. E.g. `++i`
- **Unary postfix operators** A unary operator that comes after its operand. E.g. `i++`

- **Binary operators** A binary operator takes two operands. E.g. `x + y`
- **Ternary operators** A ternary operator takes three operands. E.g. `x>0 ? 10 : 20`

Simple Assignment Operator	
`=`	Simple assignment operator. E.g. `x = 10;` , `x = y + z;`
Arithmetic Operators	
`+`	Addition operator. E.g. `x + 10` , `a + b`
`-`	Subtraction operator. E.g. `x - 10` , `a - b`
`*`	Multiplication operator. E.g. `x * 10` , `a * b`
`/`	Division operator. E.g. `x / 10` , `27 / 10` (results in 2)
`%`	Remainder operator. E.g. `x % 2` , `a % b` , `27 % 10` (results in 7)
String Concatenation Operator	
`+`	String concatenation operator. E.g. `"abc" + "xyz"` , `subject + " quiz"` , `firstName + " " + lastName`
Unary Operators	
`+`	Unary plus operator; indicates positive value (numbers are considered positive by default). E.g. `+12` (`12` is same), `+12.34` (`12.34` is same)
`-`	Unary minus operator; indicates a negative number or negates an expression. E.g. `-12` , `-12.34` , `-x` , `-(x+y)`
`++` (prefix)	Prefix increment operator; increments a value by 1 and returns the incremented value. E.g. `x=10; y=++x;` is same as `x=10; x=x+1; y=x;` and results in x having a value of 11 and y having a value of 11
`++` (postfix)	Postfix increment operator; increments a value by 1 and returns the value before increment. E.g. `x=10; y=x++;` is same as `x=10; y=x; x=x+1;` and results in x having a value of 11 and

	y having a value of 10
`--` (prefix)	Prefix decrement operator; decrements a value by 1 and returns the decremented value. E.g. `x=10; y=--x;` is same as `x=10; x=x-1; y=x;` and results in x having a value of 9 and y having a value of 9
`--` (postfix)	Postfix decrement operator; decrements a value by 1 and returns the value before decrement. E.g. `x=10; y=x--;` is same as `x=10; y=x; x=x-1;` and results in x having a value of 9 and y having a value of 10
`!`	Logical complement operator; inverts a condition — a condition that is `true` becomes `false` and a condition that is `false` becomes `true`. E.g. If x=10, then `x>0` is `true`, but `!(x>0)` is `false`. Similarly, if x=10 and y=20, then `x>y` is `false`, but `!(x>y)` is `true`
Equality and Relational Operators	
`==`	Equal to. E.g. `x == 10`, `x == y`
`!=`	Not equal to. E.g. `x != 10`, `x != y`
`>`	Greater than. E.g. `x > 10`, `x > y`
`>=`	Greater than or equal to. E.g. `x >= 10`, `x >= y`
`<`	Less than. E.g. `x < 10`, `x < y`
`<=`	Less than or equal to. E.g. `x <= 10`, `x <= y`
Conditional Operators	
`&&`	And. The condition is true if and only if both the operands of the `&&` operator are true. E.g. `a>b && x>y`
`||`	Or. The condition is true if any one of the operands of the `||` operator are true. E.g. `a>b || x>y`
`?:`	Ternary operator; it checks the condition before the `?` and if the condition is `true`, than it returns the value before the `:`, otherwise it returns the value after the `:`. E.g. `y = x>0 ? 10 : 20;` is same as `if (x>0) y=10; else y=20;`
Compound Assignment Operators	

`+=`	Addition and assignment operator. It adds the second operand to the first operand and assigns the result to the first operand. E.g. `x += 10;` is same as `x = x + 10`
`-=`	Subtraction and assignment operator. It subtracts the second operand from the first operand and assigns the result to the first operand. E.g. `x -= 10;` is same as `x = x - 10;`
`*=`	Multiplication and assignment operator. It multiplies the second operand with the first operand and assigns the result to the first operand. E.g. `x *= 10;` is same as `x = x * 10;`
`/=`	Division and assignment operator. It divides the first operand by the second operand and assigns the result to the first operand. E.g. `x /= 10;` is same as `x = x / 10;`
`%=`	Remainder and assignment operator. It divides the first operand by the second operand and assigns the remainder to the first operand. E.g. `x %= 10;` is same as `x = x % 10;`

Operator Precedence and Associativity

When a complex expression having multiple operators is evaluated, the order in which the operators are applied is determined by the rules of precedence and associativity and brackets.

- Subexpressions in brackets are evaluated first. In case of nested brackets, they are evaluated from the innermost to the outermost
- operators are divided into precedence groups. Out of two adjacent operators, the operator in the higher precedence group is evaluated first
- If two consecutive operators belong to the same precedence group, then they are evaluated according to the associativity of the group (left-to-right or right-to-left)

Group	Operators	Associativity
Unary postfix	`++` `--`	None
Unary prefix	`++` `--` `+` `-` `~` `!`	None (`++`, `--`), Right-to-left
Multiplicative	`*` `/` `%`	Left-to-right
Additive	`+` `-`	Left-to-right

Group	Operators	Associativity
Shift	`<<` `>>` `>>>`	Left-to-right
Relational	`<` `>` `<=` `>=` `instanceof`	Left-to-right
Equality	`==` `!=`	Left-to-right
Bitwise	`&`	Left-to-right
Bitwise	`^`	Left-to-right
Bitwise	`|`	Left-to-right
Logical	`&&`	Left-to-right
Logical	`||`	Left-to-right
Ternary	`? :`	Right-to-left
Assignment and Arrow	`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=` `>>>=` `->`	Right-to-left

Control structures

- if statement
- switch statement // String type is allowed in switch since JDK 1.7
- while loop
- do..while loop
- for loop
- enhanced for loop

The if statement

```
int i = -10;
if (i < 0)
    System.out.println("Negative");
/////// OUTPUT ///////
Negative
```

```
int i = 10;
if (i < 0)
    System.out.println("Negative");
/////// OUTPUT ///////
```

```
int i = 10;
if (i < 0)
    System.out.println("Negative");
else
    System.out.println("Non-negative");
/////// OUTPUT ///////
Non-negative
```

```
int i = 10;
if (i < 0)
    System.out.println("Negative");
else
    if (i == 0)
        System.out.println("Zero");
    else
    {
        System.out.println("Positive");
        System.out.println("Good");
    }
/////// OUTPUT ///////
Positive
Good
```

```
int i = 10;
if (i < 0)
    System.out.println("Negative");
else if (i == 0)
        System.out.println("Zero");
else
    {
        System.out.println("Positive");
        System.out.println("Good");
    }
/////// OUTPUT ///////
Positive
Good
```

The switch statement

```
int i = 3;
switch(i)
{
    case 1:
            System.out.println("One");
            System.out.println("Odd number");
            break;
```

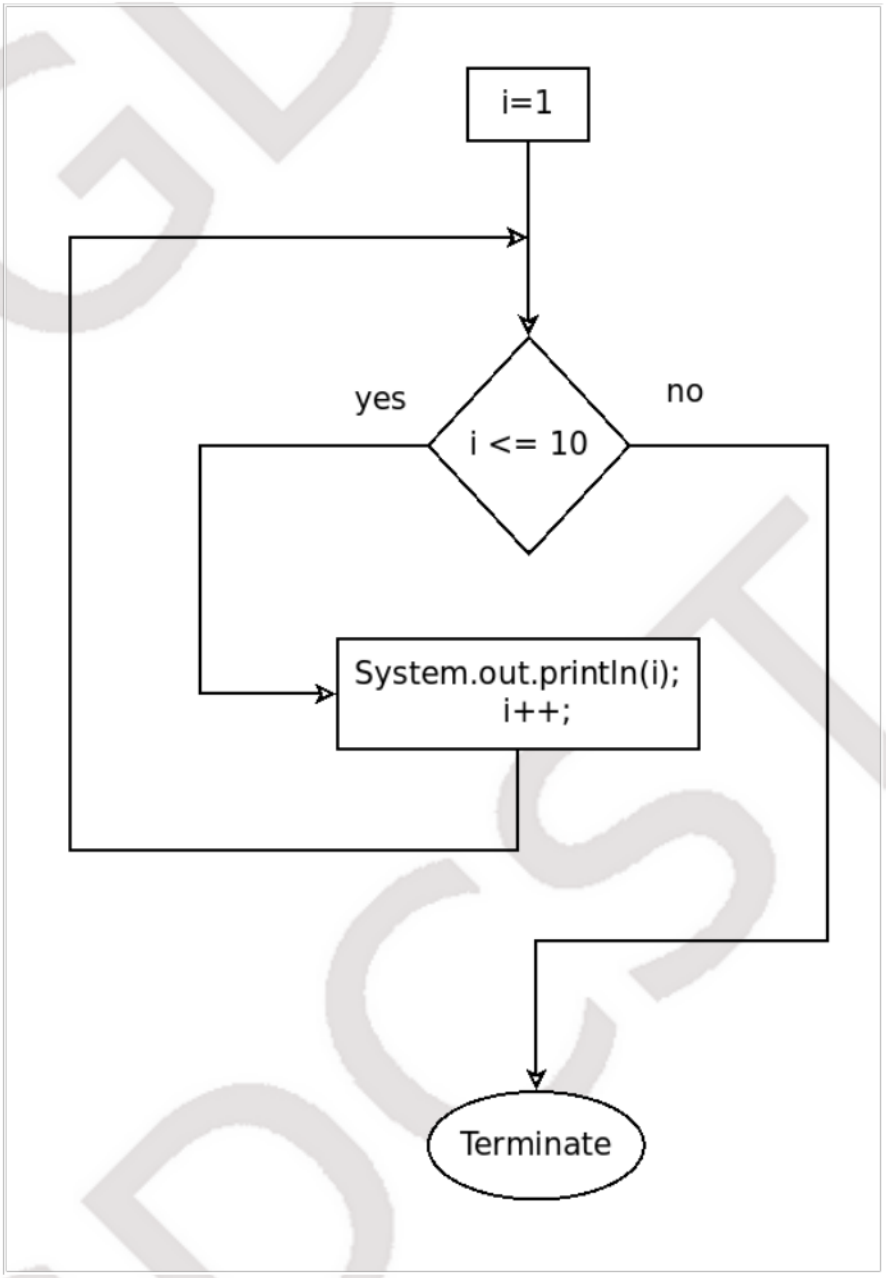
```
    case 2:
            System.out.println("Two");
            System.out.println("Even number");
            break;
    case 3:
            System.out.println("Three");
            System.out.println("Odd number");
            break;
    default:
            System.out.println("Something else");
}
/////// OUTPUT ///////
Three
Odd number
```

```
int op1 = 20, op2 = 10;
int result = -1;
String operation = "ADD";
switch(operation)
{
    case "ADD":
        result = op1 + op2;
        break;
    case "SUB":
        result = op1 - op2;
        break;
    case "MUL":
        result = op1 * op2;
        break;
    case "DIV":
        result = op1 / op2;
        break;
    case "MOD":
        result = op1 % op2;
        break;
    default:
        System.err.println("invalid operator");
}
System.out.println(result);
/////// OUTPUT ///////
30
```

The while loop

```
i = 1;
System.out.println("Before loop...");
// while is an entry-controlled loop
// The condition is checked at the point of entry
```

```java
// into the loop
while (i <= 10)
{
    System.out.println(i);
    i++;
}
System.out.println("After loop...");
System.out.println(i);
/////// OUTPUT ///////
Before loop...
1
2
3
4
5
6
7
8
9
10
After loop...
11
```

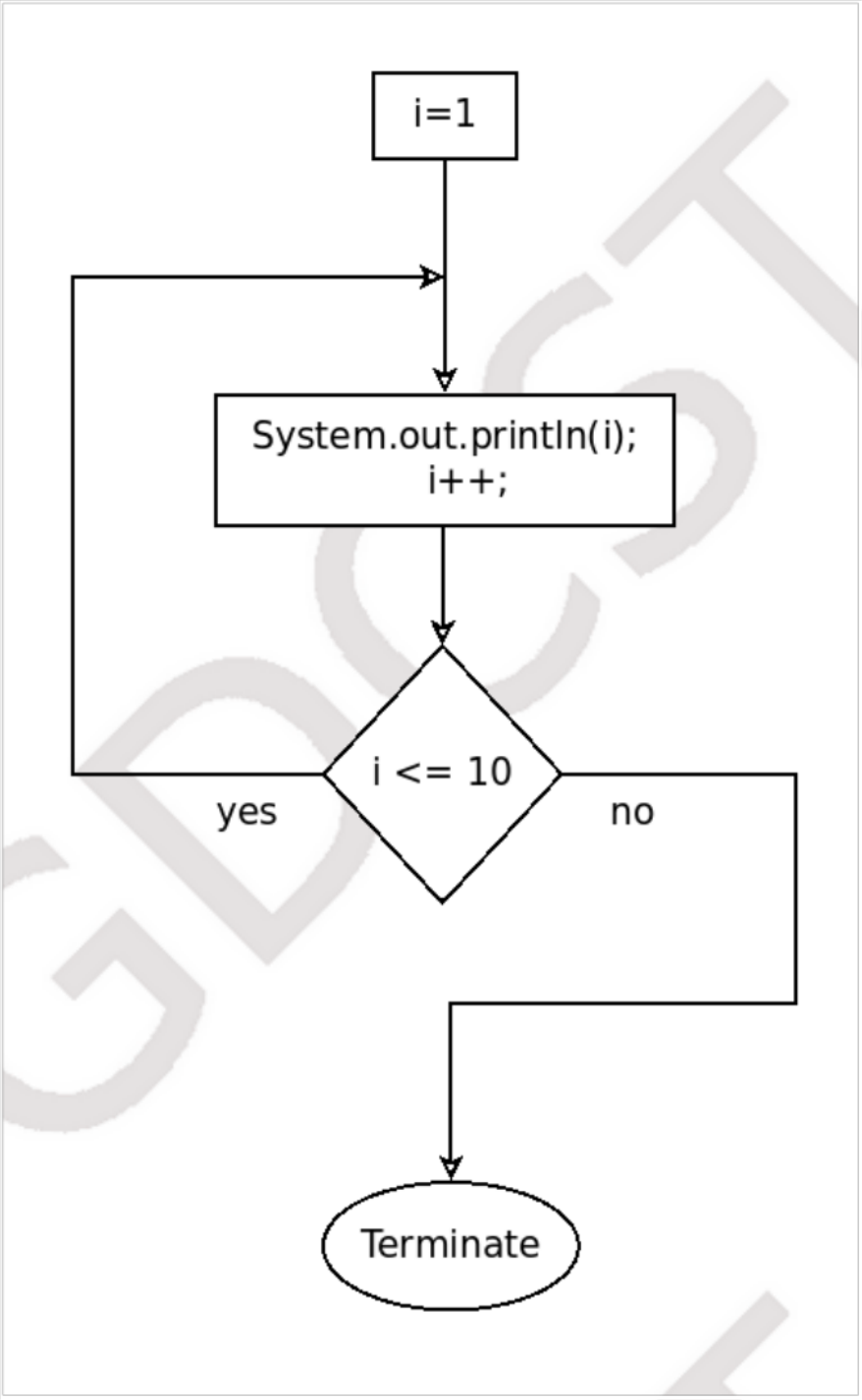


while loop

```
i = 20;
System.out.println("Before loop...");
while (i <= 10)
{
    System.out.println(i);
    i++;
}
System.out.println("After loop...");
System.out.println(i);
/////// OUTPUT ///////
Before loop...
After loop...
20
```

The do..while loop

```java
i = 1;
System.out.println("Before loop...");
// do...while is an exit-controlled loop
// The condition is checked at the point of exit
// from the loop
do
{
    System.out.println(i);
    i++;
} while (i <= 10);
System.out.println("After loop...");
System.out.println(i);
/////// OUTPUT ///////
Before loop...
1
2
3
4
5
6
7
8
9
10
After loop...
11
```



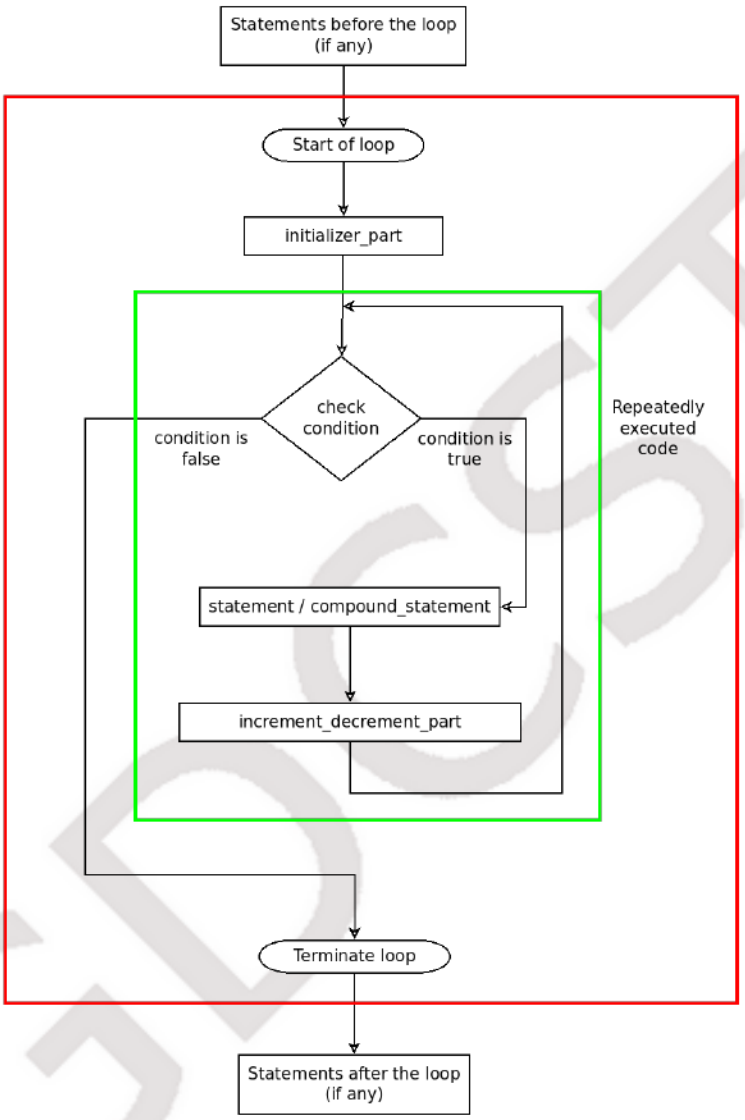
do..while loop

```
i = 20;
System.out.println("Before loop...");
do
{
    System.out.println(i);
    i++;
} while (i <= 10);
System.out.println("After loop...");
System.out.println(i);
/////// OUTPUT ///////
Before loop...
20
```

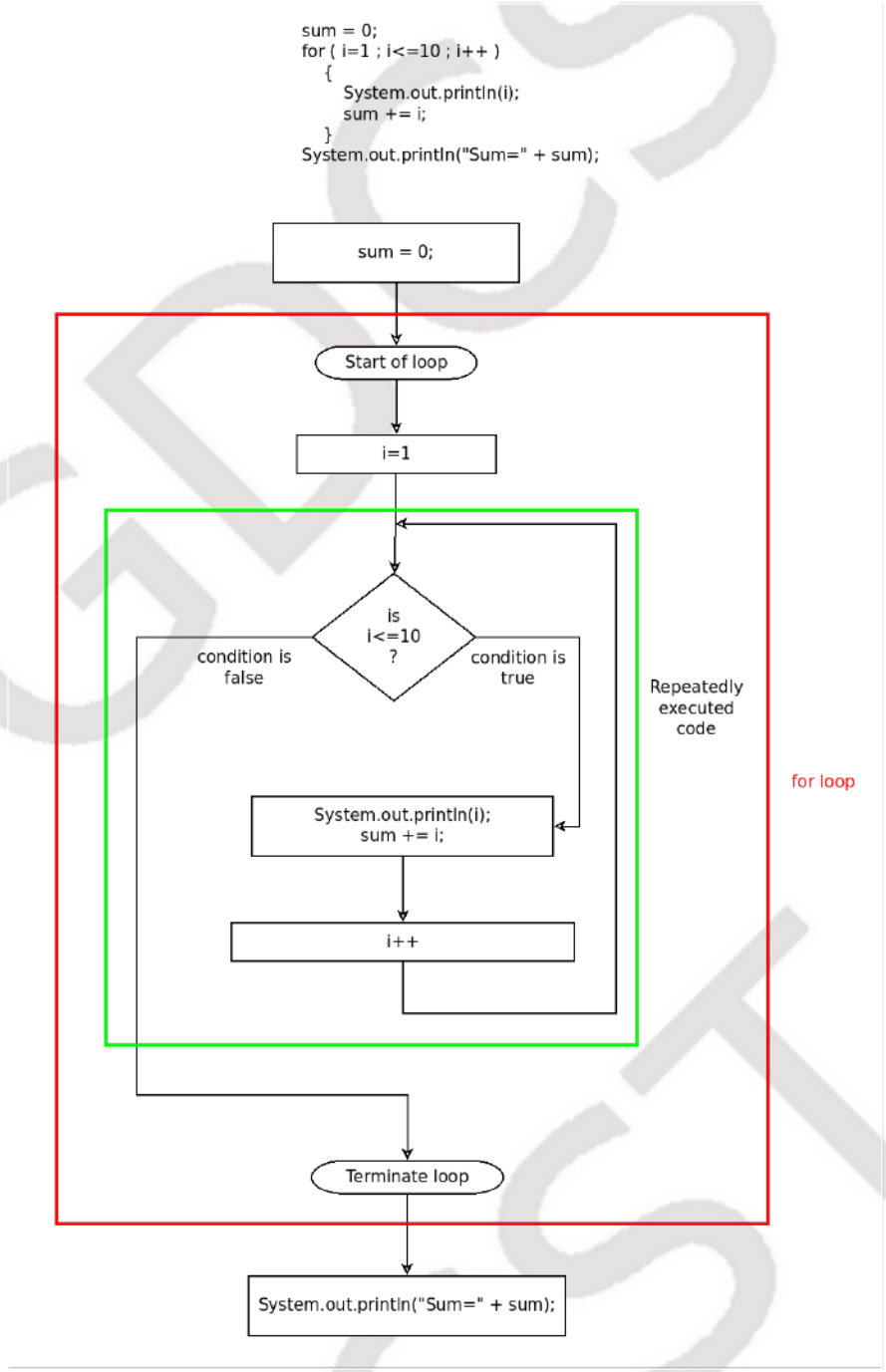
```
After loop...
21
```

The for loop

for (initializer_part ; condition ; increment_decrement_part)
statement / compound_statement



for loop



for loop

```
int i;
for (i = 0; i < 10; i++)
    System.out.println(i);
/////// OUTPUT ///////
0
1
2
3
4
5
6
7
8
```

```
9
```

```java
for (int i = 0; i < 10; i++)
{
    System.out.println(i);
    System.out.println("--------------");
}
/////// OUTPUT ///////
0
--------------
1
--------------
2
--------------
3
--------------
4
--------------
5
--------------
6
--------------
7
--------------
8
--------------
9
--------------
```

```java
int i;
for (i = 10; i > 0; i--)
    System.out.println(i);
/////// OUTPUT ///////
10
9
8
7
6
5
4
3
2
1
```

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

```
    {
        System.out.println("Before loop...");
        int sum, i;
        sum = 0;
        for (i = 1; i <= 30; i+=5) {
            System.out.println("i=" + i + " sum=" + sum);
            sum += i;
        }
        System.out.println("After loop...");
        System.out.println("i=" + i + " sum=" + sum);
    }
}
/////// OUTPUT ///////
Before loop...
i=1 sum=0
i=6 sum=1
i=11 sum=7
i=16 sum=18
i=21 sum=34
i=26 sum=55
After loop...
i=31 sum=81
```

The enhanced for loop

```
int[] arrayInt = {10, 20, 30, 40, 50, 60, 70};
// Enhanced for loop
for (int value: arrayInt) {
    System.out.println(value);
}
/////// OUTPUT ///////
10
20
30
40
50
60
70
```

```
String[] arrayDays = { "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday", "Saturday" };
// Enhanced for loop
for (String day: arrayDays) {
    System.out.println(day);
}
/////// OUTPUT ///////
Sunday
Monday
```

```
Tuesday
Wednesday
Thursday
Friday
Saturday
```

```java
int i = 1;
// Infinite loop
while (i < 10) {
    System.out.println(i);
}
/////// OUTPUT ///////
1
1
1
1
1
1
 ...
 ...
```

```java
int i = 1;
// Infinite loop
while (true) {
    System.out.println(i);
    i++;
}
/////// OUTPUT ///////
1
2
3
4
5
6
 ...
 ...
```

The break and continue statements

```java
int i = 1;
while (true) {
    System.out.println(i);
    i++;
    if (i > 10)
        break;
}
/////// OUTPUT ///////
```

```
1
2
3
4
5
6
7
8
9
10
```

```
int i = 0;
while (++i <= 10) {
    if (i%2 == 0)
        continue;
    System.out.println(i);
}
/////// OUTPUT ///////
1
3
5
7
9
```

- EnhancedForLoopExample01.java program

Conversion between primitive data types and strings

```
// Conversion between primitive data types and strings

int i1 = 10;
String s1 = new String(i1);
// s1 == "10"

String s2 = "123";
int i2 = Integer.parseInt(s2);
// i2 == 123

float f1 = 12.34;
String s3 = new String(f1);
// s3 == "12.34"

String s4 = "-56.78";
float f2 = Float.parseFloat(s4);
// f2 == -56.78
```

```
int i;
```

```
try
{
    i = Integer.parseInt("123.45");
}
catch (NumberFormatException ex)
{
    System.err.println(ex.getLocalizedMessage());
    i = 0;
}
```

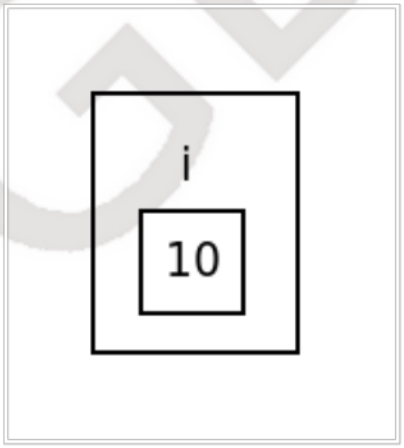
```
float f;
try
{
    f = Float.parseFloat("123.45.6");
}
catch (NumberFormatException ex)
{
    System.err.println(ex.getLocalizedMessage());
    f = 0;
}
```

References in Java

Primitive types

A variable of a primitive type directly holds the value of the variable.

```
int i = 10;
```



Java Primitive Types

Reference types

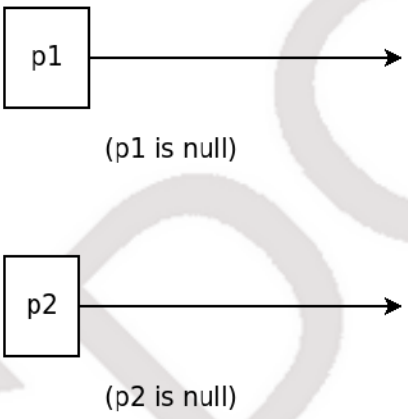
A variable of a reference type does not directly hold an object of that type. It

merely contains a reference (pointer or link) to an object of that type. Also, when the variable of a reference type is declared, an object of that type is not created automatically. Initially, the value of such a variable is `null`. `null` is a special value indicating that the reference type variable does not point to any object. The only valid operations on a variable holding a `null` value are to check whether it is null or not (`p1 == null`) and to compare it with another variable (`p1 == p2`). Any attempt to invoke the dot (member) operator `.` on a null reference results in `java.lang.NullPointerException`. A new object of a reference type may be created using the `new` operator. The `new` operator creates the object in the heap area of the JVM memory. A variable of a reference type may be initialized by assigning it an object created using the `new` operator, by assigning it another variable, or by assigning it a literal of that object type. Memory allocated to an object on the JVM heap using the `new` operator is freed automatically by the garbage collection routine when that object is no longer in use. JVM runs the garbage collection routine from time-to-time or when the JVM nearly runs out of memory.

```
String str; // str is null; it does not point to any String
        object

str = new String("ABC"); // str points to the newly created
        String object

String str2 = str;  // Existing reference type variable is
        assigned to
                    // another reference type variable.
                    // Now both variables point to the same
        object

String str3 = "GDCST";  // A literal object of type String is
        assigned to the variable
```

```
class Person
{
  int pid;
  String name;
  String address;

  Person()
  {
  }

  Person(String name)
  {
    this.name = name;
  }
```

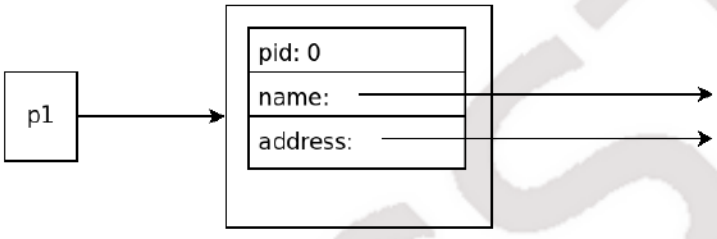
```
}

public class Main
{
 public static void main(String[] args)
    {
       Person p1, p2;
     }
}
```



Java Objects

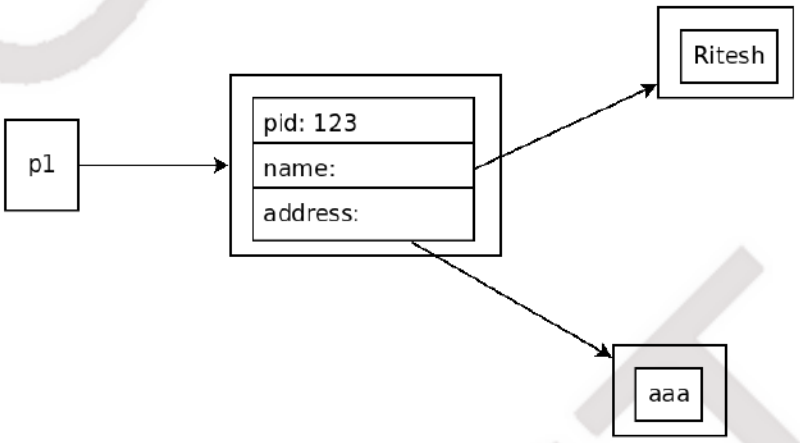
```
p1.pid = 123;  // NullPointerException
```

```
p1 = new Person();  // Construct a new object on the heap and
                    // assign a reference to the object to
                    // the variable p1
```



Java Constructed Object

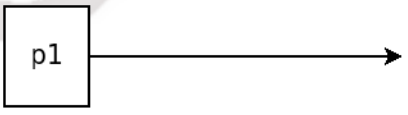
```
p1.pid = 123;
p1.name = "Ritesh"; // p1.name = new String("Ritesh");
p1.address = "aaa";
```



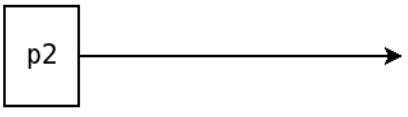
Java Object with Initialized Fields

All variables of reference type are uninitialized upon creation. We must assign them a reference to a valid object (instance) of an appropriate class before we may use them for any purpose. We may assign a reference to an existing object to such a variable or create a new object and assign its reference to the variable. The `new` operator is called with the constructor of a class to create an object of that class. If a class has multiple constructors with different arguments, any of them may be used. The `new` operator returns a reference to the newly created object. This reference may be assigned to a reference-type variable.

```
Person p1;  // the reference p1 is uninitialized
Person p2;  // the reference p2 is uninitialized
```



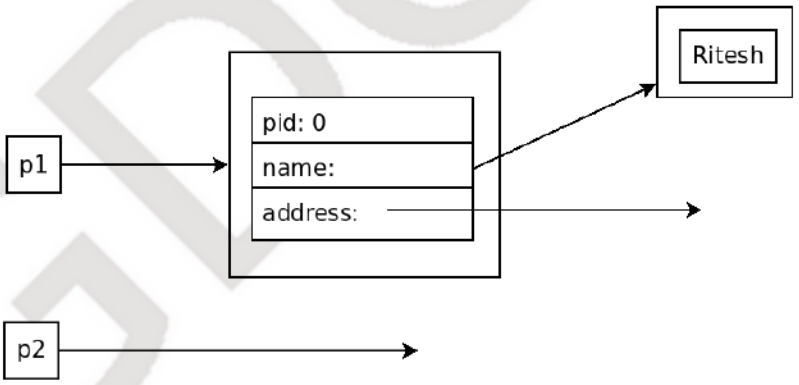
(p1 is null)



(p2 is null)

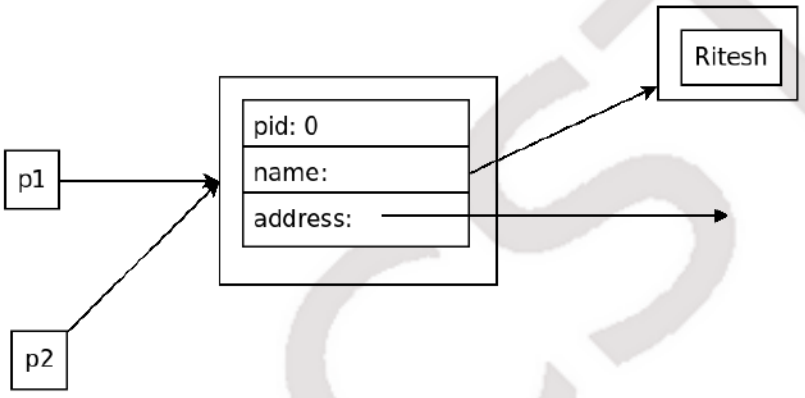
Uninitialized Java References

```
p1 = new Person(“Ritesh”); // the reference p1 now points to a
        valid instance of Person
```



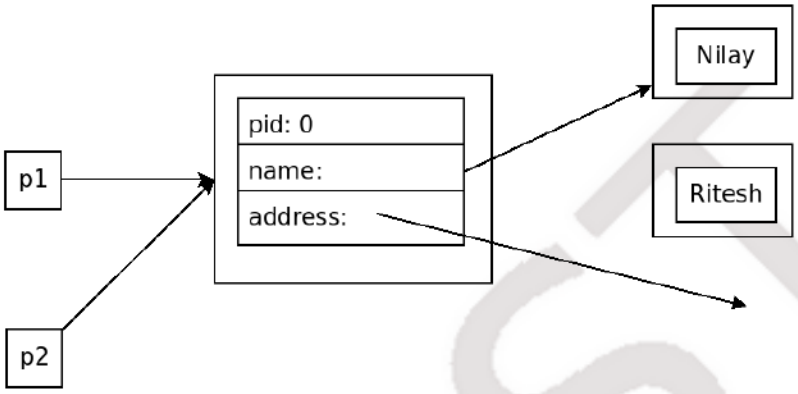
An initialized Java Reference

```
p2 = p1; // Both p2 and p1 now point to the same object.
         // Changing p1 will change p2 and vice versa!
```



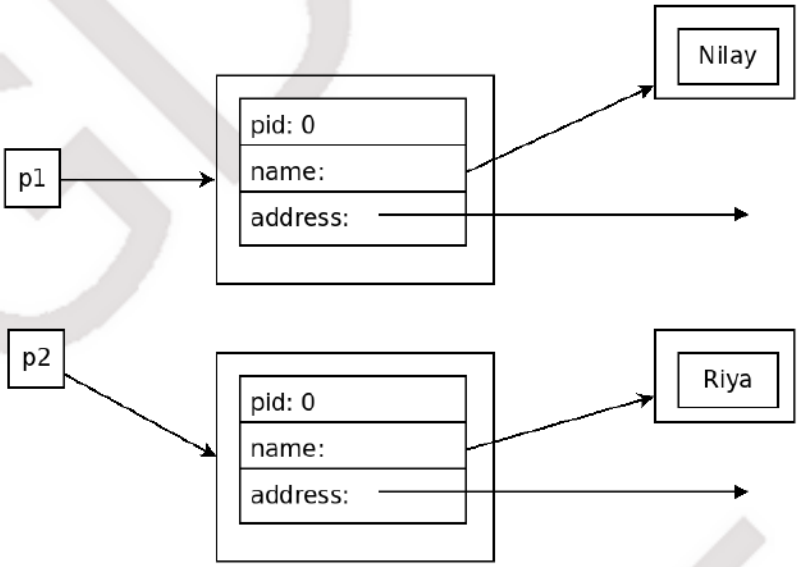
Two References Point to the Same Object

```
p1.name = "Nilay";  // Changing the name in p1
                    // This will chnage the name in p2 also
System.out.println(p2.name); // Nilay
System.out.println(p2.name); // Nilay
                    // There are no references pointing to the
                    // String object "Ritesh" now. Memory
        allocated
                    // to it will be freed by the garbage
        collector
                    // at some future point in time
```



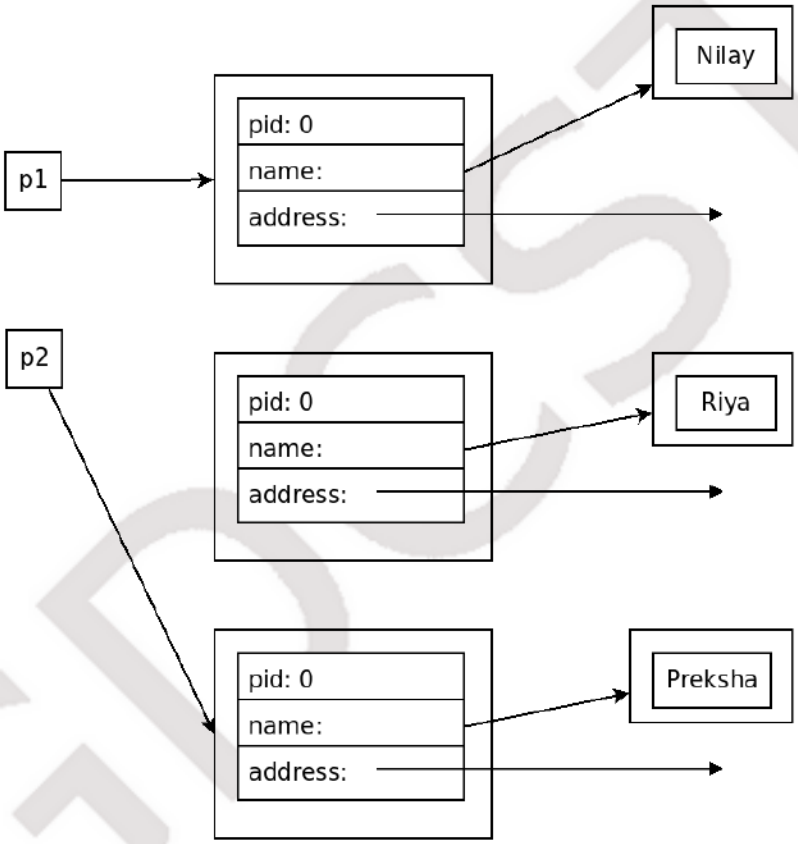
Changing the Name through One Reference

```
p2 = new Person(“Riya”); // now p2 points to a person different
        from p1
```



Two References Pointing to Different Objects

```
p2 = new Person(“Preksha”); // now p2 points to a different
        object
                    // There are no references pointing to the
        earlier Person
                    //  object now, memory allocated to it will
        be freed
                    // by the garbage collector at some future
        point in time.
                    // When that Person object is freed, there
        will be no
                    // references pointing to the String object
        "Riya" too.
                    // Hence, it will also be garbage collected
```



Reference Changed to Point to a Different Object

Practical Tips

Output

```
System.out.println("area=" + area);
//     System.out.println("area=" +
         Double.valueOf(area).toString());
```

```
System.out.print("ABC");
System.out.print("DEF");
System.out.print("HIJ");
System.out.println("ABC");
System.out.println("DEF");
System.out.println("HIJ");
/////// OUTPUT ///////
ABCDEFHIJ
ABC
DEF
HIJ
```

Input

```
import java.io.*;

public class Input1
{
  public static void main (String[] args)
  {
    double radius;
    BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
    String line=null;
    System.out.print("Enter the radius: ");
    try
    {
      line = br.readLine();
    }
    catch (IOException ex)
    {
//    Logger.getLogger(Main.class.getName()).log(Level.SEVERE,
        null, ex);
      System.err.println("Exception: " + ex.getMessage());
    }
    radius = Double.parseDouble(line);
    System.out.println("The diameter is " + radius*2);
  }
}
```

```
import java.util.Scanner;

public class Input2
{
    public static void main (String[] args)
    {
        Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter a line of text: ");
        String line = scanner.nextLine();
        System.out.println("You entered: " + line);
        System.out.print("Enter the radius of the circle: ");
        double radius = scanner.nextDouble();
        System.out.println("The diameter is " + radius*2);
        System.out.print("Enter the length of the side of the
         square: ");
        int side = scanner.nextInt();
        System.out.println("The area of the square is " + side *
         side);
    }
}
```

Strings in Java

- Strings in Java are objects of the special class java.lang.String
- Strings in Java are *not* an array of characters
- The `+` operator in the Java programming language is overloaded to provide both numeric addition and string concatenation
- Java strings are *immutable*, i.e. their contents cannot be changed. Any operation on a string or strings that requires modification of the contents of a string generates and returns a new string object as the existing string cannot be modified
- This way of working may generate a large number of intermediate string objects. Those will be removed by the garbage collector
- When performing an operation that is known to generate a large number of intermediate strings, it is better to use objects of StringBuffer or StringBuilder classes to improve performance
- The String class implements the CharSequence interface, which is the common interface for all classes that represent a sequence of characters
- Why are strings in Java immutable (interview question)
    - Strings are often used as keys in hash maps. Hence their hash codes need to be invariant. Also, the hash codes for string objects can be cached for improved performance if they will not change
    - Immutability ensures thread-safety (reduced complexity and increased performance in multi-threaded environments)
    - Security - The class loaders use string representations of class names to load classes. If a string can be changed, it may result in loading of a different class. Similarly, operations like opening of a file and network programming use strings to represent file names, remote host name /

network address, port number etc. They also run a lot of validation on these strings before making the actual system call. If the strings could be changed in-between the validation and system call, that would nullify the validation and pose grave a security risk

- Immutability enables caching of strings
- All string literals are stored in a string pool in the Java heap and two string literals with the same value are conflated into a single literal

```
String firstName = "Albert";
String middleName = "Hermann"
String lastName = "Einstein";

// Inefficient way, but the compiler usually converts this into
        the efficient way automatically
String fullName = firstName + " " + middleName + " " + lastName;

// Efficient way
StringBuilder sb = new StringBuilder();
sb.append(firstName).append(" ").append(middleName).append("
        ").append(lastName);
String fullName = sb.toString();
```

method	Description
String(String original)	Constructor to create a duplicate string object
char charAt(int index)	Returns the char value at the specified index
int compareTo(String anotherString)	Compares two strings lexicographically. The result is a negative integer if this String object lexicographically precedes the argument string. The result is a positive integer if this String object lexicographically follows the argument string. The result is zero if the strings are equal
int compareToIgnoreCase(String str)	Compares two strings lexicographically, ignoring case differences
String concat(String str)	Concatenates the specified string to the end of this string.

method	Description
boolean contains(CharSequence s)	Returns true if and only if this string contains the specified sequence of char values
boolean endsWith(String suffix)	Tests if this string ends with the specified suffix
boolean equals(Object anObject)	Compares this string to the specified object
boolean equalsIgnoreCase(String anotherString)	Compares this String to another String, ignoring case considerations
int indexOf(int ch)	Returns the index within this string of the first occurrence of the specified character
int indexOf(int ch, int fromIndex)	Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index
int indexOf(String str)	Returns the index within this string of the first occurrence of the specified substring
int indexOf(String str, int fromIndex)	Returns the index within this string of the first occurrence of the specified substring, starting at the specified index
boolean isEmpty()	Returns true if, and only if, length() is 0
int lastIndexOf(int ch)	Returns the index within this string of the last occurrence of the specified character
int lastIndexOf(int ch, int fromIndex)	Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index

method	Description
int lastIndexOf(String str)	Returns the index within this string of the last occurrence of the specified substring
int lastIndexOf(String str, int fromIndex)	Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index
int length()	Returns the length of this string
boolean matches(String regex)	Tells whether or not this string matches the given regular expression
String replace(char oldChar, char newChar)	Returns a string resulting from replacing all occurrences of oldChar in this string with newChar
String replace(CharSequence target, CharSequence replacement)	Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence
String replaceAll(String regex, String replacement)	Replaces each substring of this string that matches the given regular expression with the given replacement
String replaceFirst(String regex, String replacement)	Replaces the first substring of this string that matches the given regular expression with the given replacement
String[] split(String regex)	Splits this string around matches of the given regular expression
boolean startsWith(String prefix)	Tests if this string starts with the specified prefix
boolean startsWith(String prefix, int toffset)	Tests if the substring of this string beginning at the specified index starts with the specified prefix
String substring(int beginIndex, int endIndex)	Returns a string that is a substring of this string having characters from

method	Description
	`beginIndex` to `endIndex-1`
String substring(int beginIndex)	Returns a string that is a substring of this string having characters from `beginIndex` to the end of the string
String toLowerCase()	Converts all of the characters in this String to lower case using the rules of the default locale
String toUpperCase()	Converts all of the characters in this String to upper case using the rules of the default locale
String trim()	Returns a string whose value is this string, with any leading and trailing whitespace removed

The StringBuffer and StringBuilder Classes

- Because strings in Java are immutable; any operation that changes a string does not modify the existing string, but generates a new string object as a result. Certain sequence of operations on strings result in generation of a lot of intermediate strings. The StringBuffer and StringBuilder classes provide a way of performing those operations efficiently
- The only difference between StringBuffer and StringBuilder is that StringBuffer is synchronized and is thread-safe, while StringBuilder is not thread-safe, but more efficient due to lack of synchronization
- If multiple threads are goind to share an object, StringBuffer must be used. Otherwise, StringBuilder should be used
- StringHandlingExample1 project
- StringHandlingExample2 project
- StringHandlingExample3 project

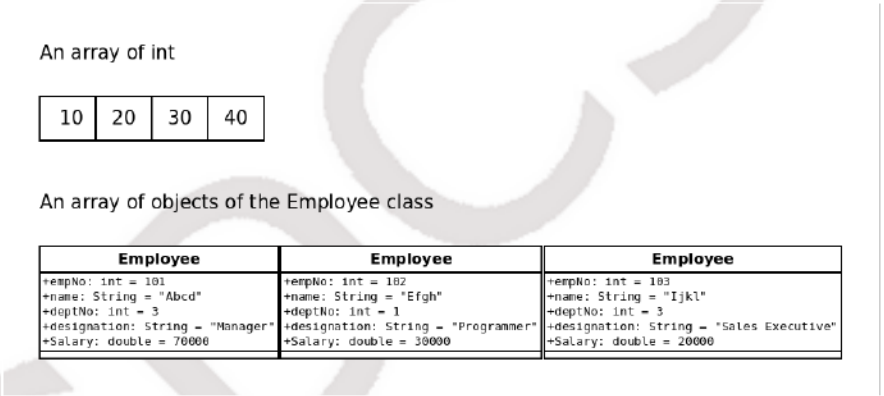
The NetBeans IDE

From inception, the NetBeans IDE has been closely associated with Java, even though now it also has support for C, C++, PHP. HTML5, JavaScript, etc. It was the first Java IDE written in Java itself. While NetBeans supports writing standalone Java programs, the usual way is to create a Java project in NetBeans and work on it. There can be several programs, modules and libraries in a project. NetBeans projects use a build tool like Apache ANT, Apache Maven, etc. NetBeans has several features like automatic code formatting, code completion, automatic code generation, etc., that make software development easier.

- Designed for Java
- Written in Java
- The concept of project
- Apache ANT based build system
- NetBeans Project Directory Structure
    - nbproject
    - src
    - build
    - dist
    - test

Arrays in Java

Arrays



Arrays

- An array is a sequence of homogeneous data items (data items of the same type) stored consecutively in memory
- An array data type is a reference type and all arrays are objects
- We can create an array of any data type. All the elemnts in the array must

have that same data type

- An array has a single name. Elements in the array are accessed through an integer array index. The first element in the array has index value 0, the second element in the array has index value 1, and so on. If the size of the array is `n`, then the last element in the array has index `n-1`
- The syntax for accessing elements of an array is `arrayName[indexValue]`. For example, `ary[3]`, `ary[i]`
- In java, array indices are always int and the first element in the array always has an index 0. Some other programming languages have 1 as the index of the first element in the array, while still other programming languages support associative arrays (maps) where the array index may also be of string or some other data type
- An array has a fixed size in Java, which is specified at the time of creating the array. The size of the array cannot be changed afterwards. It means that no element can be added or deleted from an array in Java
- Because elements in the array are stored in consecutive memory locations, access to element at a particular index in the array is very efficient using the address calculation formula
- A one-dimensional (1D) array is an array whose elements are not arrays themselves. For example, `int[] a = {1, 2, 3};`, `String[] b = {"abc", "defg", "h", "gh" };`, `Employee[] employees = { e1, e2 }` (where `Employee` is a class and `e1` and `e2` are objects of that class)
- A two-dimensional (2D) array is an array whose elements are one-dimensional arrays themselves. For example, `int[][] a = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };`, `String[][] b = { {"abc", "defg", "h"}, {"gh"} };`
- A three-dimensional (3D) array is an array whose elements are two-dimensional arrays
- An `n` dimensional array (where `n` is a natural number greater than 1) is an array whose elements are all `n-1` dimensional arrays

```
// 1-dimensional arrays

int[] a = { 1, 2, 3 };

String[] b = { "abc", "defg", "h", "gh" };

Employee e1 = new Employee();
Employee e2 = new Employee();
Employee[] employees = { e1, e2 };
```

```
// 2-dimensional arrays
float[][] a =
{
    { 1.2, 2.3, 3.4 },
    { 4.5, 5.6, 6.7 },
    { 7.8, 8.9, 9.0 }
};

String[][] b =
{
    { "abc", "defg", "h" },
    { "gh" }
};

int[][] matrix =
{
    { 11, 12, 13, 14 },
    { 21, 22, 23, 24 },
    { 31, 32, 33, 34 }
};

// 3-dimensional matrix

int[][][] identityMatrices =
{
    {
        { 1, 0 },
        { 0, 1 }
    },
    {
        { 1, 0, 0 },
        { 0, 1, 0 },
        { 0, 0, 1 }
    },
    {
        { 1, 0, 0, 0 },
        { 0, 1, 0, 0 },
        { 0, 0, 1, 0 },
        { 0, 0, 0, 1 }
    }
};
```

- Declaration and creation of arrays

```
int[] ai;                   // array of int

int ai[];                   // array of int

short[][] aas;              // array of array of short or
                            // two-dimensional array of short
```

```
short aas[][];              // array of array of short or
                            // two-dimensional array of short

short[] aas[];              // array of array of short or
                            // two-dimensional array of short

short aas[][];              // array of array of short or

short s, aas[][];           // scalar short, array of array of
        short

Object[] ao, otherAo;       // array of Object, array of Object

// The declarations above do not create array objects.
// All variables in the declarations above will be NULL.
// Array objects can be created using the new operator or using
        array literals
// The following examples are
// declarations of array variables that do create array objects:

int ai[] = new int[10];
Exception ae[] = new Exception[3];
Object aao[][] = new Object[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ',
                'S', 't', 'r', 'i', 'n', 'g' };
String[] aas = { "array", "of", "String", };
```

```
// Rectangular array
int[][] a = new int[3][5];

// Equivalent way of creating the above array
// int[][] a = new int[3][];
// a[0] = new int[5];
// a[1] = new int[5];
// a[2] = new int[5];

a[0][0] = 11;
a[0][1] = 12;
a[0][2] = 13;
a[0][3] = 14;
a[0][4] = 15;

a[1][0] = 21;
a[1][1] = 22;
a[1][2] = 23;
a[1][3] = 24;
a[1][4] = 25;

a[2][0] = 31;
```

```
a[2][1] = 32;
a[2][2] = 33;
a[2][3] = 34;
a[2][4] = 35;

// The above is equivalent to
int[][] a =
{
    { 11, 12, 13, 14, 15 },
    { 21, 22, 23, 24, 25 },
    { 31, 32, 33, 34, 35 }
};

for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a[i].length; j++)
        System.out.print(a[i][j] + " ");
    System.out.println();
}
/////// OUTPUT ///////
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
```

```
// Jagged array
int[][] a;
a = new int[4][];

a[0] = new int[2];
a[1] = new int[4];
a[2] = new int[3];
a[3] = new int[2];

a[0][0] = 11;
a[0][1] = 12;

a[1][0] = 21;
a[1][1] = 22;
a[1][2] = 23;
a[1][3] = 24;

a[2][0] = 31;
a[2][1] = 32;
a[2][2] = 33;

a[3][0] = 41;
a[3][1] = 42;

// The above is equivalent to
int[][] a =
{
```

```
    { 11, 12 },
    { 21, 22, 23, 24 },
    { 31, 32, 33 },
    { 41, 42 }
};

for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a[i].length; j++)
        System.out.print(a[i][j] + " ");
    System.out.println();
}
/////// OUTPUT ///////
11 12
21 22 23 24
31 32 33
41 42
```

```
// Accessing the length of an array
int len = ai.length;
```

- ArrayExample1 project
- ArrayExample2 project
- ArrayExample3 project
- CommandLineArguments1.java program
- CommandLineArguments2Arithmetic.java program

Object-oriented Programming in Java

- Classes and instances / objects (instance and object are synonyms)
- Fields (data members) and methods (member functions)
- Java supports Method Overloading
- Java supports Method Overriding
- Constructors
    - A constructor is a special method used to initialize an instance (object) of a class
    - A constructor has the same name as the class and no return type, not even void
    - A constructor can have zero or more arguments
    - A constructor with zero arguments is called a default constructor

- If a class defines no constructor, a default constructor (with no arguments) is automatically defined by Java. However, if one or more constructors with arguments are defined in a class, Java does not define the default no-arguments constructor automatically
- A constructor in a subclass may call the constructor of the parent class using the keyword *super*. The *super()* call, if present, must be the first statement in the constructor
- Whenever an instance of a class is created using the *new* operator, appropriate constructor of the class is called
- When an instance of a subclass is created using the *new* operator, the constructor of the subclass must invoke one of the constructors of the superclass using the *super* keyword as the first step in the initialization. If the subclass constructor does not invoke the parent class constructor like this, Java automatically calls the default (no-argument) constructor of the parent class. If the parent class does not have such no-argument constructor, the compiler reports an error

- Java does not have destructors. Objects are destroyed automatically by the garbage collector
- Packages
  - The concept of packages
  - Overview of the Java SE API packages
    - java.lang
      - › This package contains core classes for use in Java programming. java.lang is a special package. Classes in this package can be used in any class by their name without importing the package
    - java.util
      - › This package contains utility classes. It provides commonly used data structures and algorithms
    - java.io
      - › This package contains input/output related classes
    - java.nio
      - › This package contains new (improved) set of input/output related classes
    - java.nio2
      - › This package contains further improved set of input/output related classes
    - java.net
      - › This package contains networking related classes

        - java.awt
            - This package contains the AWT (Abstract Window Toolkit) GUI class library
        - javax.swing
            - This package contains the Swing GUI class library
        - javax.sql
            - This package contains common database-related classes
    - package statement
    - import statement
    - Correspondence between packages and directories
    - Using packages on the command line
        - Editing
        - Compiling
        - Executing

Method Overloading

- Java supports overloading of methods. One may define multiple methods with the same name in the same class. The methods must differ either in the number of arguments or the data types of arguments. Methods that differ only in the return type are not permitted in the same class. Where the method is called, the compiler determines which method is to be called using the number of arguments and data types of arguments. Method overloading is sometimes also known as compile-time polymorphism.

Operator Overloading

- Java does not permit programmers to overload operators
- However, the `+` operator is overloaded in the Java programming language itself to represent both numeric addition and string concatenation

Inheritance

- One class may inherit from another class
- The inheriting class is called the subclass or the derived class or the child class and the class from which it derives is called the superclass or the base class or the parent class
- All the non-private fields and methods of the superclass automatically become part of the subclass by virtue of inheritance. That is to say, the

subclass inherits all the non-private members of the superclass

- A subclass may redefine a method it inherited from the superclass. This is called overriding a method. If a subclass overrides a method it inherited from the superclass, all calls to that method on instances (objects) of that class resolve to the redefined method. If the overridden method in the subclass wants to call the method definition in the superclass, then it can use the *super* keyword. It is recommended, but not compulsory, to mark the overriding method with the `@Override` annotation
- Inheritance provides for code reuse and helps avoid code redundancy
- In java, a class may inherit from a single superclass only. Java does not support multiple inheritance
- Java uses the *extends* keyword to express inheritance

IS_A and HAS_A relationships

- If every instance of a class B is also an instance of a class A, then there is an IS_A relationship between B and A; namely B IS_A A
- Examples of IS_A relationship
  - Every employee is a person. Therefore, EMPLOYEE IS_A PERSON
  - Every manager is an employee. Therefore, MANAGER IS_A EMPLOYEE
  - Every person is not a taxpayer; therefore we cannot say PERSON IS_A TAXPAYER. Similarly, every taxpayer is not a person (companies and organizations may also be taxpayers); therefore we cannot say TAXPAYER IS_A PERSON. However, the definition of a *legal person* includes persons as well as organizations; hence TAXPAYER IS_A LEGAL_PERSON
- IS_A relationship is modeled using inheritance. If A IS_A B, then A should inherit from B
- If every instance of a class A contains an instance of class B, then there is a HAS_A relationship between A and B; namely A HAS_A B
- HAS_A relationship can be one-to-one or one-to-many
- Examples of HAS_A relationship
  - Every person has a name. Therefore, PERSON HAS_A NAME
  - Every company has (multiple) employees. Therefore, COMPANY HAS_A EMPLOYEE
- HAS_A relationship is modeled by containment. If A HAS_A B, then A should contain B as a member. If A can contain multiple instances of B,

then A should contain a list of B as a member. Thus, PERSON should contain a NAME as a member and COMPANY should contain a list of EMPLOYEE as a member

```
class Person
{
  int pid;
  String name;
  String address;

  Person(String name)
  {
    this.name = name;
  }

}

class Manager extends Person
{
}

Person p3 = new Manager(“Rina”); // OK, a Manager IS_A Person
        (always),
                                 // because Manager inherits from
        Person
Manager m1 = new Manager();      // OK, same type
Manager m2 = new Person(“Rina”); // Error - a Person not (always)
        IS_A  Manager
                                 // because Person does not
        inherit from Manager


class SomeClass
{
  public static void discussWith(Person p)
  {
  }
  public static void hire(Manager m)
  {
  }
}

SomeClass.discussWith(p1); // OK, same type
SomeClass.discussWith(m1); // OK, a Manager IS_A Person (always),
                           // because Manager inherits from
        Person

SomeClass.hire(m1);     //OK, same type
SomeClass.hire(p1);     // Error - a Person not (always) IS_A
        Manager,
```

```
                    // because Person does not inherits from
        Manager
```

- **Access Modifiers**
  - **None** Package access, the member can be accessed from within any class in the same package
  - **public** The member can be accessed from any class in any package
  - **private** The member can only be accessed from within the same class
  - **protected** The member can only be accessed from within the same class and its subclasses (in any package) as well as from within any class in the same package (because protected access is a superset of package access)

Permitted Access for Different Access Modifiers

	Same Class	Subclass in same package	Non-subclass in same package	Subclass in another package	Non-subclass in another package
private	Yes	No	No	No	No
package	Yes	Yes	Yes	No	No
protected	Yes	Yes	Yes	Yes[1]	No
public	Yes	Yes	Yes	Yes	Yes

[1] Only if accessed as its own member

- PackageExample01 project
- InheritanceExample1 project
- InheritanceExample2 project
- InheritanceExample3 project
- SubClassOverridingExample.java program

```
class Person
{

    private String name;
```

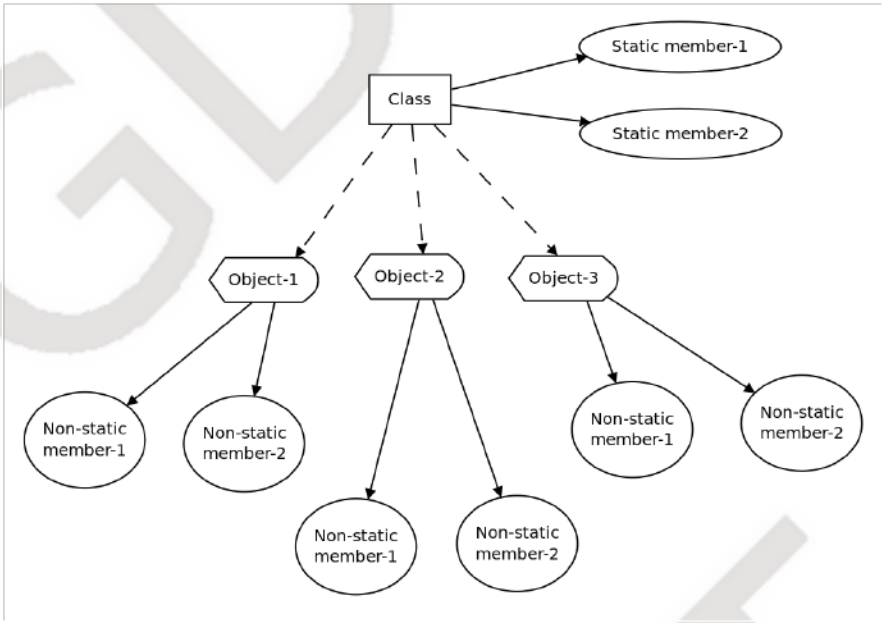
```
    public Person()
    {
        name = null;
    }

    public Person(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String newName)
    {
        name = newName;
    }

    public String getFirstName()
    {
        if (name == null)
        {
            return null;
        }
        if (name.indexOf(' ') == -1)
        {
            return name;
        }
        else
        {
            return name.substring(0, name.indexOf(' '));
        }
    }

}

class Employee extends Person
{

    private int empNo;
    private String designation;

    public Employee()
    {
        super();
        empNo = -1;
        designation = null;
    }
```

```
    public Employee(int empNo, String name, String designation)
    {
        super(name);
        this.empNo = empNo;
        this.designation = designation;
    }

    public int getEmpNo()
    {
        return empNo;
    }

    public void setEmpNo(int newEMpNo)
    {
        empNo = newEMpNo;
    }

    public String getDesignation()
    {
        return designation;
    }

    public void setDesignation(String newDesignation)
    {
        designation = newDesignation;
    }

    public boolean isProgrammer()
    {
        if (designation != null &&
        designation.equals("Programmer"))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

}

class Manager extends Employee
{

    private String departmentName;

    public Manager()
    {
        super();
        departmentName = null;
    }
```

```
    public Manager(int empNo, String name,
            String designation,
            String departmentName)
    {
        super(empNo, name, designation);
        this.departmentName = departmentName;
    }

    public String getDepartmentName()
    {
        return departmentName;
    }

    public void setDepartmentName(String newDepartmentName)
    {
        departmentName = newDepartmentName;
    }

}
```

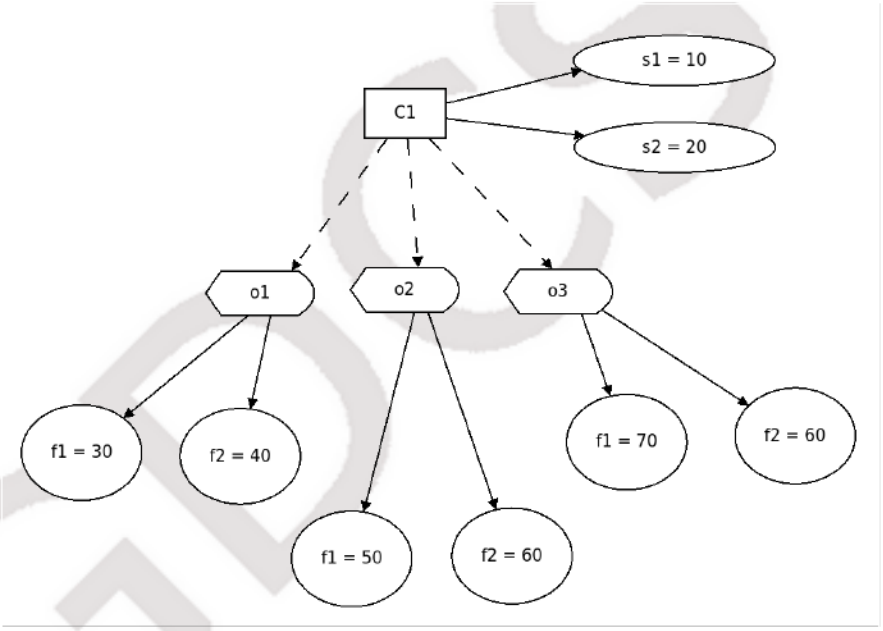
Static Members v/s Non-static Members

- A static member belongs to the class, not to the individual instances (objects)
- For every class, there is only one set of static members irrespective of the number of instances (objects) created
- A static member is accessed using the syntax Class.member
- A non-static member (instance member) belongs to the instances (objects) created, not to the class
- There is a separate set of non-static member for every instance (object) of the class
- A non-static member can be accessed using the syntax object.member where object must be a valid instance of that class. If object is null, NullPointerException is thrown



Static v/s Non-static Fields-1

```
class C1
{
    static int s1 = 10;
    static int s2 = 20;
    int f1;
    int f2;
}

. . .

C1 o1 = new C1();
o1.f1 = 30;
o1.f2 = 40;
C1 o2 = new C1();
o2.f1 = 50;
o2.f2 = 60;
C1 o3 = new C1();
o3.f1 = 70;
o3.f2 = 60;
```



Static v/s Non-static Fields-2

Static Methods v/s Non-static Methods

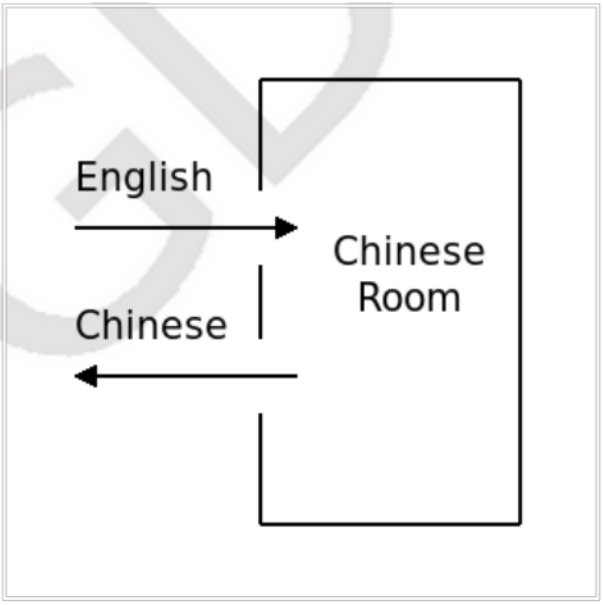
- Just like a static field, a static method belongs to the class, not individual instances (objects). A non-static method belongs to an instance (object) of the class
- A static method can only access static fields and static methods of the class. To access a non-static member of the class, it needs an object of the class
- A non-static method can access both static as well as non-static members of the class
- Every non-static method is passed an implicit argument called *this*, which is a reference to the instance / object on which the method is invoked. All non-static members of the class are implicitly accessed through the *this* reference
- Static methods belong to the class, not to individual instances / objects. Hence they are not passed the implicit argument *this*
- StaticExample01.java

Encapsulation

- Encapsulation is used to enforce a separation between the implementation (internal logic) of a class and its external interface

- Encapsulation allows the programmer to change the internal logic of a class without affecting or breaking client code
- Encapsulation is used to bundle a set of data and all possible operations on them in a class. The data are hidden from the client code and can only be accessed and manipulated through the operations provided in the form of public methods
- Thus a class encapsulates or bundles together *data* and *operations on the data* in a single entity
- Encapsulation requires that the fields of a class are not exposed to other classes directly. Rather, access methods (getter and setter methods) are provided to access them where needed. The fields themselves should be private or protected, while the methods providing operations (including getters and setters) can be public
- Encapsulation enforces *information hiding*. This reduces *coupling* in software
- When there is a change in the internal logic, it should be possible to carry out the change by changing the fields and modifying the implementation of the operation methods suitably, without changing their public interfaces (public method signatures)

Encapsulation Example - ATM



Encapsulation Example - Chinese Room

- EncapsulationExample1 project
- EncapsulationExample2 project
- EncapsulationExample3 project
- EncapsulationExample4 project

Abstract Methods and Classes

- An abstract method is a method declaration without the corresponding implementation (code)
- An abstract method must be marked with the keyword *abstract*
- An abstract class is a class that contains one or more abstract methods
- An abstract class must be marked with the keyword `abstract`
- An abstract class is an incomplete class in the sense that some of the methods lack implementation. Hence abstract classes cannot be instantiated (we cannot create objects of abstract classes)
- A class in which there are no abstract methods (all methods have their implementation or code) is called a *concrete* class
- A subclass of an abstract class can be created. This subclass may or may not provide implementation for some of the abstract methods in the superclass. If a subclass provides implementation for all the abstract methods in the superclass, then it becomes a *concrete* (regular) class and can be instantiated as usual. A subclass that does not provide implementation for all the abstract methods in the superclass remains abstract itself

- Just as a class provides a blueprint that is used to create objects, an abstract class provides a partial template or blueprint that can be used to create subclasses. At the minimum, all direct and indirect concrete subclasses of an abstract class will have all the methods (abstract as well as non-abstract) of that abstract class defined in it. Thus, there will be a minimum guaranteed functionality in all the non-abstract subclasses of the abstract class. Those subclasses may have additional members and functionality of their own

Final Classes

A class declared using the keyword `final` is known as a final class. The implementation of the "final" class is fixed and cannot be overridden. To enforce this requirement, java does not allow inheriting from a final class. We cannot define a class that extends a final class.

Interfaces

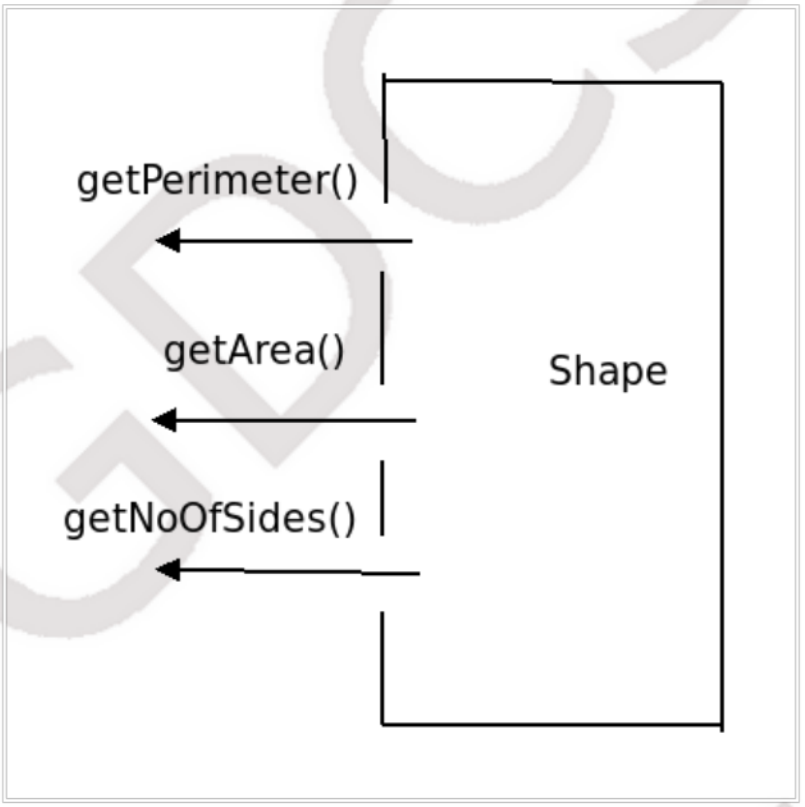
- An interface is a set of operations (methods) used for accessing and manipulating a class or its objects by the client code (other classes)
- Interface naming follows class naming conventions
- Encapsulation requires that a class must only be accessed through its interface
- An interface hides implementation details from the client code. It helps in implementing information hiding
- A Java interface contains a set of method declarations without implementation code (starting with Java 9, default implementation can be provided)
- A class that contains ***all*** the methods in an interface and is declared as such is said to *implement* that interface. This is signified using the keyword `implements` (`class C1 implements I1`)
- A class may implement zero or more interfaces
- An interface provides partial template for classes. All the classes implementing the interface must provide the minimum functionality specified in that interface (in the form of methods). They may provide additional

functionality or implement additional interfaces

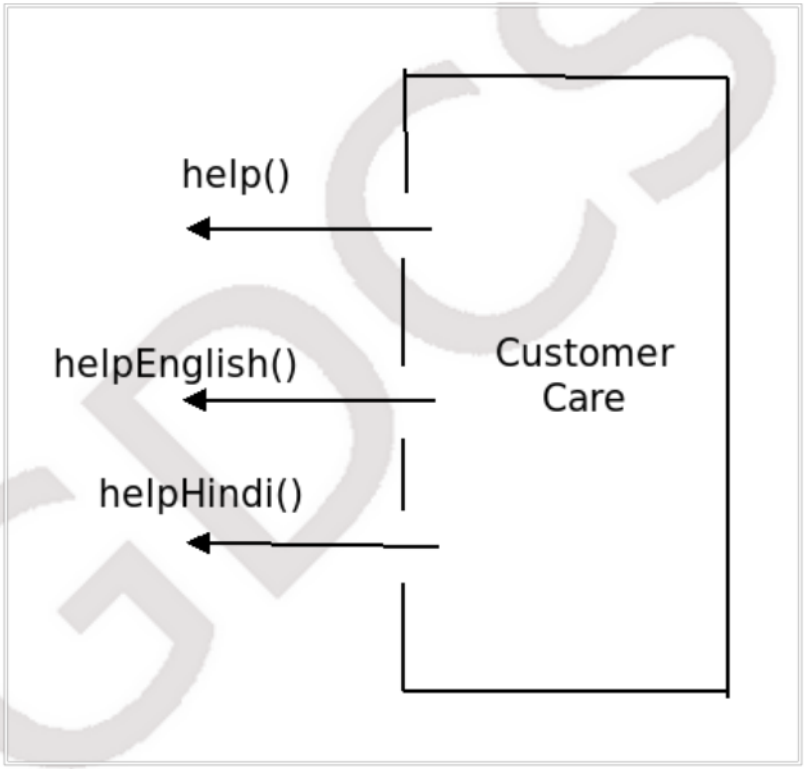
- Interfaces can be used as data types for declaring variables and method arguments
- A variable of interface type can be initialized with a reference to an instance of any class that implements that interface
- Similarly, when a method has a formal parameter of interface type, an instance of any class that implements that interface can be passed as the corresponding actual parameter
- InterfaceExample01.java program
- InterfaceExample02Iterable.java program

```
interface ChineseRoom
{
    void enter(String englishText);
    String getChineseTranslation();
}

// CNN neural networks
class ChineseTranslator1 implements ChineseRoom
{
    void enter(String englishText)
    {
        ...
        ...
    }
    String getChineseTranslation()
    {
        ...
        ...
    }
}

// Statistical approach
class ChineseTranslator2 implements ChineseRoom
{
    void enter(String englishText)
    {
        ...
        ...
    }
    String getChineseTranslation()
    {
        ...
        ...
```

```
    }
}

class SomeOtherClass
{
//    ChineseTranslator1 chineseTranslator1;
    ChineseRoom chineseRoom;

//    chineseRoom = new ChineseRoom(); // Error

    chineseRoom = new ChineseTranslator1();
    chineseRoom.enter("Good morning!");
    String chineseTranslation =
        chineseRoom.getChineseTranslation();

    chineseRoom = new ChineseTranslator2();
    chineseRoom.enter("Good morning!");
    String chineseTranslation =
        chineseRoom.getChineseTranslation();

    ...
    ...
}
```



Interface Example - Shapes



Interface Example - Customer Care

- InterfaceExample1Shapes project
- InterfaceExample2CustomerCare project
- InterfaceExample02Iterable.java program

Method Overriding

If a method has been defined in a superclass and am subclass defines a method with the same name and same arguments (parameters), then it is said that the subclass overrides the definition of the method in the superclass. In other words, method overriding occurs when a subclass redefines a method it has inherited from a superclass. In this case; when the method is invoked on an instance of the superclass, the method defined in the superclass is called, while when the method is invoked on an instance of the subclass, the method defined in the subclass is called. This an example of polymorphism. Method overriding is sometimes also known as runtime polymorphism.

Even though it is not compulsory, it is recommended that when a subclass defines a method that overrides a superclass method, the subclass indicates that fact to the compiler by annotating the method using the `@Override` annotation.

If a method is defined using the `final` keyword, it cannot be overridden.

```java
class Employee {
    String name;

    public Employee(String name) {
        this.name = name;
    }

    public void describeSelf() {
        System.out.println("Hi! I am an ***employee***. My name
        is " + name);
    }

}

class Manager extends Employee {

    public Manager(String name) {
        super(name);
    }

    @Override
    public void describeSelf() {
        System.out.println("Hi! I am a ***manager***. My name is
        " + name);
    }
}

public class MethodOverriding
{
    public static void main (String[] args)
    {
        Employee e1 = new Employee("Abc");
        e1.describeSelf(); // describeSelf() method called on
        superclass object
        Manager m1 = new Manager("Xyz");
        m1.describeSelf(); // describeSelf() method called on
        subclass object
    }

}
/////// OUTPUT ///////
Hi! I am an ***employee***. My name is Abc
Hi! I am a ***manager***. My name is Xyz
```

```java
public final void finalMethod() {
    System.out.println("final method");
    System.out.println("Cannot be overridden by a subclass");
}
```

- MethodOverriding.java program

Polymorphism

- Polymorphism is the ability of different objects with the same compile-time type but different runtime types to behave differently according to their runtime types
- In object-oriented programming, calling a method on an object is seen as passing a message to that object. Polymorphism is the ability of different objects with the same compile-time type but different runtime type to react differently to the same message according to their runtime type
- Method Overriding is an example of polymorphism. Sometimes, it is also called rumtime polymorphism.
- Sometimes, Method Overloading is called compile-time polymorphism.
- Polymorphism1.java program
- Polymorphism2.java program

Some Important Java Classes

java.lang.Object

- The class java.lang.Object is the root of the Java class hierarchy
- All classes inherit, directly or indirectly, from Object
- Object is the superclass of all Java classes
- The Object class contains some common methods that are inherited by all classes

method	Description
public int hashCode();	Returns a hash code value for the object
public boolean equals(Object otherObject);	Indicates whether some other object is *equal to* this object. The definition of when two objects are considered equal depends on their class.
public String toString()	Returns a string representation of the object.

Hash Code

- A hash is a mathematical function that maps arbitrary amounts of data (textual or binary) to a fixed sized integer value
- If the amount of data is greater than the information contents of the integer hash, several sets of data may map to the same hash value
- A hash function tries to distribute all possible input across the hash values evenly
- hashes play important role in detecting changes, map-like data structures and cryptography
- The default implementation of `hashCode()` in java.lang.Object returns a different number for different objects in most cases, even if their contents are same
- In Java, if a class overrides `equals()`, it must also override `hashCode()`

Equality checks

- The `==` and `!=` operators should only be used to check equality of values of primitive type
- If used on reference types, the `==` and `!=` operators compare the references (pointers) rather than the objects themselves. That means, they consider two references equal if both point to the same object, and different if they point to different objetcs
- The `equals()` method should always be used to compare objects
- Every class is supposed to override the `equals()` (and, conequently, `hashCode()`) methods
- If the class does not override the `equals()` method, the default implementation in java.lang.Object is used. It compares objects by comparing their hashCode
- EqualsMethodEqualToOperator.java program

toString()

- Every Java class is supposed to override the `toString()` method
- If a class does not provide its own impmenetation of the `toString()` method, the default implementation in java.lang.Object is used. It converts an object to a string by appending an @ sign and the hash code of the

object in hexadecimal to the name of the class

- Company.java program

Input / Output

In Java, input/output operations are abstracted as if they occur from/to streams.

- A stream is a sequence of units
- A binary stream is a sequence of bytes
- A character stream is a sequence of Unicode characters
- A character stream is always based on an underlying binary stream, but it performs the necessary conversion between Unicode character representations and bytes
- An input stream is a stream from which input may be obtained
- An output stream is a stream to which output may be sent

Naming Conventions in the Java SE API

- A class implementing an input stream has the word Input before the word Stream in the name
- A class implementing an output stream has the word Output before the word Stream in the name
- A class implementing a character stream has the word Reader or Writer in its name, depending on whether it is an input stream or output stream respectively. A class not having either of the words Reader or Writer in its name implements a binary stream
- Classes performing input/output from/to a specific type of source have the name of the source as part of the class name. E.g. FileInputStream, ByteArrayOutputStream, FileReader, StringWriter
- Classes that perform buffering of input/output rather than writing the data immediately (to improve performance) have name starting with the word Buffered

Standard Methods

- A class implementing a binary input stream (*InputStream) has a method int read() that reads a single byte from the stream and returns it, the method returns -1 on EOF and throws IOException on error
- A class implementing a character input stream (*Reader) has a constructor

to construct an object based on its source (InputStream/File/String). It also has a method int read() that reads a single character from the stream and returns it as a Unicode code point, the method returns -1 on EOF and throws IOException on error

- A *Reader causes a read operation on the underlying binary input stream for every character. For efficient operations, a BufferedReader should be created based on the* Reader. A BufferedReader has a constructor that accepts a Reader. It improves performance by buffering the input. It too provides a method int read() that reads a single character from the stream and returns it as a Unicode code point, the method returns -1 on EOF and throws IOException on error. It also provides a String readLine() method that reads one line from the stream and returns it, the method returns null on EOF and throws IOException on error
- A class implementing a binary output stream (*OutputStream) has a method void write(int byte) that writes a single byte to the stream, the method throws IOException on error. It also has a method void write(byte[] byteArray) that writes the whole byte array to the stream, the method throws IOException on error
- A class implementing a character output stream (*Writer) has a constructor to construct an object based on its destination (OutputStream/File/String). It has a method void write(int ch) that writes a single character to the stream and throws IOException on error. It also has methods to write whole strings and character arrays to the stream
- A *Writer causes a write operation on the underlying binary output stream for every character. For efficient operations, a BufferedWriter should be created based on the* Writer. A BufferedWriter has a constructor that accepts a Writer. It improves performance by buffering the output. It too provides a method void write(int ch) that writes a single character to the stream and throws IOException on error. It also has methods to write whole of or selected portions of strings and character arrays as well as a method to write the (platform-specific) newline to the stream

Important Notes:

- All streams are implicitly opened when the stream object is constructed. The system may allocate resources to support the stream operations when the stream is opened
- All streams should be closed using the void close() method provided in the stream class when the stream is no longer needed. This will release the

resources associated with the stream. Usually, when an enclosing class is closed, it automatically closes the underlying stream. E.g., closing a BufferedReader automatically closes the underlying Reader, which, in turn, closes the underlying InputStream

- Closing the output streams, especially the buffered ones, ensures that any data pending to be written are actually written to the destination
- Closing a stream based on a file ensures that the file is properly closed and locks, if any, are released
- All output streams also support the void flush() method that immediately writes to the destination any data pending to be written

The I/O Classes

- InputStream - An abstract class representing a binary input stream, the static object System.in is an instance of InputStream representing the standard input of the Java console (if available)
    - FileInputStream - obtains binary input from a file
    - ByteArrayInputStream - obtains binary input from an array of bytes
- Reader - An abstract character input stream based on an underlying binary stream
    - InputStreamReader - Reads characters from the underlying InputStream
        - FileReader - Reads characters from the underlying text file
    - StringReader - Reads characters from the underlying String
    - BufferedReader - Buffered Reader based on a *Reader
- OutputStream - An abstract class representing a binary output stream
    - FileOutputStream - Writes binary output to a file
    - ByteArrayOutputStream - Writes binary output to an array of bytes
        - FilterOutputStream - A generic superclass for all classes that filter/modify/transform output before writing to the underlying output stream. This class itself does no filtering, but its subclasses may do so
            - PrintStream - Provides the following facilities
                - to output values of different types conveniently, supports both binary and character output
                - to flush the stream automatically when a newline is output
                - to perform well-formatted output using the format()

method (somewhat similar to printf() in C)
The static objects System.out and System.err are instances of PrintStream representing the standard output and standard error respectively of the Java console (if available)
› LogStream - Used to write logs

- Writer - An abstract character output stream based on an underlying binary stream
  - OutputStreamWriter - Writes characters to the underlying OutputStream
    • FileWriter - Writes characters to the underlying text file
  - StringWriter - Writes characters to the underlying String
  - BufferedWriter - Buffered Writer based on a *Writer
  - PrintWriter - Similar to PrintStream, but supports only character output
- java.util.Scanner - Utility class for scanning / parsing the input and extracting tokens of different types

	Binary	**Character**
Input	**InputStream** (abstract class) FileInputStream ByteArrayInputStream `int read()` - reads a byte	**Reader** (abstract class) InputStreamReader FileReader StringReader `int read()` - reads a char
Output	**OutputStream** (abstract class) FileOutputStream ByteArrayOutputStream FilterOutputStream PrintStream (can output primitive types and flush the stream) `void write(int b)` - writes a byte `void write (byte[] ba)` - writes a byte array	**Writer** (abstract class) OutputStreamWriter FileWriter StringWriter PrintWriter (formatted output) `void write()` methods to write a char, String or character array

	Binary	Character
Notes		**Readers** and **Writers** are based on some **InputStream** and **OutputStream** respectively.
		BufferedReader wraps around a **Reader** and provides buffering for efficiency and recognizes lines `String readLine()` - reads a line
		BufferedWriter wraps around a **Writer** and provides buffering for efficiency and platform-specific newlines `void newLine()` - writes a platform-specific newline

		PrintWriter contains the following methods for formatted output
		`PrintWriter format(String format, Object... args)`
		`PrintWriter printf(String format, Object... args)`
		`void println()` methods to write the argument provided, followed by newline

		The **System** class has three objects for input / output from / to the terminal **System.in** corresponds to standard input and extends InputStream **System.out** corresponds to standard output and extends PrintStream **System.err** corresponds to standard error and extends

	Binary	Character
		PrintStream
		It is important to make sure that any stream, reader, writer, etc. we open are **closed** properly. They have a `close()` method for this purpose

java.util.Scanner

Constructors

Constructor	Description
`Scanner(File source)`	Constructs a new Scanner that produces values scanned from the specified file
`Scanner(InputStream source)`	Constructs a new Scanner that produces values scanned from the specified input stream
`Scanner(String source)`	Constructs a new Scanner that produces values scanned from the specified string

Methods

Method	Description
`void close()`	Closes this scanner
`boolean hasNextBoolean()`	Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false"
`boolean hasNextByte()`	Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the nextByte() method
`boolean hasNextByte(int radix)`	Returns true if the next token in this scanner's input can be interpreted as a byte

Method	Description
	value in the specified radix using the nextByte() method
`boolean hasNextDouble()`	Returns true if the next token in this scanner's input can be interpreted as a double value using the nextDouble() method
`boolean hasNextFloat()`	Returns true if the next token in this scanner's input can be interpreted as a float value using the nextFloat() method
`boolean hasNextInt()`	Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the nextInt() method
`boolean hasNextInt(int radix)`	Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the nextInt() method
`boolean hasNextLine()`	Returns true if there is another line in the input of this scanner
`boolean hasNextLong()`	Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the nextLong() method
`boolean hasNextLong(int radix)`	Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the nextLong() method
`boolean hasNextShort()`	Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the nextShort() method

Method	Description
`boolean hasNextShort(int radix)`	Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the nextShort() method
`String next(String pattern)`	Returns the next token if it matches the pattern constructed from the specified string
`String next(Pattern pattern)`	Returns the next token if it matches the specified pattern
`boolean nextBoolean()`	Scans the next token of the input into a boolean value and returns that value
`byte nextByte()`	Scans the next token of the input as a byte
`byte nextByte(int radix)`	Scans the next token of the input as a byte
`double nextDouble()`	Scans the next token of the input as a double
`float nextFloat()`	Scans the next token of the input as a float
`int nextInt()`	Scans the next token of the input as an int
`int nextInt(int radix)`	Scans the next token of the input as an int
`String nextLine()`	Advances this scanner past the current line and returns the input that was skipped
`long nextLong()`	Scans the next token of the input as a long
`long nextLong(int radix)`	Scans the next token of the input as a long
`short nextShort()`	Scans the next token of the input as a short
`short nextShort(int radix)`	Scans the next token of the input as a short
`Scanner skip(String pattern)`	Skips input that matches a pattern constructed from the specified string
`Scanner skip(Pattern pattern)`	Skips input that matches the specified pattern, ignoring delimiters

Method	Description
`Scanner useDelimiter(String pattern)`	Sets this scanner's delimiting pattern to a pattern constructed from the specified String
`Scanner useDelimiter(Pattern pattern)`	Sets this scanner's delimiting pattern to the specified pattern

- InputOutputExample1 project
- InputStreamOutputStreamExample1 project

File Handling

The File Class

The `File` class represents a file or directory.

```
// A very simple Notepad program using Swing

public class NotepadFrame extends javax.swing.JFrame {

    /**
     * Creates new form NotepadFrME
     */
    public NotepadFrame() {
        initComponents();
//            FileReader isr = new FileReader("my-content.txt");
            InputStreamReader isr;
        try {
            isr = new InputStreamReader(
                    new FileInputStream("my-content.txt"));
            BufferedReader br = new BufferedReader(isr);
            StringBuilder sb = new StringBuilder();
            String line;
            while (true) {
                line = br.readLine();
                if (line == null) {
                    break;
                } else {
                    sb.append(line).append("\n");
                }
            }
            br.close();
            jTextArea1.setText(sb.toString());
        } catch (FileNotFoundException ex) {
```

```
        Logger.getLogger(NotepadFrame.class.getName()).log(Level.SEV
        null, ex);
        } catch (IOException ex) {

        Logger.getLogger(NotepadFrame.class.getName()).log(Level.SEV
        null, ex);
        }
    }

    /**
     * This method is called from within the constructor to
        initialize the form.
     * WARNING: Do NOT modify this code. The content of this
        method is always
     * regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated
        Code">//GEN-BEGIN:initComponents
    private void initComponents() {

        jScrollPane1 = new javax.swing.JScrollPane();
        jTextArea1 = new javax.swing.JTextArea();
        jlFilename = new javax.swing.JLabel();
        jbSave = new javax.swing.JButton();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON

        jTextArea1.setColumns(20);
        jTextArea1.setRows(5);
        jScrollPane1.setViewportView(jTextArea1);

        jbSave.setText("Save");
        jbSave.addActionListener(new
        java.awt.event.ActionListener() {
            public void
        actionPerformed(java.awt.event.ActionEvent evt) {
                jbSaveActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout layout = new
        javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(

        layout.createParallelGroup(javax.swing.GroupLayout.Alignment
            .addGroup(layout.createSequentialGroup()
                .addContainerGap()

        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout
                    .addComponent(jScrollPane1)
                    .addGroup(layout.createSequentialGroup()
```

```
                        .addComponent(jlFilename)
                        .addGap(0, 0, Short.MAX_VALUE))))
            .addGroup(layout.createSequentialGroup()
                .addGap(319, 319, 319)
                .addComponent(jbSave)
                .addGap(0, 363, Short.MAX_VALUE))
        );
        layout.setVerticalGroup(

        layout.createParallelGroup(javax.swing.GroupLayout.Alignment
            .addGroup(layout.createSequentialGroup()
                .addGap(10, 10, 10)
                .addComponent(jlFilename)

        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
                .addComponent(jScrollPane1,
        javax.swing.GroupLayout.PREFERRED_SIZE, 364,
        javax.swing.GroupLayout.PREFERRED_SIZE)

        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement
                .addComponent(jbSave)
                .addContainerGap(28, Short.MAX_VALUE))
        );

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void jbSaveActionPerformed(java.awt.event.ActionEvent
        evt) {//GEN-FIRST:event_jbSaveActionPerformed
        String content = jTextArea1.getText();
        try {
//            FileWriter bw = new FileWriter("my-content.txt");
            BufferedWriter bw = new BufferedWriter(
                    new OutputStreamWriter(
                            new FileOutputStream("my-
        content.txt")));
            bw.write(content);
            bw.close();
        } catch (FileNotFoundException ex) {

        Logger.getLogger(NotepadFrame.class.getName()).log(Level.SEV
        null, ex);
        } catch (IOException ex) {

        Logger.getLogger(NotepadFrame.class.getName()).log(Level.SEV
        null, ex);
        }
    }//GEN-LAST:event_jbSaveActionPerformed

    /**
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        /* Set the Nimbus look and feel */
```

```
        //<editor-fold defaultstate="collapsed" desc=" Look and
        feel setting code (optional) ">
        /* If Nimbus (introduced in Java SE 6) is not available,
        stay with the default look and feel.
         * For details see http://download.oracle.com/javase
        /tutorial/uiswing/lookandfeel/plaf.html
         */
        try {
            for (javax.swing.UIManager.LookAndFeelInfo info :
        javax.swing.UIManager.getInstalledLookAndFeels()) {
                if ("Nimbus".equals(info.getName())) {

        javax.swing.UIManager.setLookAndFeel(info.getClassName());
                    break;
                }
            }
        } catch (ClassNotFoundException ex) {

        java.util.logging.Logger.getLogger(NotepadFrame.class.getNar
        null, ex);
        } catch (InstantiationException ex) {

        java.util.logging.Logger.getLogger(NotepadFrame.class.getNar
        null, ex);
        } catch (IllegalAccessException ex) {

        java.util.logging.Logger.getLogger(NotepadFrame.class.getNar
        null, ex);
        } catch (javax.swing.UnsupportedLookAndFeelException ex)
        {

        java.util.logging.Logger.getLogger(NotepadFrame.class.getNar
        null, ex);
        }
        //</editor-fold>
        //</editor-fold>

        /* Create and display the form */
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new NotepadFrame().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JTextArea jTextArea1;
    private javax.swing.JButton jbSave;
    private javax.swing.JLabel jlFilename;
    // End of variables declaration//GEN-END:variables
}
```

- Notepad1 project
- FileCopy.java program
- FileFilterTextData.java program
- FileInputStreamFileOutputStreamExample1 project
- FileReaderFileWriterExample1 project

Exception Handling

An exception is an abnormal or unusual situation. At any point in the execution of our program, the JVM can be in one of the two states - normal state and exception state. In normal state, it continues to execute the normal code (not the exception handling code). When an exception (error condition) occurs, the code in which the error condition occurs *throws* an exception (an exception is usually an object of the `java.lang.Exception` class or one of its subclasses). The exception needs to be *caught* by an exception handler. As soon as an exception is thrown, the JVM suspends the execution of normal code and starts looking for an exception handler that catches that exception. It looks for an exception handler in the current block, if it is not found in the current block than its outer block, if it is not found in the outer block than still outer block, and so on. If an exception handler catching the current exception handler is not found, the JVM terminates the program with an error and outputs a *stack trace* on the standard error. If a suitable exception handler is found, the code in the exception handler is executed. After this, JVM assumes that the exception condition has been resolved and resumes normal execution *from the next statement after the exception handler*, not from where it had suspended the normal execution.

Code that *may* throw an exception should be written inside a `try` block, which should be followed by one or more `catch` blocks catching various possible exceptions.

```
public class ExceptionHandlingNew2 {

    public static void main(String[] args) {
//        String input = null;
        String input;
        System.out.print("Enter an integer: ");
        BufferedReader br = new BufferedReader(new
         InputStreamReader(System.in));
        try {
```

```
            input = br.readLine();
            int no1;
            try {
                no1 = Integer.parseInt(input);
                System.out.println("no1=" + no1);
            } catch (NumberFormatException nfe) {
                System.err.println("Enter a valid integer");
            }
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
//            input = "";
        }
    }

}
```

```
public class ExceptionHandlingNew3 {

    public static void main(String[] args) {
        String input;
        System.out.print("Enter an integer: ");
        BufferedReader br = new BufferedReader(new
         InputStreamReader(System.in));
        try {
            input = br.readLine();
            int no1;
            no1 = Integer.parseInt(input);
            System.out.println("no1=" + no1);
        } catch (IOException ioe) {
            System.err.println("IOException: " + ioe);
        } catch (NumberFormatException nfe) {
            System.err.println("Invalid integer entered");
//        } catch (Exception ex) {
//            System.err.println("Some error occured" + ex);
        }
    }
}
```

- ExceptionHandlingNew1 project
- ExceptionHandlingNew2 project
- ExceptionHandlingNew3 project
- ExceptionHandlingDisplayFile01.java program
- ExceptionHandlingDisplayFile02.java program
- ExceptionHandlingDisplayFile03.java program

- ExceptionHandlingDisplayFile04.java program
- ExceptionHandlingDisplayFile05.java program
- ExceptionHandlingDisplayFile06.java program
- ExceptionHandlingDisplayFile07.java program
- ExceptionHandlingDisplayFile08.java program
- ExceptionHandlingDisplayFile09.java program
- ExceptionHandlingExample5.java program
- ExceptionHandlingExample1 project
- ExceptionHandlingExample2 project
- ExceptionHandlingExample3 project
- ExceptionHandlingExample4 project
- ExceptionHandlingExample5 project
- ExceptionHandlingExample6 project
- ExceptionHandlingExample7 project
- ExceptionHandlingExample8 project
- ExceptionHandlingExample9 project
- ExceptionHandlingExample10 project
- ExceptionHandlingExample11 project

Some Important Classes from the java.util Package

The java.util.List Interface

This interface represents a dynamically-sized list of elements. The list maintains the order of the elements. It also allows duplicate elements.

The java.util.Set Interface

This interface represents a dynamically-sized set of elements. The set does not maintain the order of the elements. Further, it does not allow duplicate elements.

The java.util.ArrayList Class

This class implements the `java.util.List` interface using a resizeable array.

```
ArrayList<String> names = new ArrayList<>();
names.add("aaa");
names.add("bbb");
System.out.println(names.size());
names.add("ccc");
System.out.println(names.size());
for (String name : names) {
    System.out.println(name);
}
for (int i = 0; i < names.size(); i++) {
    System.out.println(names[i]);
}
```

The java.util.Scanner Class

See java.util.scanner

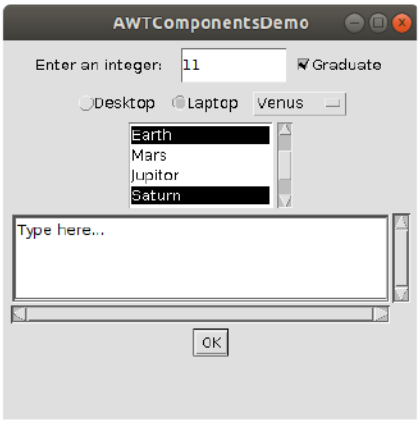
The java.util.Date Class

This class is used to store and manipulate datetime values.

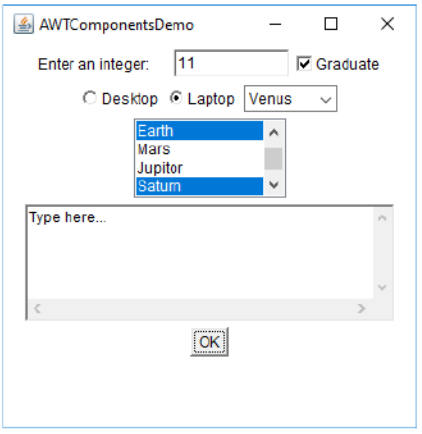
Graphical User Interface (GUI) Programming

- GUI toolkits
  - AWT (Abstract Window Toolkit)
    - package: java.awt
    - The *original* Java GUI toolkit
    - A *heavyweight* toolkit
    - It uses the native system libraries for creating GUI components
    - Every AWT component has a corresponding native *peer* component. That is why it is called a heavyweight toolkit
    - Because AWT uses native components, look and feel of components differ from platform to platform

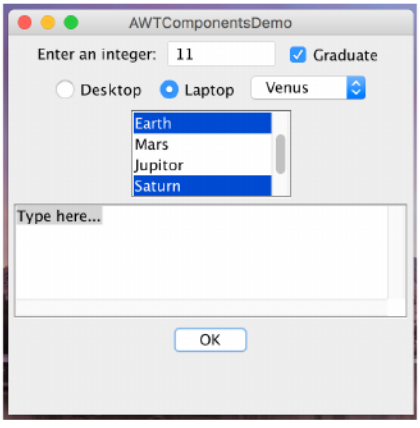
- Swing
  - package: javax.swing
  - A *lightweight* toolkit
  - Only uses the basic graphics primitives from the operating system libraries. rest of the functionality is developed using pure Java
  - Does not have native *peer* component. that is why it is called a lightweight toolkit
  - Because Swing is 100% Java (except basic graphics primitives), look and feel are same across platforms (they may be changed using pluggable look-and-feel)



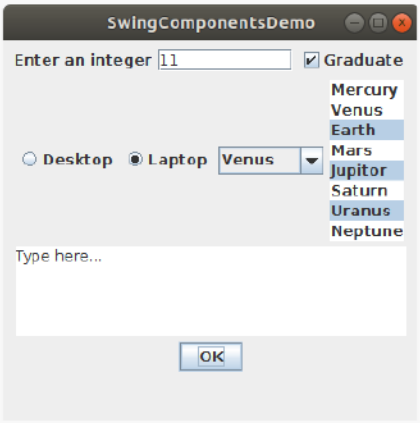
AWT Program under Linux



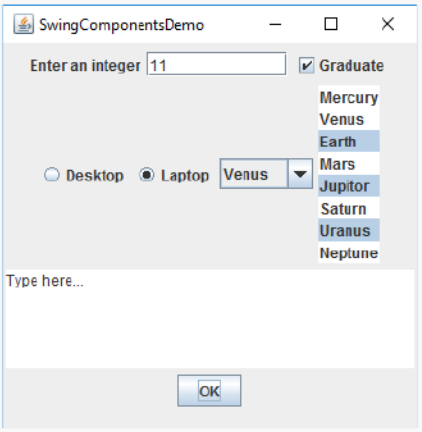
AWT Program under Windows



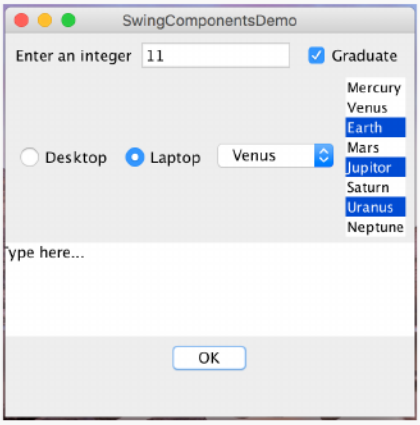
AWT Program under macOS



Swing Program under Linux



Swing Program under Windows



Swing Program under macOS

```
// AWTAccumulator.java program

import java.awt.*;
import java.awt.event.*;

public class AWTAccumulator extends Frame implements
        ActionListener
{
   private Label lblInput;
   private Label lblOutput;
   private TextField tfInput;
   private TextField tfOutput;
   private int sum = 0;

   public AWTAccumulator()
   {
      setLayout(new FlowLayout());

      lblInput = new Label("Enter an Integer: ");
      add(lblInput);

      tfInput = new TextField(10);
```

```
        add(tfInput);

        tfInput.addActionListener(this);

        lblOutput = new Label("The Accumulated Sum is: ");
        add(lblOutput);

        tfOutput = new TextField(10);
        tfOutput.setEditable(false);
        add(tfOutput);

        setTitle("AWT Accumulator");
        setSize(350, 120);
        setVisible(true);

        MyWindowAdapter myWindowAdapter =
                  new MyWindowAdapter();
        this.addWindowListener(myWindowAdapter);

        /*
        this.addWindowListener(
                new WindowAdapter()
                {
                    public void windowClosing(WindowEvent we)
                    {
                          dispose();
                    }
                }
            );
        */

    }

    class MyWindowAdapter extends WindowAdapter
    {
         @Override
         public void windowClosing(WindowEvent we)
         {
             System.out.println("windowClosing()...");
             dispose();
         }

    }

    public static void main(String[] args)
    {
        AWTAccumulator frame = new AWTAccumulator();
    }

    @Override
    public void actionPerformed(ActionEvent evt)
    {
```

```
      int numberIn = Integer.parseInt(tfInput.getText());
      sum += numberIn;
//      tfOutput.setText(sum + "");
      tfOutput.setText(String.valueOf(numberIn));
      tfInput.setText("");
   }

}
```

```
// AWTComponentsDemo.java program

import java.awt.*;
import java.awt.event.*;

public class AWTComponentsDemo extends Frame
{

    private Label labelTextField;
    private TextField textFieldInteger;
    private Checkbox checkboxGraduate;
    private Label labelRadioButtonGroup1;
    private CheckboxGroup radioButtonGroup1;
    private Checkbox radioButtonDesktop;
    private Checkbox radioButtonLaptop;
    private Label labelRadioButtonGroup2;
    private CheckboxGroup radioButtonGroup2;
    private Checkbox radioButtonTubelight;
    private Checkbox radioButtonCFL;
    private Checkbox radioButtonLEDBulb;
    private Choice choice;
    private List list;
    private TextArea textArea;
    private Button buttonOK;
    private Button buttonTest;

    public AWTComponentsDemo()
    {
        setLayout(new FlowLayout());

        labelTextField = new Label("Enter an integer: ");
        add(labelTextField);

        textFieldInteger = new TextField(10);

        textFieldInteger.addKeyListener(new KeyListener()
        {
            @Override
            public void keyTyped(KeyEvent e)
            {
```

```
            System.out.println("textFieldInteger keyTyped:
        KeyChar=" + e.
                    getKeyChar());
        }

        @Override
        public void keyPressed(KeyEvent e)
        {
            System.out.println("textFieldInteger keyPressed:
        KeyCode=" +
                    e.getKeyCode() + " KeyChar=" +
        e.getKeyChar());
        }

        @Override
        public void keyReleased(KeyEvent e)
        {
            System.out.println("textFieldInteger keyReleased:
        KeyCode=" +
                    e.getKeyCode() + " KeyChar=" +
        e.getKeyChar());
        }
    });

    add(textFieldInteger);

    checkboxGraduate = new Checkbox("Graduate", true);
    add(checkboxGraduate);

    labelRadioButtonGroup1 = new Label("System Type: ");
    add(labelRadioButtonGroup1);

    radioButtonGroup1 = new CheckboxGroup();

    radioButtonDesktop = new Checkbox("Desktop",
    radioButtonGroup1, true);
    radioButtonLaptop = new Checkbox("Laptop",
    radioButtonGroup1, false);

    add(radioButtonDesktop);
    add(radioButtonLaptop);

    labelRadioButtonGroup2 = new Label("Lighting Type: ");
    add(labelRadioButtonGroup2);

    radioButtonGroup2 = new CheckboxGroup();

    radioButtonTubelight = new Checkbox("Tubelight",
    radioButtonGroup2, true);
    radioButtonCFL = new Checkbox("CFL", radioButtonGroup2,
    false);
    radioButtonLEDBulb = new Checkbox("LED Bulb",
    radioButtonGroup2, false);
```

```
        add(radioButtonTubelight);
        add(radioButtonCFL);
        add(radioButtonLEDBulb);

        choice = new Choice();
        choice.add("Mercury");
        choice.add("Venus");
        choice.add("Earth");
        choice.add("Mars");
        choice.add("Jupitor");
        choice.add("Saturn");
        choice.add("Uranus");
        choice.add("Neptune");
//        choice.select(2);
        choice.select("Earth");
        add(choice);

        list = new List();
        list.add("Mercury");
        list.add("Venus");
        list.add("Earth");
        list.add("Mars");
        list.add("Jupitor");
        list.add("Saturn");
        list.add("Uranus");
        list.add("Neptune");
        Dimension preferredSize = new Dimension(550, 500);
        list.setPreferredSize(preferredSize);
        list.setMultipleMode(true);
        list.select(2);
        list.select(5);
        add(list);

        textArea = new TextArea("Type here...", 5, 40);
        textArea.selectAll();
        add(textArea);

        MyActionListener myActionListener = new
         MyActionListener();

        buttonOK = new Button("OK");
        buttonOK.setActionCommand("ActionCommandOK");
        buttonOK.addActionListener(myActionListener);
        add(buttonOK);

        buttonTest = new Button("Test");
        buttonTest.setActionCommand("ActionCommandTest");
        buttonTest.addActionListener(myActionListener);
        add(buttonTest);
```

```
        setTitle("AWTComponentsDemo");
        setSize(350, 350);
        setVisible(true);
        textArea.requestFocusInWindow();
        this.addWindowListener(
                new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                dispose();
            }
        }
        );
    }

    void displayContents(ActionEvent e)
    {
        System.out.println("buttonOK ActionListener
        ActionCommand=" + e.getActionCommand());
        String textFieldContents = textFieldInteger.getText();
        System.out.println("textFieldContents=" +
        textFieldContents);
        System.out.println("Checkbox Graduate=" +
        checkboxGraduate.getState());
        System.out.println("RadioButton Desktop:" +
        radioButtonDesktop.getState());
        System.out.println("RadioButton Laptop:" +
        radioButtonLaptop.getState());
        System.out.println("RadioButton Tubelight:" +
        radioButtonTubelight.getState());
        System.out.println("RadioButton CFL:" +
        radioButtonCFL.getState());
        System.out.println("RadioButton LED Bulb:" +
        radioButtonLEDBulb.getState());
        System.out.println("Choice Planet=" +
        choice.getSelectedItem());
        for (String selectedItem : list.getSelectedItems())
            System.out.println("Selected Planet in the list: " +
        selectedItem);
        System.out.println("TextArea contents=" +
        textArea.getText());
    }

    class MyActionListener implements ActionListener
    {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                String actionCommand = e.getActionCommand();
                System.out.println("Button pressed...
        actionCommand=" + actionCommand);
                switch (actionCommand)
                {
                    case "ActionCommandOK":
```

```
                    case "OK":
                        displayContents(e);
                        break;
                    case "ActionCommandTest":
                    case "Test":
                        System.out.println("Test button
        pressed");
                        break;
                    default:
                        System.out.println("Unknown button
        pressed");
                        break;
                }
            }
    }

    public static void main(String[] args)
    {
        AWTComponentsDemo frame = new AWTComponentsDemo();
    }

}
```

```
// SwingAccumulator.java program

import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Color;

/**
 *
 * @author class
 */
public class SwingAccumulator
{

    JFrame frame;
    JLabel jLabelInput;
    JTextField jTextFieldInput;
    JLabel jLabelAccumulator;
    JTextField jTextFieldAccumulator;
    int sum = 0;

    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
```

```
     */
    private void createAndShowGUI()
    {
        //Create and set up the window.

        frame = new JFrame("SwingAccumulator");
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jLabelInput = new JLabel("Enter an integer");
        frame.add(jLabelInput);

        jTextFieldInput = new JTextField(10);
        jTextFieldInput.addActionListener(new
        MyActionListener());
        frame.add(jTextFieldInput);

        jLabelAccumulator = new JLabel("The accumulated sum
        is:");
        frame.add(jLabelAccumulator);

        jTextFieldAccumulator = new JTextField(10);
        jTextFieldAccumulator.setEnabled(false);
//
        jTextFieldAccumulator.setDisabledTextColor(Color.DARK_GRAY),
        jTextFieldAccumulator.setDisabledTextColor(Color.RED);
        frame.add(jTextFieldAccumulator);

        //Display the window.
        frame.setSize(350, 350);
//        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                SwingAccumulator swingAccumulator = new
        SwingAccumulator();
                swingAccumulator.createAndShowGUI();
            }
        });
    }

    class MyActionListener implements ActionListener
    {
        @Override
        public void actionPerformed(ActionEvent evt)
        {
            int numberInput =
        Integer.parseInt(jTextFieldInput.getText());
```

```
            sum += numberInput;
            jTextFieldAccumulator.setText(sum + "");
            jTextFieldInput.setText("");
        }
    }

}
```

```
// SwingComponentsDemo.java program

import java.awt.Dimension;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JRadioButton;
import javax.swing.ButtonGroup;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.ListSelectionModel;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

/**
 *
 * @author class
 */
public class SwingComponentsDemo
{

    JFrame frame;
    JLabel jLabelJTextField;
    JTextField jTextField;
    JLabel jLabelJCheckBox;
    JCheckBox jCheckBox;
    JLabel jLabelJRadioButtonDesktop;
    JRadioButton jRadioButtonDesktop;
    JLabel jLabelJRadioButtonLaptop;
    JRadioButton jRadioButtonLaptop;
    JComboBox<String> jComboBox;
    JList<String> jList;
    JTextArea jTextArea;
    JButton jButton;
```

```
    JOptionPane jOptionPane;

    /**
     * Create the GUI and show it.  For thread safety,
     * this method should be invoked from the
     * event-dispatching thread.
     */
    private void createAndShowGUI()
    {
        //Create and set up the window.

        frame = new JFrame("SwingComponentsDemo");
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jLabelJTextField = new JLabel("Enter an integer");
        frame.add(jLabelJTextField);

        jTextField = new JTextField(10);
        jTextField.addKeyListener(new MyKeyAdapter());
        frame.add(jTextField);

//        jLabelJCheckBox = new JLabel("JCheckBox");
//        frame.add(jLabelJCheckBox);

        jCheckBox = new JCheckBox("Graduate", true);
        frame.add(jCheckBox);

        jLabelJRadioButtonDesktop = new JLabel("RadioButton
        Desktop");
//        frame.add(jLabelJRadioButtonDesktop);

        jRadioButtonDesktop = new JRadioButton("Desktop", true);
        frame.add(jRadioButtonDesktop);

//        jLabelJRadioButtonLaptop = new JLabel("RadioButton
        Laptop");
//        frame.add(jLabelJRadioButtonLaptop);

        jRadioButtonLaptop = new JRadioButton("Laptop", true);
        frame.add(jRadioButtonLaptop);

        ButtonGroup radioButtonGroup = new ButtonGroup();
        radioButtonGroup.add(jRadioButtonDesktop);
        radioButtonGroup.add(jRadioButtonLaptop);

        jComboBox = new JComboBox<>();
        jComboBox.addItem("Mercury");
        jComboBox.addItem("Venus");
        jComboBox.addItem("Earth");
        jComboBox.addItem("Mars");
        jComboBox.addItem("Jupitor");
```

```
        jComboBox.addItem("Saturn");
        jComboBox.addItem("Uranus");
        jComboBox.addItem("Neptune");
        jComboBox.setSelectedIndex(2);
        frame.add(jComboBox);

        String[] jListItems = {"Mercury", "Venus", "Earth",
        "Mars", "Jupitor", "Saturn", "Uranus", "Neptune",};
        jList = new JList<>(jListItems);
//      Default Value
//
        jList.setSelectionMode(ListSelectionModel.MULTIPLE_INTERVAL_
//
        jList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION),
        jList.setSelectedIndex(2);
        frame.add(jList);

        jTextArea = new JTextArea(5, 30);
        jTextArea.setText("Type here...");
        frame.add(jTextArea);
        jTextArea.selectAll();

        jButton = new JButton("OK");
        jButton.addActionListener(new ActionListener()
                {
                    @Override
                    public void actionPerformed(ActionEvent e)
                    {
//                      System.out.println("Waiting...");
//                      try
//                      {
//                          Thread.sleep(20000);
//                      }
//                      catch (InterruptedException ie)
//                      {
//                      }
                        System.out.println("----------------");
                        System.out.println("TextField:" +
        jTextField.getText());
                        System.out.println("CheckBox:" +
        jCheckBox.isSelected());
                        System.out.println("RadioButton Desktop:"
        + jRadioButtonDesktop.isSelected());
                        System.out.println("RadioButton Laptop:"
        + jRadioButtonLaptop.isSelected());
                        System.out.println("ComboBox:" +
        jComboBox.getSelectedItem());
                        System.out.print("List selected items
        (indices): ");
                        for (int selectedItem :
        jList.getSelectedIndices())
                        {
                            System.out.print(selectedItem + " ");
                        }
```

```
                        System.out.println();
                        System.out.print("List selected items
        (values): ");
                        for (String selectedItem :
        jList.getSelectedValuesList())
                        {
                            System.out.print(selectedItem + " ");
                        }
                        System.out.println();
                        System.out.println("TextArea:" +
        jTextArea.getText());
                    }
                });
        frame.add(jButton);

        //Display the window.
        frame.setSize(350, 350);
//        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
        //creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                SwingComponentsDemo swingComponentsDemo = new
        SwingComponentsDemo();
                swingComponentsDemo.createAndShowGUI();
            }
        });
    }

    class MyKeyAdapter extends KeyAdapter
    {
       @Override
       public void keyTyped(KeyEvent evt)
       {
           int keyCode = evt.getKeyCode();
           char keyChar = evt.getKeyChar();
           System.out.println("Key Typed: Char=" + keyChar);
       }
    }

}
```

- AWTAccumulator.java program
- AWTKeyAdapter.java program
- AWTKeyEventListener.java program

- AWTComponentsDemo.java program
- AWTGridLayout.java program
- AWTComponentsDemoGridLayout.java program
- SwingAccumulator.java program
- SwingComponentsDemo.java program
- SwingMenuDemo.java program
- SwingOptionPaneDemo.java program
- SwingScrollBarDemo.java program
- SwingDemo01 project
- JdbcSwingExample1 project

JDBC

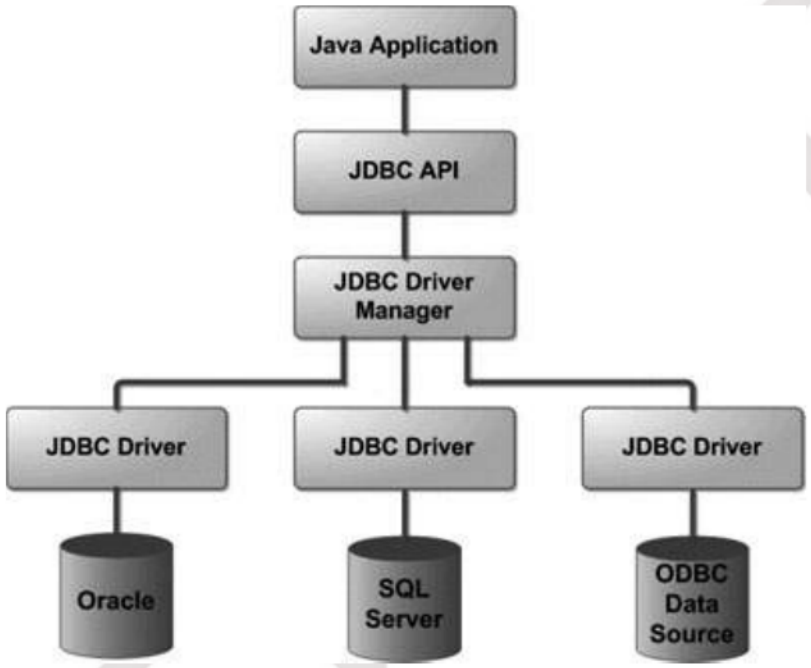


Diagram of JDBC architecture

Types of JDBC Drivers

- **Type 1** Type 1 JDBC driver calls native code of the locally available ODBC driver (i.e. JDBC-ODBC bridge). (Note: In JDBC 4.2, JDBC-ODBC bridge has been removed)
- **Type 2** Type 2 JDBC driver calls database vendor native library on a client

side. This code then talks to database over the network

- **Type 3** Type 3 JDBC driver is the pure-java driver that talks with the server-side middleware that then talks to the database
- **Type 4** Type 4 JDBC driver is the pure-java driver that uses database native protocol

```
// JdbcMysqlExample1.java program

// Please note:
//      To run this example from the command line, you must
        include the
//      MySQL JDBC driver library as well as the current
        directory in
//      the class path.
//      To run this example in NetBeans, you must include the
        MySQL
//      JDBC driver JAR file in the project's libraries.
//  Refer to the topic "Class Loaders and the Class Path" to
//      learn how to do this.

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.Scanner;

class DataAccessException extends Exception
{

    public DataAccessException(String msg)
    {
        super(msg);
    }
}

/**
 *
 * @author jvs
 */
public class JdbcMysqlExample1
{

    static String schema = "testemployees";
    static String userName = "root";
    static String password = "gdcst";
    static String url = "jdbc:mysql://localhost:3306/" +
                schema + "?autoReconnect=true&useSSL=false";
    static String driver = "com.mysql.cj.jdbc.Driver";
    static Connection connection = null;
```

```
static void insertEmployee(int empNo, String empName, String
    department, String designation, Double salary) throws
    DataAccessException
{
    String query;
    query = "insert into employees values(" + empNo
            + ", '" + empName + "', '" + department
            + "', '" + designation + "', " + salary + ")";
    System.out.println("query=" + query);
    try
    {
        Statement st = connection.createStatement();
        int rowCount = st.executeUpdate(query);
        if (rowCount != 1)
        {
            throw new DataAccessException("Insert failed.");
        }
        else
        {
            System.out.println();
            System.out.println(rowCount + " row(s)
    inserted.");
        }
    }
    catch (SQLException s)
    {
        s.printStackTrace();
        throw new DataAccessException("Insert failed.");
    }
}

static void updateEmployee(int empNo, String empName, String
    department, String designation, Double salary) throws
    DataAccessException
{
    String query;
    query = "update employees set "
            + "name='" + empName
            + "', department='" + department
            + "', designation='" + designation
            + "', salary=" + salary
            + " where empno=" + empNo;
    System.out.println("query=" + query);
    try
    {
        Statement st = connection.createStatement();
        int rowCount = st.executeUpdate(query);
        System.out.println();
        System.out.println(rowCount + " row(s) updated.");
    }
    catch (SQLException s)
```

```
        {
            s.printStackTrace();
            throw new DataAccessException("Update failed.");
        }
    }

    static void deleteEmployee(int empNo) throws
        DataAccessException
    {
        String query;
        query = "delete from employees where empno=" + empNo;
        System.out.println("query=" + query);
        try
        {
            Statement st = connection.createStatement();
            int rowCount = st.executeUpdate(query);
            System.out.println();
            System.out.println(rowCount + " row(s) deleted.");
        }
        catch (SQLException s)
        {
            s.printStackTrace();
            throw new DataAccessException("Deletion failed.");
        }
    }

    static void showTable() throws DataAccessException
    {
        String query;
        query = "select empno, name, department, designation,
        salary from employees order by empno";
        int count = 0;
        try
        {
            Statement st = connection.createStatement();
            ResultSet rs = st.executeQuery(query);
            System.out.println();
            System.out.println("Table Employees");
            System.out.println("----- ---------");
            while (rs.next())
            {
                int empno = rs.getInt(1);
                String name = rs.getString(2);
                String department = rs.getString(3);
                String designation = rs.getString(4);
                double salary = rs.getDouble(5);
                String msg = String.format(
                        "| %1$3d | %2$-10s | %3$-10s | %4$-10s |
        %5$-10s |",
                        empno, name, department, designation,
        salary);
                System.out.println(msg);
```

```
                count++;
            }
            System.out.println("**** " + count + " row(s) ****");
            System.out.println();
        }
        catch (SQLException s)
        {
            s.printStackTrace();
            throw new DataAccessException("select failed.");
        }
    }

    public static void main(String[] args) throws Exception
    {
        try
        {
            Class.forName(driver).newInstance();
        }
        catch (ClassNotFoundException | InstantiationException |
                IllegalAccessException ex)
        {
            ex.printStackTrace();
            throw ex;
        }
        try
        {
            System.out.println("JDBC URL: " + url);
            connection = DriverManager.getConnection(url,
        userName, password);
        }
        catch (SQLException se)
        {
            se.printStackTrace();
            throw se;
        }
        try
        {
            Scanner scanner = new Scanner(System.in);

            showTable();
            System.out.print("Press the ENTER key to continue:");
            scanner.nextLine();
            System.out.println();

            insertEmployee(501, "aaa", "Dept111", "Desig111",
        12345.67);
            showTable();
            System.out.print("Press the ENTER key to continue:");
            scanner.nextLine();
            System.out.println();
```

```
            updateEmployee(501, "zzz", "Deptzzz", "Desigzzz",
        99999.99);
            showTable();
            System.out.print("Press the ENTER key to continue:");
            scanner.nextLine();
            System.out.println();

            deleteEmployee(501);
            showTable();
            System.out.print("Press the ENTER key to continue:");
            scanner.nextLine();
            System.out.println();

        }
        catch (DataAccessException exc)
        {
            exc.printStackTrace();
        }
    }
}
```

- JdbcMysqlExample1.java program
- JdbcMysqlExample2.java program
- JdbcOracleExample1.java program
- JdbcSQLiteExample1.java program
- JavaMySQLJDBCConnection project
- JdbcSwingExample1 project

Generics

Sometimes, there is a need to create several classes that offer the same functionality or implement the same algorithm(s), but differ only in the data type(s) used. This requires us to create a generic meta-class or class template from which the classes can be created by specifying the data type(s) to be used. C++ provides a mechanism called *templates* for this. Since C++ supports classes as well as standalone functions, it has both function templates and class templates. In the following example, a function template for bubble sort is defined and then two functions are created from that template: one for the `int` data type and another for the `char` data type.

```cpp
// C++ code for bubble sort
// using template function
#include <iostream>
using namespace std;

// A template function to implement bubble sort.
// We can use this for any data type that supports
// comparison operator < and  whose values can be swapped by
        assignment.
template <class T>
void bubbleSort(T a[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = n - 1; i < j; j--)
            if (a[j] < a[j - 1])
            swap(a[j], a[j - 1]);
}

// Driver Code
int main() {
    int a[] = {10, 50, 30, 40, 20};
    int n1 = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort<int>(a, n1);

    cout << " Sorted array : ";
    for (int i = 0; i < n1; i++)
        cout << a[i] << " ";
    cout << endl;

    char b[] = {'Z', 'D', 'P', 'B', 'Q', 'A', 'M'};
    int n2 = sizeof(a) / sizeof(a[0]);

    // calls template function
    bubbleSort<char>(b, n2);

    cout << " Sorted array : ";
    for (int i = 0; i < n2; i++)
        cout << b[i] << " ";
    cout << endl;

return 0;
}
```

Output

```
Sorted array : 10 20 30 40 50
Sorted array : B D P Q Z
```

When we use C++ templates, usage of the template with each unique data type or data types results in the compiler creating a different function / class internally. In the above example, the compiled code has two functions, one for sorting `int` arrays and another for sorting `char` arrays.

Java added support for class templating in J2SE 5.0. In Java, the feature is called *generics*. Since Java does not support standalone functions, it only has generic classes and generic methods within generic classes but no standalone generic functions. Like C++ templates, Java generic classes can be used as blueprints to create multiple classes that differ only in the data type(s) used. The Java class library itself makes heavy use of generics. For example, the class `java.util.ArrayList` is a generic class that can be used to create dynamically sized lists of any data type. It has a generic method `iterator()` that returns an iterator of the specified type.

```
import java.util.ArrayList;
import java.util.Iterator;

 public class GenericsExample01
 {
     public static void main (String[] args)
     {
         // Create an ArrayList of Integer
         ArrayList<Integer> ArrayListInteger = new ArrayList<>();
         ArrayListInteger.add(10);
         ArrayListInteger.add(20);
         ArrayListInteger.add(30);
         // iterator() in ArrayList<Integer> returns Iteretor of
        Integer
         Iterator<Integer> iteratorInteger =
        ArrayListInteger.iterator();
         while(iteratorInteger.hasNext()) {
             System.out.println(iteratorInteger.next());
         }

         // Create an ArrayList of String
         ArrayList<String> ArrayListString = new ArrayList<>();
         ArrayListString.add("aaa");
         ArrayListString.add("bbb");
         ArrayListString.add("ccc");
         ArrayListString.add("ddd");
         // iterator() in ArrayList<String> returns Iteretor of
        String
         Iterator<String> iteratorString =
        ArrayListString.iterator();
         while(iteratorString.hasNext()) {
             System.out.println(iteratorString.next());
         }
```

```
    }

}
```

Output

```
10
20
30
aaa
bbb
ccc
ddd
```

Here, the type(s) specified between `< ... >` are known as type parameters. In the example above, we have used the generic class `java.util.ArrayList` with two different type parameters — Integer and String. In Java, **only reference types can be used as type parameters**. Primitive types are not allowed as type parameters, because type parameters must be classes. Unlike C++ templates, Java generics work by type erasure. At compile time, `ArrayList<Integer>` and `ArrayList<String>` behave like two different classes. The compiler enforces type safety. Thus, object of one class cannot be assigned to variable of the other type. However, during compilation, the type parameters are removed (erased) and there is only one class `java.util.ArrayList` in the compiled bytecode.

Generic classes can have more than one type parameters too.

```
import java.util.HashMap;

public class GenericsExample02
{
    public static void main (String[] args)
    {
        // Map Integer roll numbers to String names
        HashMap<Integer, String> rollNoNameMapping = new
         HashMap<>();
        rollNoNameMapping.put(1, "Qwe");
        rollNoNameMapping.put(2, "Asd");
        rollNoNameMapping.put(3, "Zxc");
        System.out.println(rollNoNameMapping.get(2));

        // Map String names to Integer ages
        HashMap<String, Integer> nameAgeMapping = new
         HashMap<>();
        nameAgeMapping.put("Brother", 19);
```

```
        nameAgeMapping.put("Aunt", 40);
        nameAgeMapping.put("Grandmother", 72);
        System.out.println(nameAgeMapping.get("Brother"));
        System.out.println(nameAgeMapping.get("Aunt"));

    }
}
/////// OUTPUT ///////
Asd
19
40
```

- ArrayListExample1 project
- ArrayListExample2 project
- ArrayListExample3 project

We can also define our own generic classes.

- GenericsExample01.java program
- GenericsExample02.java program
- GenericsCustomGenericClass1 project

Enumeration

An enumeration (`enum`) is a collection of related constants that can be accessed by their more user-firendly names. Unlike C / C++ enum values, Java enum values are printed in the user-friendly names form. By convention, enumeration constants have names in all uppercase. Value of one enum type cannot be assigned to another, ensuring type safety.

```
public enum Programme
{
    MCA, PGDCA, MSC_IT, MPHIL_CS, PHD_CS
}

public enum BookType
{
    BOOK, MAGAZINE, JOURNAL
}

public class EnumerationExample1
{

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

```
        Programme p;
        p = Programme.MCA;
        System.out.println("p1=" + p);
        p = Programme.MSC_IT;
        System.out.println("p2=" + p);
        BookType b;
        b = BookType.JOURNAL;
        System.out.println("b=" + b);
//        p = BookType.BOOK; //Compiler error
//        p = (Programme)BookType.BOOK; //Compiler error
        for (int i = 0; i < Programme.values().length; i++)
        {
            System.out.println("Programme=" +
        Programme.values()[i]);
        }
        for (BookType bt : BookType.values())
        {
            System.out.println("BookType=" + bt);
        }
    }
}
```

Output

```
p1=MCA
p2=MSC_IT
b=JOURNAL
Programme=MCA
Programme=PGDCA
Programme=MSC_IT
Programme=MPHIL_CS
Programme=PHD_CS
BookType=BOOK
BookType=MAGAZINE
BookType=JOURNAL
```

- EnumerationExample1 project
- EnumerationExample02.java program

Inner Classes

An inner class is a class defined inside another class (the outer class). It can only be used in the context of the outer class. Inner classes are heavily used by GUI and other frameworks.

```
public class InnerClassExample01
```

```
{

    // A class defined inside another class is called an inner
        class
    class MyInnerClass {
        int instanceNumber;

        public MyInnerClass(int instanceNumber) {
            this.instanceNumber = instanceNumber;
        }

        @Override
        public String toString() {
            return "MyInnerClass[" + instanceNumber + "]";
        }

    }

    public void demonstrate() {
        MyInnerClass myInnerClassObject1 = new MyInnerClass(1);
        System.out.println(myInnerClassObject1);
        MyInnerClass myInnerClassObject2 = new MyInnerClass(2);
        System.out.println(myInnerClassObject2);
    }

    public static void main (String[] args)
    {
        InnerClassExample01 innerClassExample01 = new
        InnerClassExample01();
        innerClassExample01.demonstrate();
    }

}
/////// OUTPUT ///////
MyInnerClass[1]
MyInnerClass[2]
```

- InnerClassExample01.java program

Anonymous Classes

Since Java is a purely object-oriented programming language, it does not support standalone functions (functions that are not part of any class). However, there are many situations where we just need to define and use a single function. This situation often arises in event-drive programming like GUI development, web-based software development, asynchronous programming, etc. It is also common where we need to implement an interface having a single

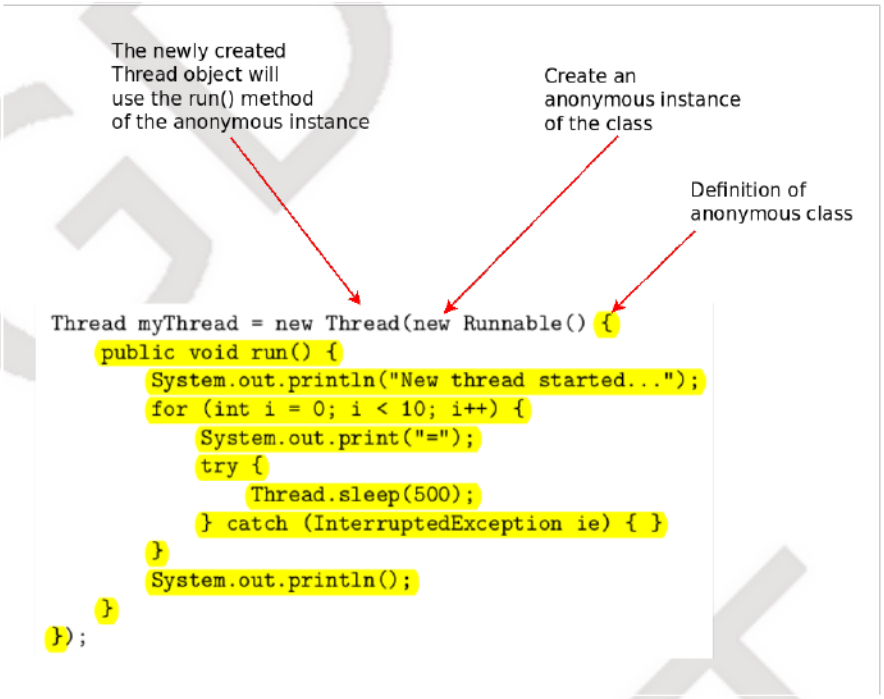
method (e.g. the `java.lang.Runnable` interface). In such cases, usually we have to define a class, then create a single instance of the class and use the only method in it. For example, in the program given below, we only need the `run()` method and that too only in one place in our code - where we create a new `Runnable` and then a new `Thread` based on it. Still, we are forced to define a class, create an instance of it and then use it for creating a new thread that would execute the `run()` method.

```
public class AnonymousClassExample01
{
    public static void main (String[] args)
    {
        MyTask myTask = new MyTask();
        Thread myThread = new Thread(myTask);
        myThread.start();
        for (int i = 0; i < 10; i++) {
            System.out.print("@");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) { }
        }
        System.out.println();
    }
}

class MyTask implements Runnable {

    @Override
    public void run() {
        System.out.println("New thread started...");
        for (int i = 0; i < 10; i++) {
            System.out.print("=");
            try {
                Thread.sleep(500);
            } catch (InterruptedException ie) { }
        }
        System.out.println();
    }

}
/////// OUTPUT ///////
@New thread started...
==@==@==@==@==@
@@@@
```

Since this situation is very common, Java provides a shortcut method (syntactic sugar) for it – right at the point where we need the method, we can define an

anonymous class (anonymous means not having a name), create its one and only anonymous instance using the `new` operator and then use the method of that instance. An anonymous class does not have a name, hence it can only be used at the point where it is defined. It cannot be used anywhere else in our code. In the same way, the instance we create is anonymous – it is not assigned to a variable – and hence cannot be used anywhere else. We define an anonymous class and create its anonymous instance using the syntax `new Constructor() { ... implementation code for the class ... }`. Using this feature, the above code can be shortened to -

```
public class AnonymousClassExample02
{
    public static void main (String[] args)
    {
        Thread myThread = new Thread(new Runnable() {
            public void run() {
                System.out.println("New thread started...");
                for (int i = 0; i < 10; i++) {
                    System.out.print("=");
                    try {
                        Thread.sleep(500);
                    } catch (InterruptedException ie) { }
                }
                System.out.println();
            }
        });
        myThread.start();
        for (int i = 0; i < 10; i++) {
            System.out.print("@");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) { }
        }
        System.out.println();
    }
}
/////// OUTPUT ///////
@New thread started...
==@==@==@==@==@
@@@@
```



Using an Anonymous Class

You can also find the use of anonymous classes in GUI programming examples (AWT / Swing) for event handling.

Class Loaders and the Class Path

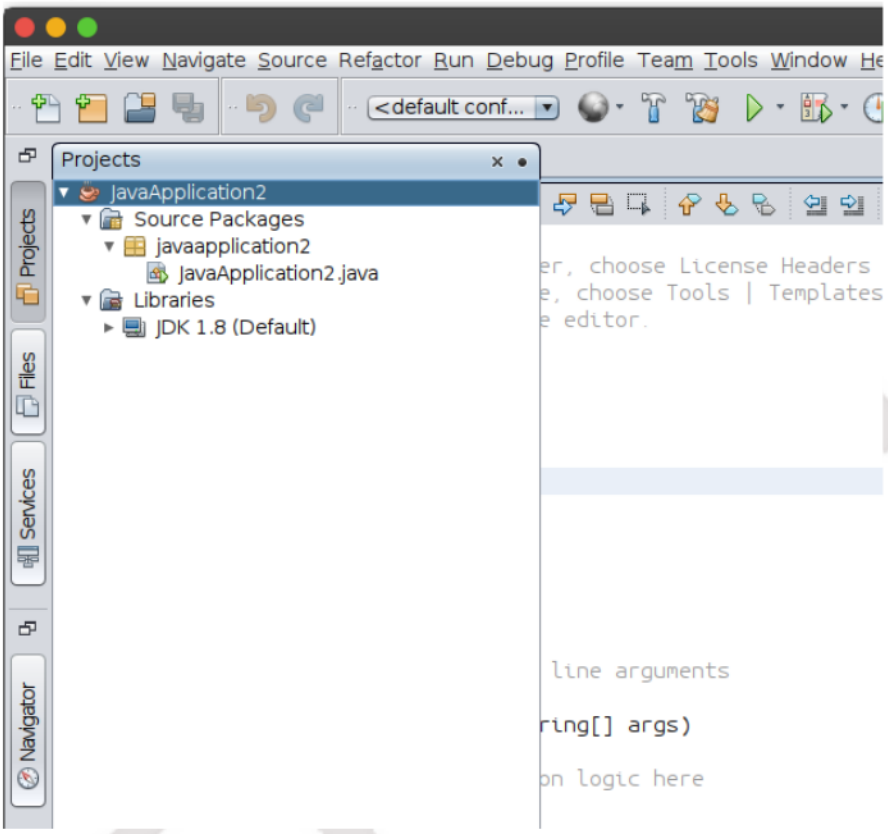
In our Java programs, we may have used several classes from the Java SE standard class library. These classes are not included in the programs during compilation. Also, there is no distinct *linking* phase in the Java development cycle, so the classes are not statically linked to our programs. Rather, Java uses dynamic linking – a class is loaded into the JVM when a program is running and it uses that class for the first time. A component of the JRE called *class loader* is used to load the required class into the JVM at the time of its first use. The class loader needs to know the location of the class so it can load it into the JVM. For the classes that are part of the standard library (and hence part of the JRE itself), it already knows from where to load them. However, when we use third-party libraries or custom class libraries developed by us as part of the same or different project, the class loader needs to know the location where it can find those third-party or custom classes. It uses the *class path* for that purpose. A class path is a variable (like the PATH variable used by operating systems) that holds a list of directories or library files from where classes need to be loaded. The `:` (colon) character is used as the separator in the class path. When working on the command line, the class path can be defined using the command line option `-cp` or `-classpath` as shown below. For example, to

access the MySQL database, we need to use the MySQL JDBC driver class library (say mysql-connector-java-8.0.19.jar). We can use a command as follows to run our program:

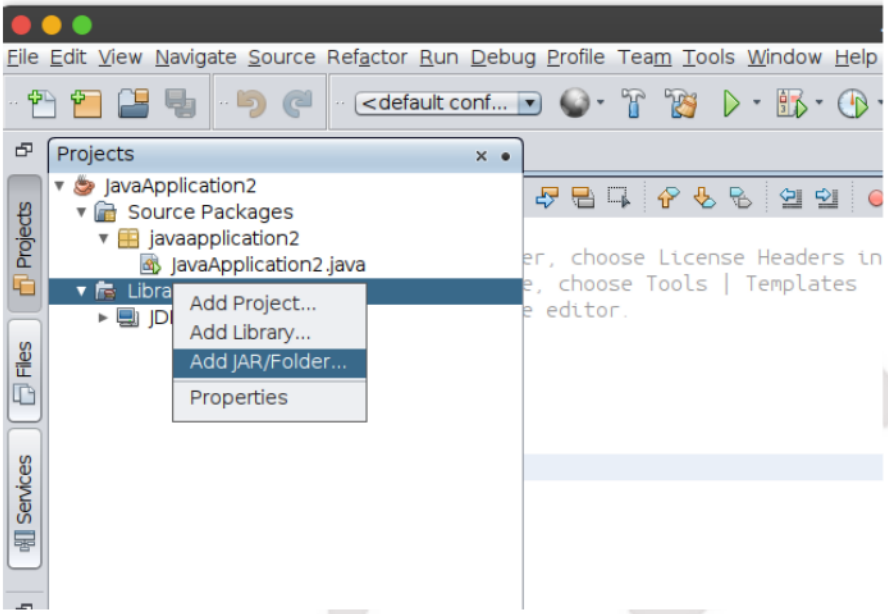
```
java -cp mysql-connector-java-8.0.19.jar:. JdbcMysqlExample1
```

Here, `mysql-connector-java-8.0.19.jar` is the MySQL JDBC driver class library and `.` indicates the current directory from where the class `JdbcMysqlExample1` will be loaded.

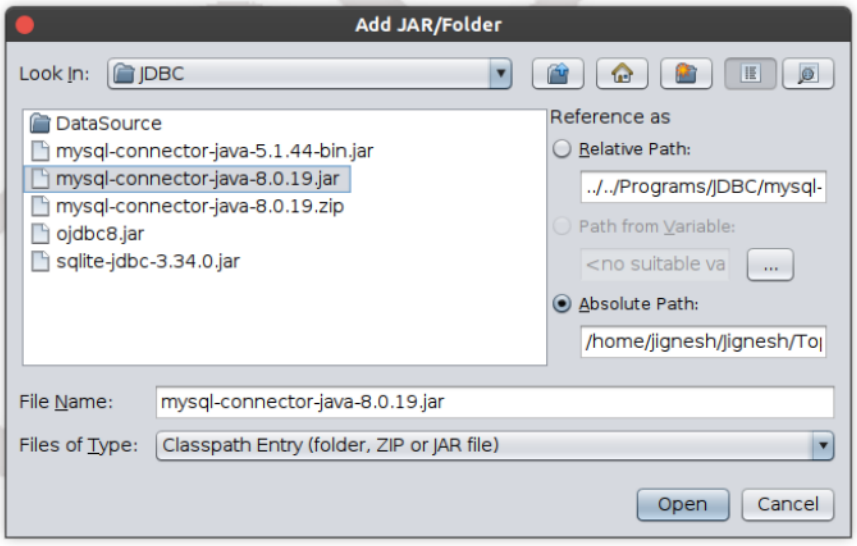
When working in a NetBeans project using the Apache ANT build tool, we can add third-party as well as our own custom libraries to the project by right-clicking on the *library* option and selecting the *Add JAR/Folder...* option as shown in the following screenshots. This will automatically add the libraries to the class path at runtime.



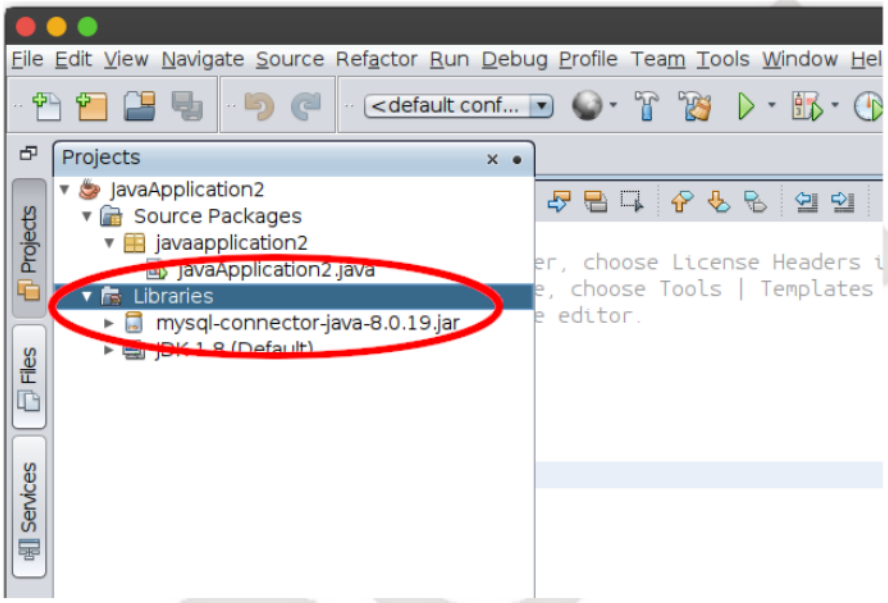
Adding a library to a NetBeans project-1



Adding a library to a NetBeans project-2



Adding a library to a NetBeans project-3



Adding a library to a NetBeans project-4

The Java Collections Framework

(From the Java™ Tutorials)

What Is the Collections Framework?

The Collections Framework is a unified architecture for representing and manipulating collections. The Collections Frameworks contains the following:

- **Interfaces** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations** These are the concrete implementations of the collection interfaces in the form of classes. In essence, they are reusable data structures.
- **Algorithms** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Benefits of the Java Collections Framework

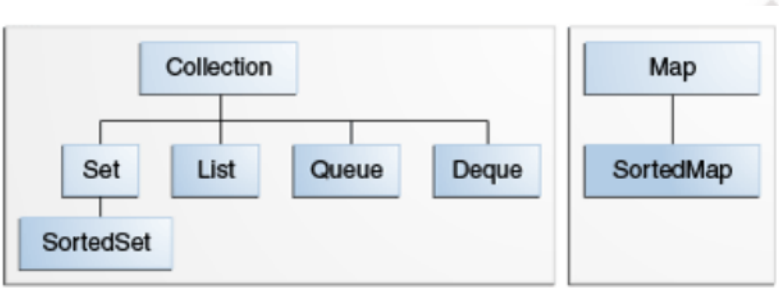
The Java Collections Framework provides the following benefits:

- **Reduces programming effort** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs
- **Increases program speed and quality** The Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. The programmer is freed from the drudgery of writing one's own data structures
- **Allows interoperability among unrelated APIs** The collection interfaces are the common types by which APIs pass collections back and forth. If a

network administration API furnishes a collection of node names and if a GUI toolkit expects a collection of column headings, the APIs will interoperate seamlessly, even though they were written independently

- **Reduces effort to learn and to use new APIs** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- **Reduces effort to design new APIs** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- **Fosters software reuse** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

The Collection Framework Interfaces



The Core Interfaces in the Collections framework

- **Collection** The root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List

- **Set** A collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. Sets cannot

have duplicate members and there is no ordering among members of the set. A `HashSet` is an implementation of Set

- **List** An ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). `Vector`, `LinkedList` and `Stack` are concrete implementations of a List

- **Queue** A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. `LinkedList`, `ArrayDeque` and `PriorityQueue` are concrete implementations of Queue

- **Deque** A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Deque provides additional insertion, extraction, and inspection operations. Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see The Deque Interface section. `LinkedList`, `ArrayDeque` and `PriorityQueue` are concrete implementations of Queue

- **Map** An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. Keys in the Map do not have any particular ordering. `HashMap` and `Hashtable` is a concrete implementation of a Map

- **SortedSet** A Set that maintains its elements in ascending order. It cannot have duplicate members. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls

- **SortedMap** A Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered

collections of key/value pairs, such as dictionaries and telephone directories

- ListExample1 project
- VectorExample1 project
- StackExample1 project
- HashMapExample1 project
- HashSetExample1 project

Serialization and Deserialization

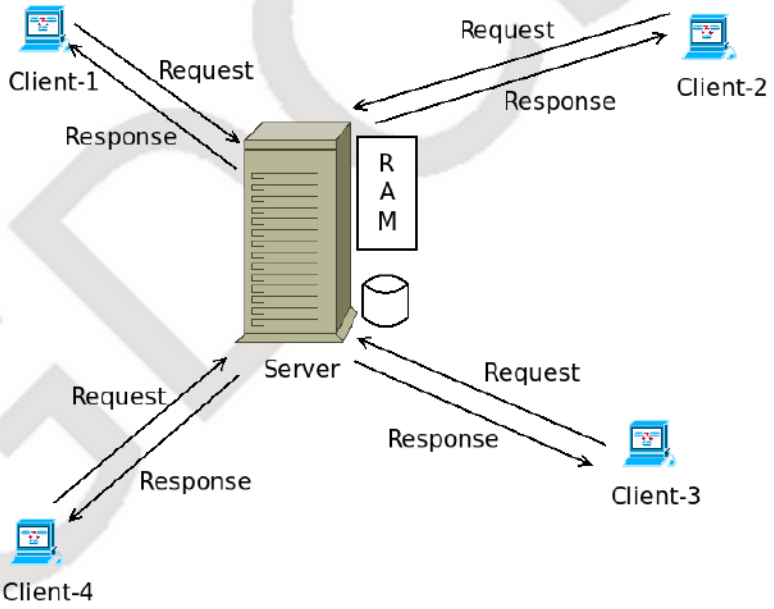
When we run a Java program and store values in objects, those objects are stored in JVM memory. As soon as the JVM shuts down, all the values are lost. If there is a requirement to save some or all of the objects, we may use the serialization mechanism to create a binary representation of the objects. This binary representation can be saved to a file. It can be read from the file at a later stage and the objects can be restored through the mechanism of deserialization. This technique is often used by Java-based software and frameworks for storing values of objects representing configuration to a binary configuration file so that those configuration objects can be recreated when the software is started again. The mechanism of serialization and deserialization is also useful in a distributed system when one needs to send objects from one node in the network to the other. The objects can be serialized on the first node and the binary representation of them can be sent to the other node. The objects can be recreated on the other node using deserialization.

To enable objects of a class to be serialized (and deserialized), the class must implement the `java.io.serializable` interface. This interface is just a marker interface and does not require any methods to be implemented. However, it is strongly recommended that every class implementing of this interface have a member `static final long serialVersionUID`. It can have any value.

- SerializeAndSendToServer.java program
- ReceiveFromClientAndDeserialize.java program
- Serialization project
- Deserialization project

Multithreading

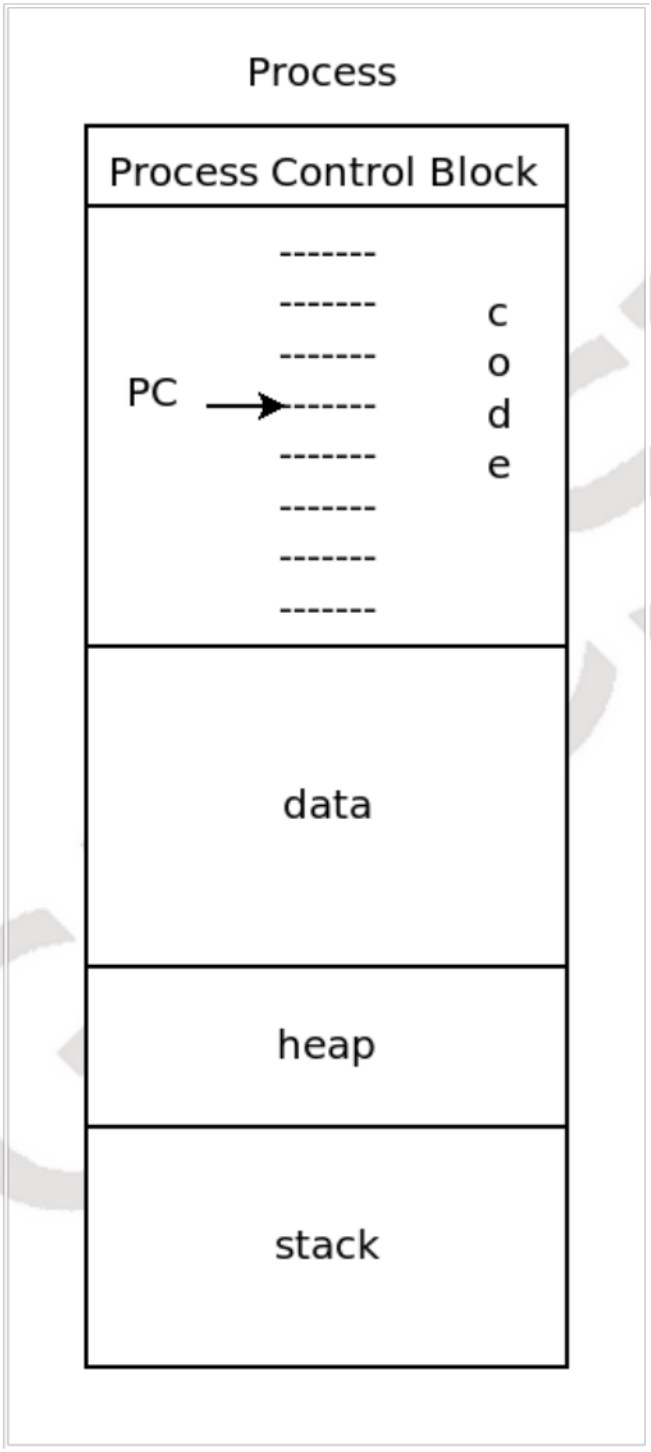
In a client-server architecture, one server services requests from multiple clients.



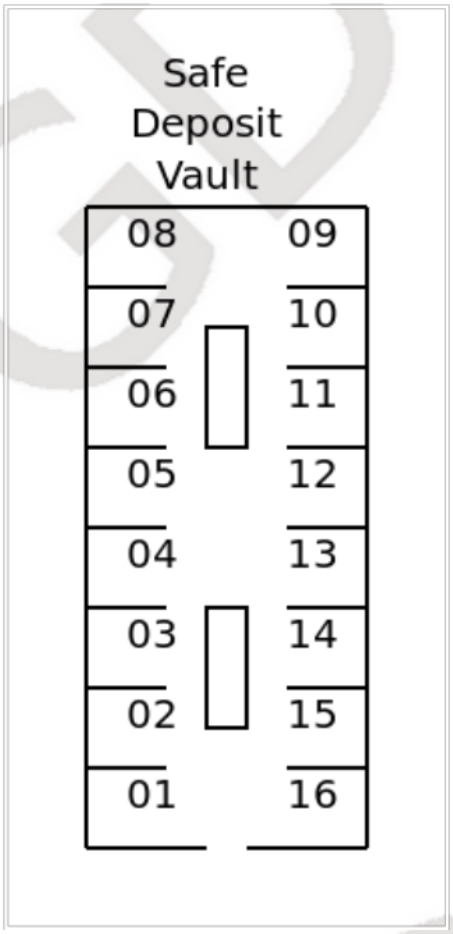
Client-Server-Environment

Single Process Solution

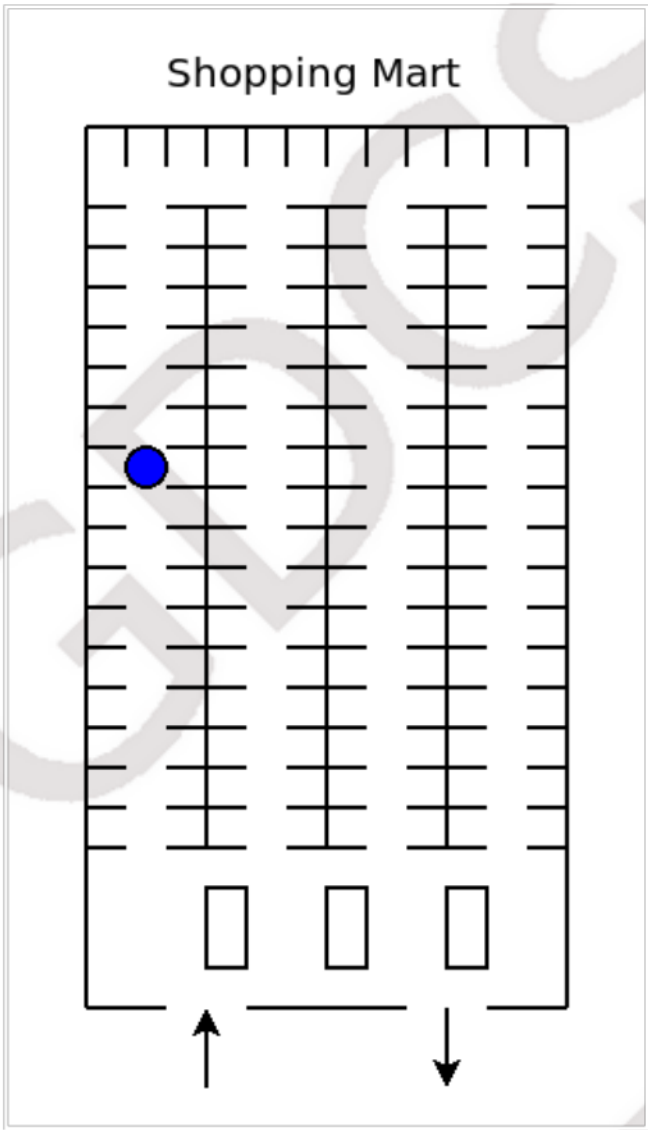
If there is only one server process, and that process is currently in waiting state due to I/O operation or for any other reason, it cannot serve other clients. Those other clients will have to wait even if some of their requests are such that they can be served immediately.



Single Process



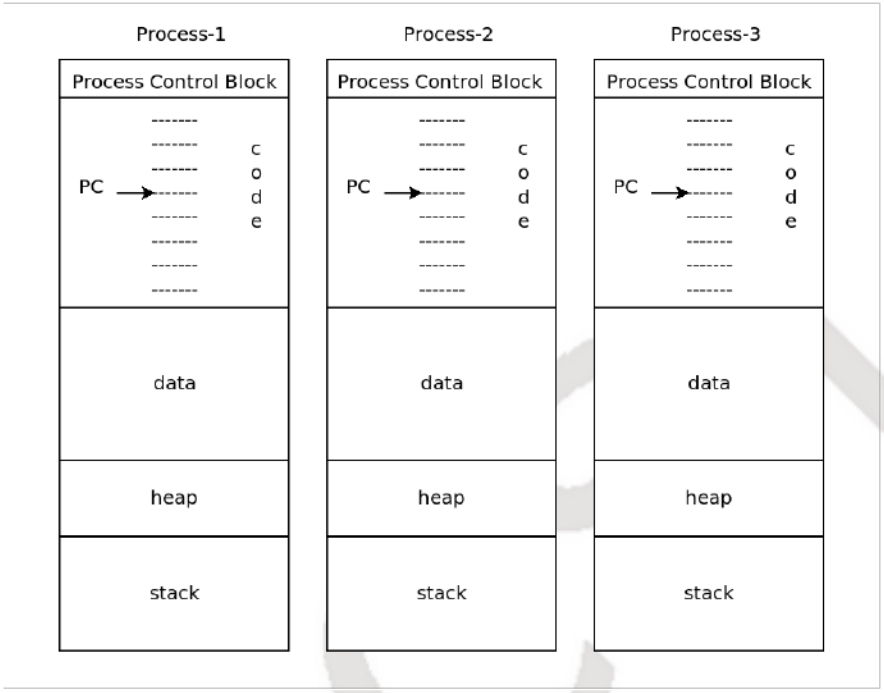
Single Safe Deposit Vault



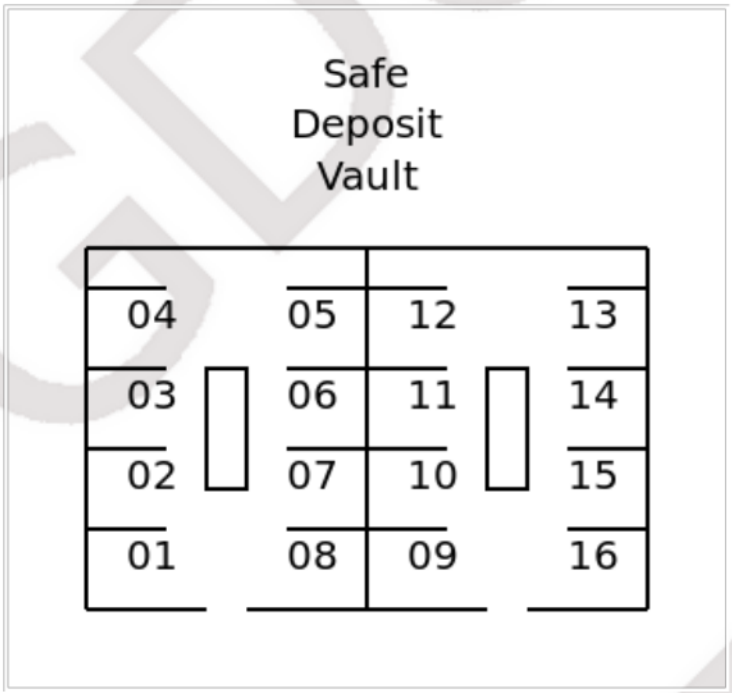
Shopping Mart - Single Customer Entry

Multi-process Solution

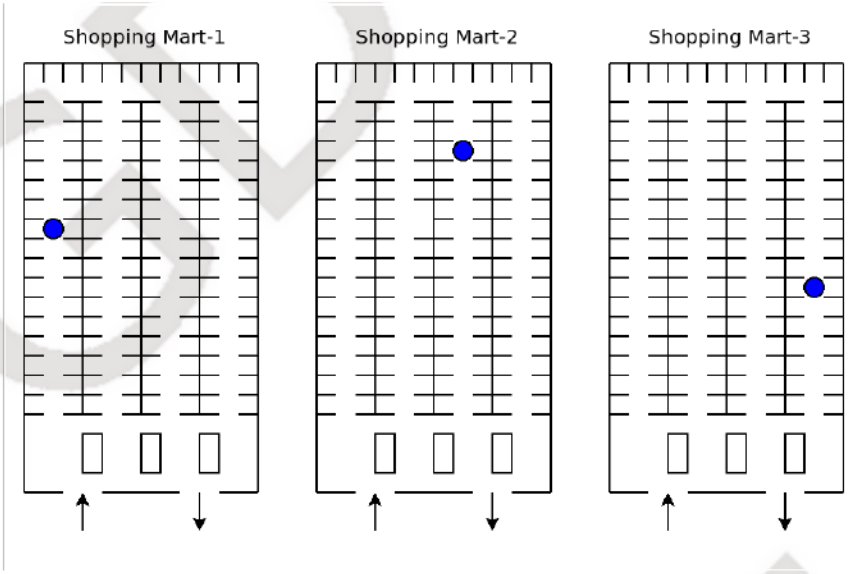
To be able to serve multiple clients, a multi-process solution may be adopted. Multiple processes can be created running the same server program. Even if some of those processes are in waiting state, other processes can continue to serve client requests. However, this solution is costly as major the parts of the process may be duplicated. Hence memory will be wasted. Also, switching between processes is slower because of the overhead of context switch.



Multi-process Solution



Multiple Safe Deposit Vaults

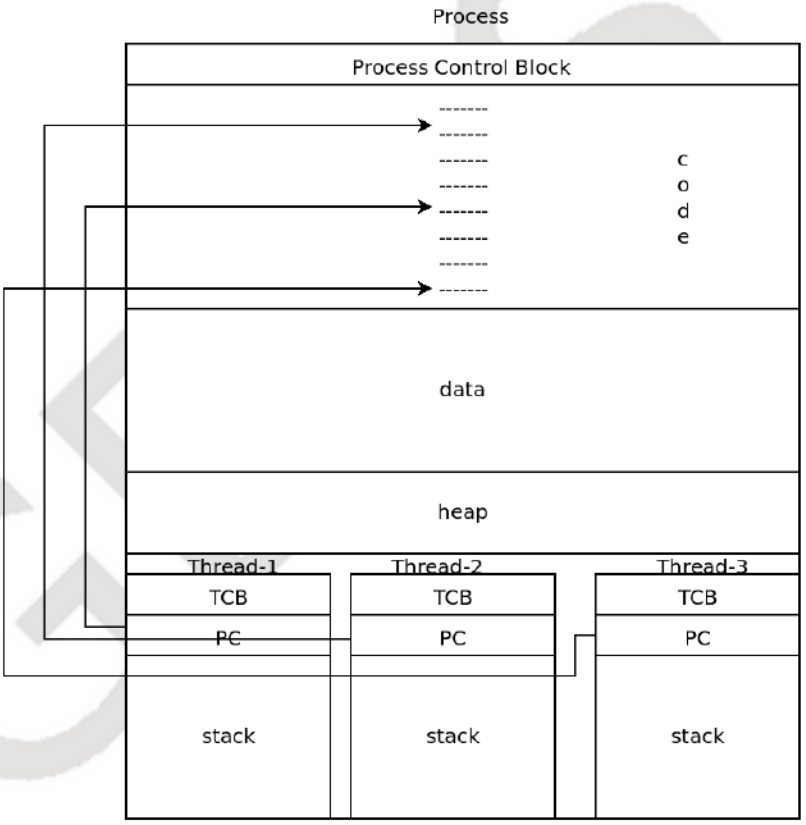


Multiple Shopping Marts

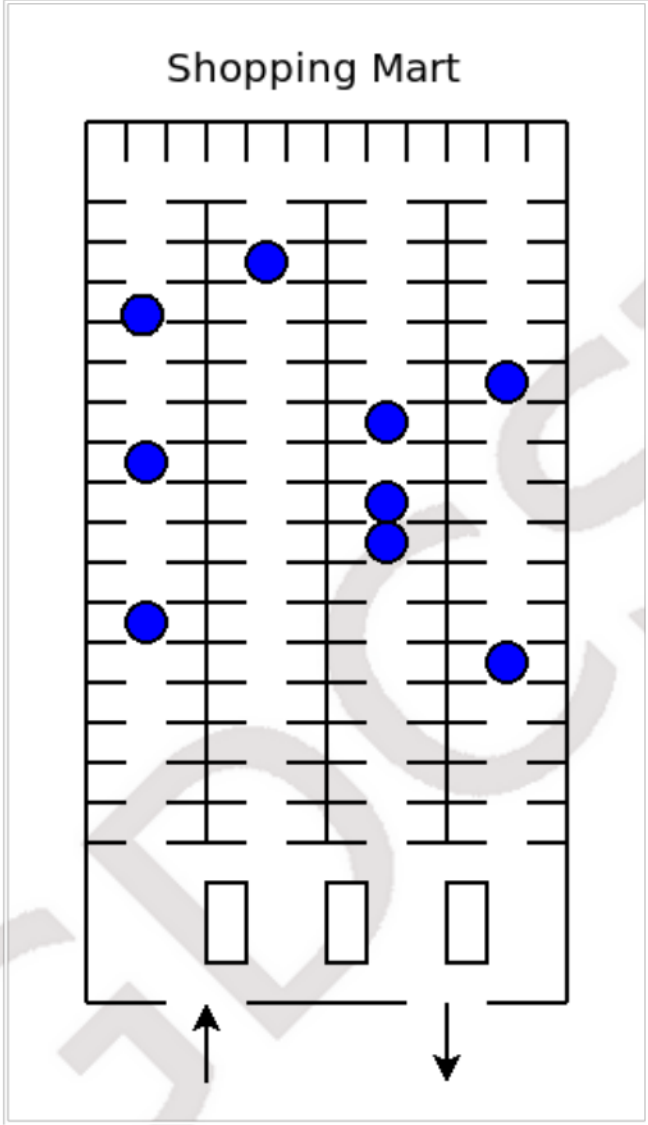
Multithreading Solution

In the multithreading solution, There is only one process, but the code of the process can be executed multiple times. Each execution is known as a thread. Initially, there is one thread. This thread may create more threads at different times, as and when the need arise. The threads may run at different speeds and may terminate at different times. The code is like a room in which different people enter and exit at different times. The process itself will terminate when all the threads terminate.

All the threads share the code, data and heap. Hence, duplication and wastage of memory is minimized. Each thread has a Thread Control Block (TCB), a stack and a PC (Program Counter) register value indicating the next instruction to be executed. Since threads share the memory, switching between threads is much faster than switching between processes. However, if multiple threads try to use the same value in memory at the same time, it may result in inconsistency. Care must be taken to write *thread-safe* code for such situations.



Multithreading



Single Shopping Mart - Multiple Customers Entry

Multithreading in Java

Java supports multithreading from the very first version. There are two ways of creating a new thread in Java - subsclassing the `java.lang.Thread` class and implementing the `java.lang.Runnable` interface.

```
public class MultiThreadingExample03
{
    public static void main (String[] args)
    {
        System.out.println("Main thread started...");
        char[] displayCharacters = { '@', '#', '$' };
        for (int i = 0; i < displayCharacters.length; i++)
        {
            MyTask myTask = new MyTask(displayCharacters[i],
        i+1);
            myTask.start();
        }
```

```
        System.out.println("Main thread terminated...");
    }
}

class MyTask extends Thread
{
    private char displayChar;
    private int delayPeriod;

    MyTask(char displayChar, int delayPeriod)
    {
        this.displayChar = displayChar;
        this.delayPeriod = delayPeriod;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.print(displayChar + String.valueOf(i) + "
        ");
            try
            {
                Thread.sleep(delayPeriod * 1000);
            }
            catch (InterruptedException ex)
            {
            }
        }
        System.out.println();
    }

}
```

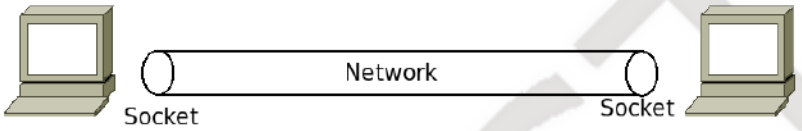
```
public class MultiThreadingExample02
{
    public static void main (String[] args)
    {
        System.out.println("Main thread started...");
        char[] displayCharacters = { '@', '#', '$' };
        for (int i = 0; i < displayCharacters.length; i++)
        {
            MyTask myTask = new MyTask(displayCharacters[i],
        i+1);
            Thread thread = new Thread(myTask);
            thread.start();
        }
        System.out.println("Main thread terminated...");
    }
}
```

```
class MyTask implements Runnable
{
    private char displayChar;
    private int delayPeriod;

    MyTask(char displayChar, int delayPeriod)
    {
        this.displayChar = displayChar;
        this.delayPeriod = delayPeriod;
    }

    @Override
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            System.out.print(displayChar + String.valueOf(i) + "
        ");
            try
            {
                Thread.sleep(delayPeriod * 1000);
            }
            catch (InterruptedException ex)
            {
            }
        }
        System.out.println();
    }

}
```

- MultiThreadingExample01.java program
- MultiThreadingExample02.java program
- MultiThreadingExample03.java program
- CreatingThreadsUsingRunnable project
- creatingThreadsUsingThreadSubclass project
- ThreadExample1 project
- ThreadExample2 project
- ThreadExample3 project
- GUILongRunningOperationExample1 project

- GUILongRunningOperationExample2 project

Network Programming



Network Sockets

A computer network connects two or more computing devices and allows them to communicate. There are several modes of communication among devices, like one-to-one, multicast, broadcast, etc. One-to-one communication mode is by far the most common. In this mode, one computing device connects with another computing device and then they exchange data over the network connection. The network acts like a conduit or pipe between the two devices. In the TCP protocol, the two ends of the pipes are known as *sockets*. The IP address and the TCP port number are two important parameters when working with sockets.

There are two types of sockets. A server computer uses a server socket. A server socket remains in listening mode, waiting for requests from client machines. As and when a request arrives from a client machine, it is processed. In Java, the `java.net.ServerSocket` class implements server sockets. A client socket is used by the client computers to send requests to a server computer. In Java, the `java.net.Socket` class implements a client socket.

- SocketSingleClientServerExample1 project
- SocketClientExample1 project
- SocketMultipleClientServerExample1 project

Functional Programming and Lambda Expressions

Like structured programming and object-oriented programming, there is a programming paradigm called functional programming. Its significance is increasing due to the prevalence of multicore CPUs. Functional programming and lambda expressions make it easier to implement parallelism in our code.

Lambda expression is a construct in the functional programming paradigm. It defines an anonymous function (a function without a name). It uses the `->` (arrow) operator.

For example,

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

is equivalent to the pseudo-code

```
list.forEach(
        function display(String str) {
            System.out.println(str);
        }
        );
```
