Object Oriented Programming Methodology

Module 1 : Fundamentals of Object Oriented Programming

Faculty In-charge:
Pragya Gupta
E-mail: pragya.g@somaiya.edu

Contents

- Procedural Programming Approach
- Structured Programming Approach
- Modular Programming Approach
- Object Oriented Programming Approach
- Objects and Classes
- OOP Features
- Static and Dynamic Binding
- Cohesion and Coupling

Objective

- Understand what is Object Oriented programming
- Understand the principles of OOP
- How is OOP different from Procedural languages
- Problems in Procedural programming and how OOP overcomes them
- Know Java features and its Runtime Environment
- Understand basic structure of a Java program
- Know about the various constituents of JDK and its development environments

Procedural Programming Approach

- Specifying the **steps** the program must take to reach the desired **state**
- Based upon the concept of the **procedure call**
- Procedures:
  - Routines, subroutines, methods, or functions that contain a series of computational steps to be carried out
- Any given procedure might be called at any point during a program's execution, including by other procedures or itself
- Using a procedural language, the programmer specifies **language statements** to perform a **sequence of algorithmic steps**

Procedural Programming Approach

- Complexity increases as the length of a program increases .
- Divide a large program into different functions or modules
- Problems with Procedural languages
  - functions have unrestricted access to global data
  - that they provide poor mapping to real world
  - Procedural languages are not extensible
- Example: Fortran, C, Pascal

Structured Programming Approach

- A technique for organizing and coding programs in which hierarchy of modules is used
- Each module has a single entry and single exit point
- Control is passed downward through the structure without unconditional branches to higher levels
- Three types of control flow: Sequential, Test or Selection, Iteration

Modular Programming Approach

- A technique for separating functionality into independent, interchangeable modules
- Each module contains everything necessary to execute only one aspect
- Sub-programs or functions

Introduction to OOP

- A programming paradigm
- deals with concepts of object to build programs and software applications
- Modeled around real world
- The world we live in is full of objects
- Every object has a well defined identity, attributes and behavior
- Objects exhibit the same behavior in programming.

OOP Features

- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OOP Approach

- Uses "objects"
  - Data structures encapsulating data fields and procedures together with their interactions – to design applications and computer programs.
- Object-oriented programming (OOP) involves programming using objects
- An *object* represents an entity in the real world that can be distinctly identified
  - For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects
- OOP features : **Data abstraction, encapsulation, polymorphism, and inheritance**

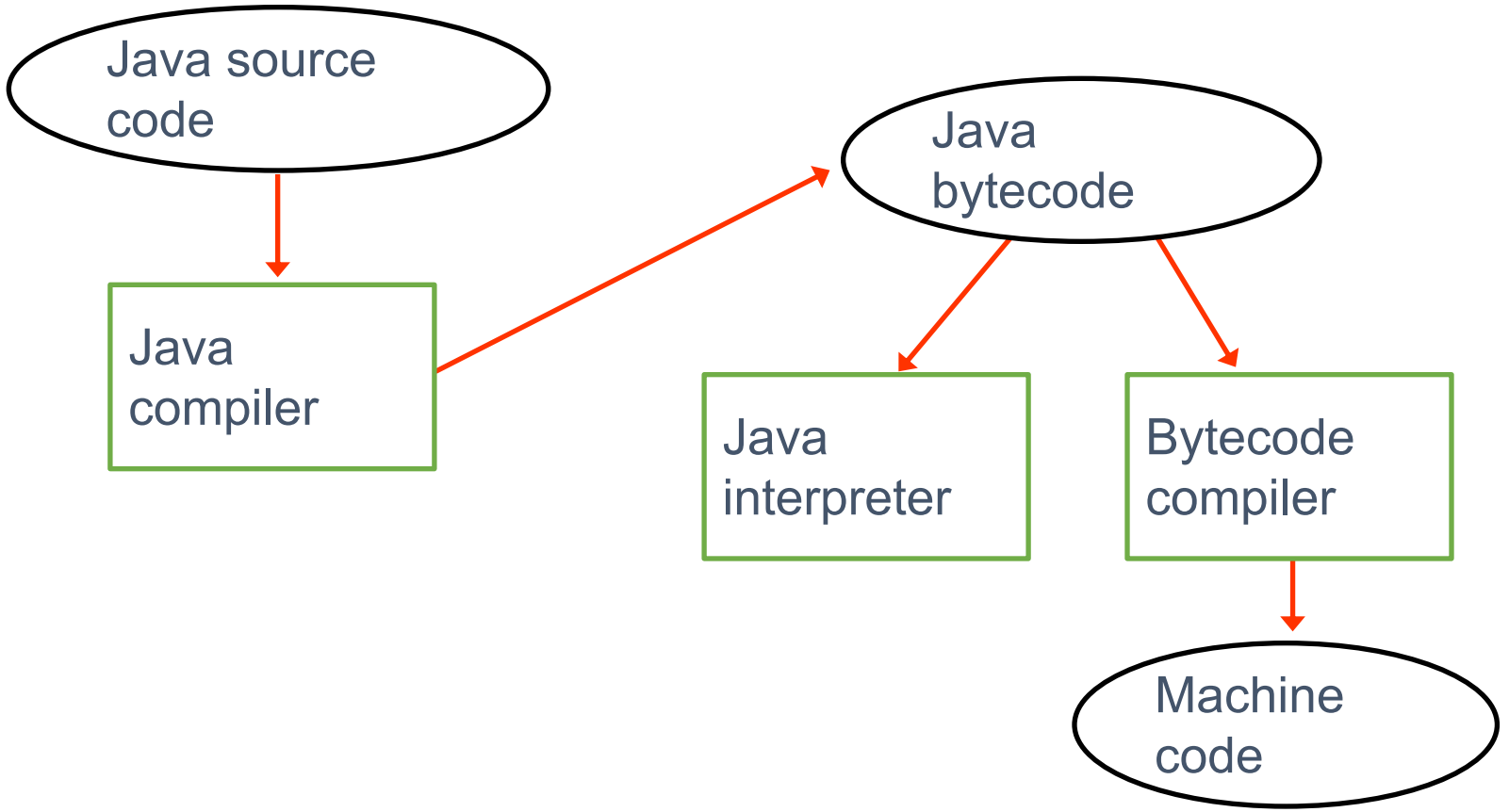
Comparison of OO and Procedural Languages

Procedural language	Object Oriented language
Separate data from function that operate on them	Encapsulate data and methods in a class
Not suitable for defining abstract types	Suitable for defining abstract types
Debugging is difficult	Debugging is easier
Difficult to implement change	Easier to manage and implement change
Not suitable for larger applications/programs	Suitable for larger programs and applications
Analysis and design not so easy	Analysis and Design Made Easier
Faster	Slower
Less flexible	Highly flexible
Data and procedure based	Object oriented
Less reusable	More reusable
Only data and procedures are there	Inheritance, encapsulation and polymorphism are key features
Use top down approach	Use bottom up approach
Only a function call to another	Object communication is there
C, Basic, FORTRAN	JAVA,C++, VB.NET, C#.NET

Java Essentials

- A high level language
- Java Bytecode – intermediate code
- Java Virtual Machine (JVM) – interpreter for bytecode

Java Translation

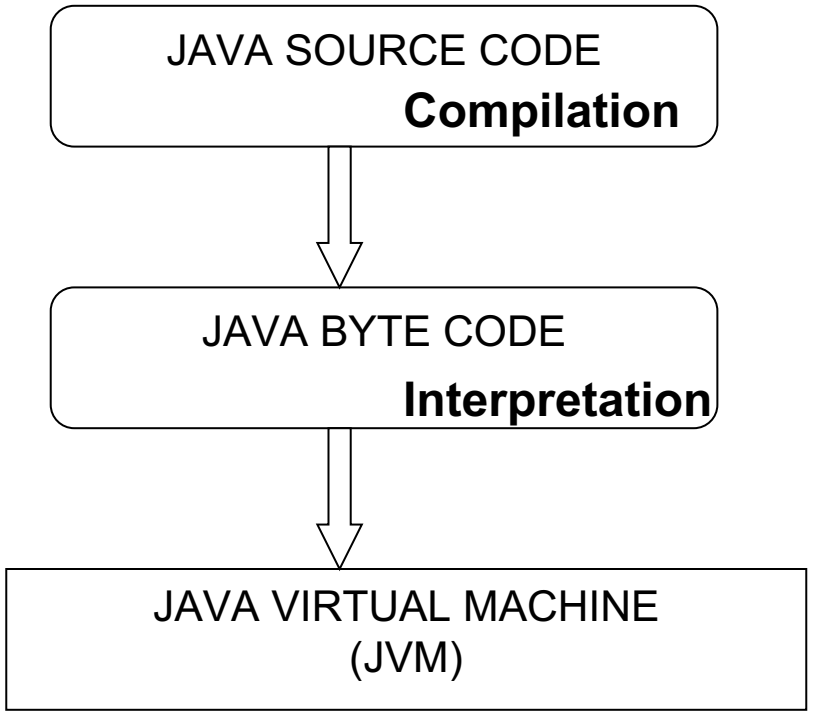


Java Translation

- The Java compiler translates Java source code into a special representation called ***bytecode***
- Java bytecode is not the machine language for any traditional CPU
- Another software tool, called an interpreter translates bytecode into machine language and executes it
- Therefore the Java compiler is not tied to any particular machine
- Java is considered to be architecture-neutral

Java Runtime

Java Runtime Environment includes JVM, class libraries and other supporting files



Java Approach

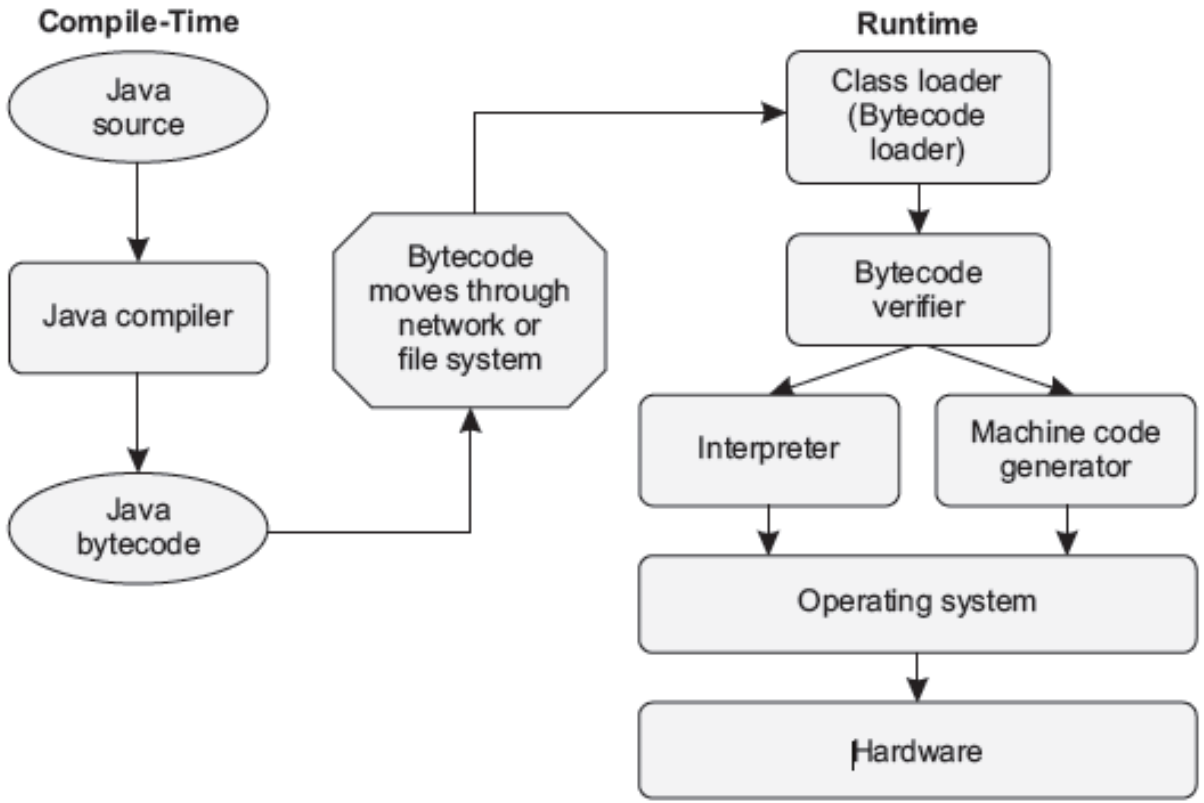


Fig. 2.6 Compilation and Interpretation in Java

Java Features

- Platform Independence
- Object oriented
- Compiled and interpreted
- Robust
- Security
- Strictly typed language
- Lack of pointers
- Garbage collection
- Strict compile time checking
- Sandbox security

Java Features

- Multithreaded
- Dynamic binding
- Performance
- Networking
- No pointers
- No global variables
- Automatic Garbage collection

Differences between C++ and Java

- **Operator overloading**
  - Not supported in Java
  - Exception is '+'
- **Explicit boolean type**
  - Boolean is an explicit type, different from int
  - Only two boolean literals are provided i.e. true and false
  - These cannot be compared with integers 0 and 1 as used in some other languages
- **Array length accessible**
  - All array objects in java have a length variable associated with them to determine the length of the array

Differences between C++ and Java

- **goto**
  - Instead of goto, break and continue are supported
- **Pointers**
  - There are no pointers in Java
- **null pointers reasonably caught**
  - Null pointers are caught by a NullPointerException
- **Memory management**
  - Explicit destructor is not needed
  - The use of garbage collection prevents memory leaks and referencing freed memory
- **Automatic variable initialization**
  - Variables are automatically initialized except local variables
- **Runtime container bounds checks**
  - The bounds of containers (arrays, strings, etc.) are checked at runtime and an IndexOutOfBoundsException is thrown if necessary.

Differences between C++ and Java

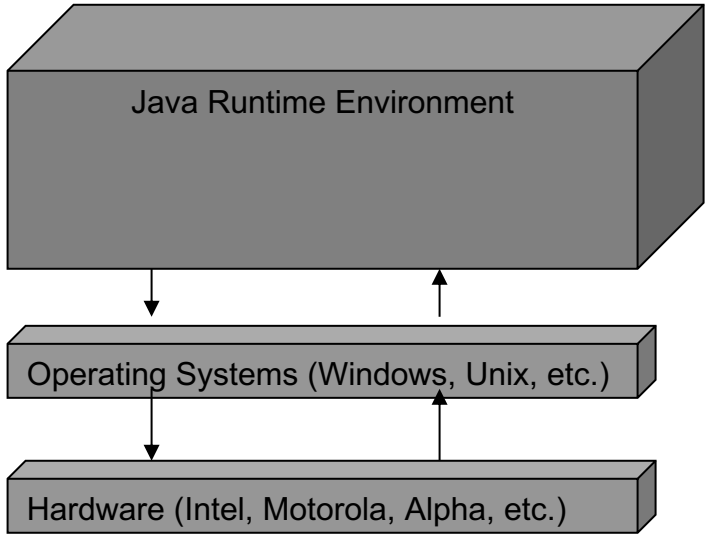
- **All definitions are well defined**
  - Methods and fields carry explicitly one of the access modifiers
- **Sizes of the integer types defined**
  - The sizes of the integer types byte, short, int and long are defined to be 1, 2, 4 and 8 bytes.
- **Unicode provided**
  - Unicode represents character in most of the languages for e.g. Japanese, Latin etc
- **String class**
  - An explicit predefined String class is provided along with StringBuffer and new StringBuilder class

Differences between C++ and Java

- **Extended utility class libraries: *package java.util***
  - Supported among others: Enumeration (an Iterator interface), Hashtable, Vector
- **Multithreading support with synchronization**
  - Java supports Multithreading with synchronization among them.
- **Default access specifier added**
  - By default, in java all variables, methods and classes have default privileges which are different from private access specifier
  - Private is the default access specifier in C++

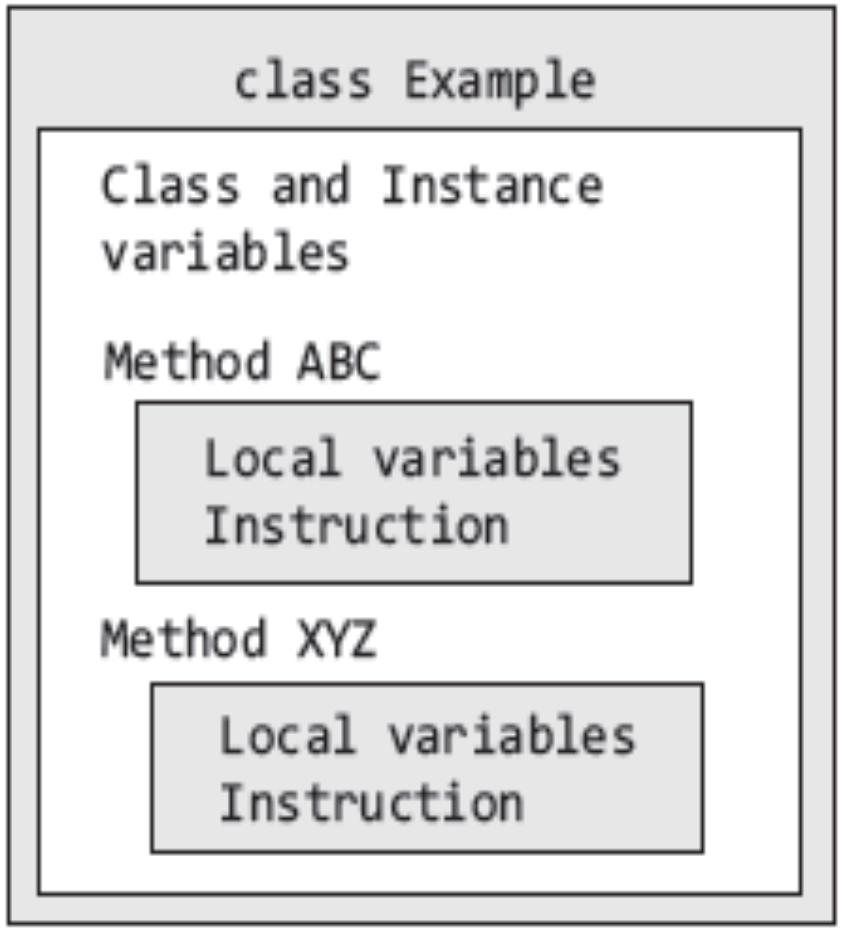
JVM and JRE

JVM is a part of JRE



Program Structure

- A Java Application consists of a collection of classes
- A class is a template containing methods and variables



First Java Program

```
/* Call this file “Example.java”.*/
class Example {
//your program starts execution with a call to //main()
public static void main(String args[ ]){
System.out.println(“This is a simple Java program”);
}
}
```

First Java Program

```
class Example {

  public static void main (String args[])
  {
 System.out.println("Welcome to OOPM 2022-23");
  }
}
```

Executing Java Programs

- **Entering the source code:** text editor like notepad or any IDE
- **Saving the source code:**
  - Select File | Save As from the notepad menu
  - In the 'File name' field, type "Example.java" within the double quotes
  - In the 'Save as type' field select All Files (*.*).
  - Click enter to save the file
- **Compiling & running the source**
  - type cmd at the run prompt
  - move to the folder that contains the saved Example.java file
  - compile the program using javac
  - C:\javaeg\>javac Example.java

Executing Java Programs

- Compilation creates a file called **Example.class**
- This class contains bytecode which is interpreted by JVM.
- To execute the program type the following command at the dos prompt:
  - C:\javaeg\>java Example
- The output of the program is shown below:
  - This is a simple Java program

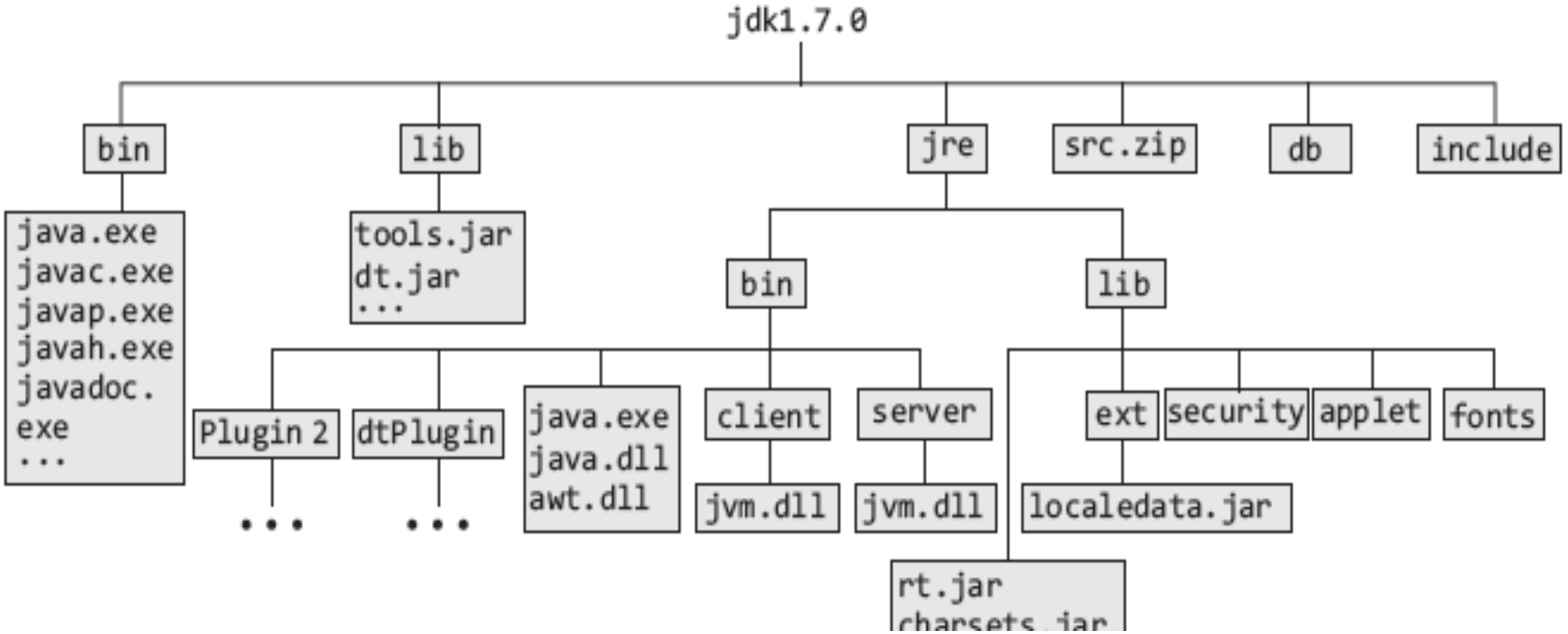
Why save as Example.java?

- The name of the .class file will match exactly with the name of the source file
- That is why it is a good idea to give the Java source files the same name as that of the class they contain
- Java is case-sensitive
- So example and Example are two different class names

Installation of Java

- Download the JDK installer
- Run the JDK installer.
- Update PATH Environment variables.
- Test the installation – run javac and java on command prompt

Installed Directory structure

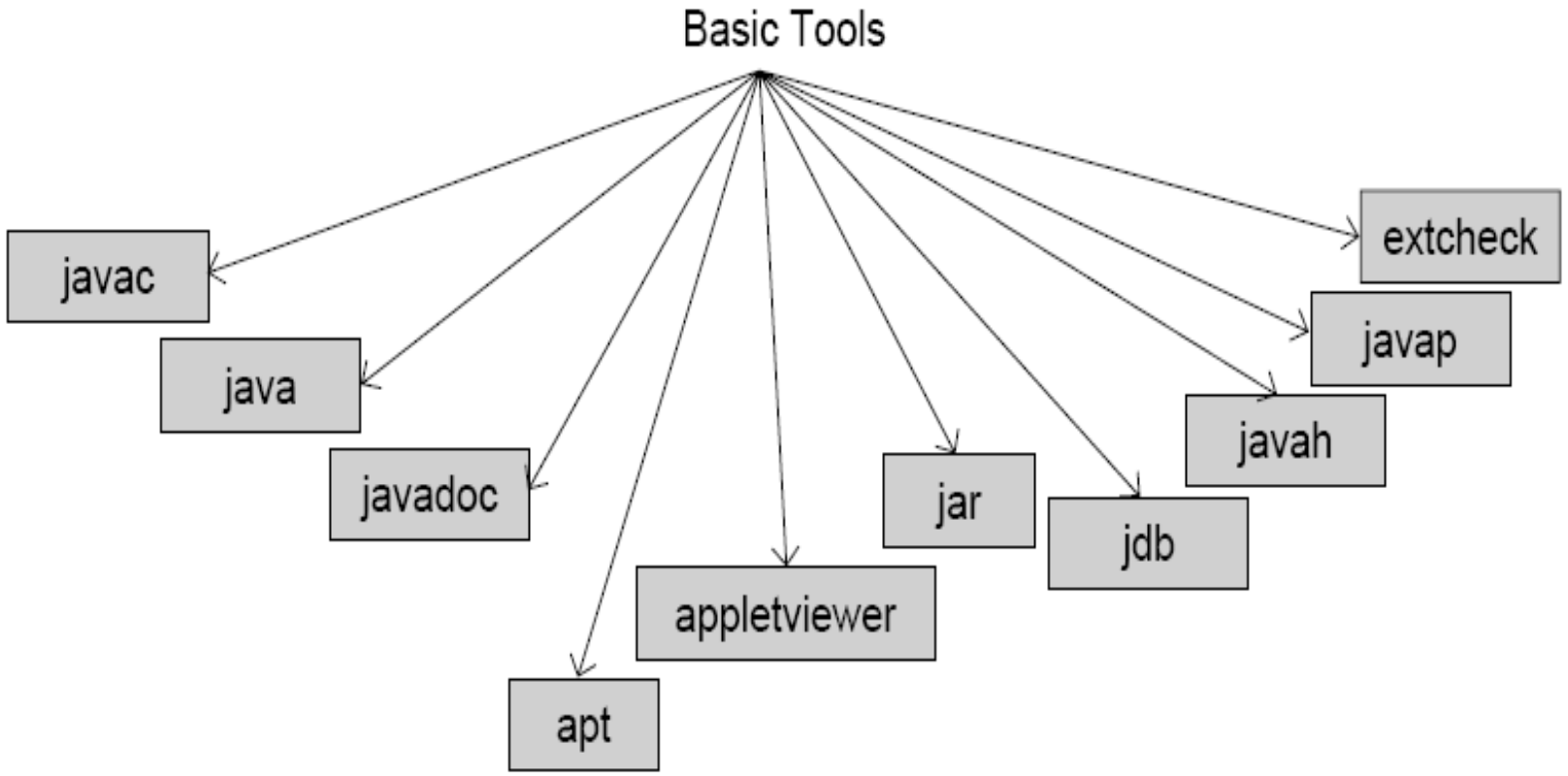


Installed Directory Structure

- **src.zip file** contains all the core class binaries, and is used by JDK in this form
- **include** directory contains a set of C and C++ header files for interacting with C and C++
- **lib** directory contains non-core classes like **dt.jar** and **tools.jar** used by tools and utilities in JDK
- **bin** The bin directory contains the binary executables for Java
  - For example, Java Compiler (Java), Java Interpreter (Java)
- **jre** is the root directory for the Java runtime environment
- **db** contains java database

Tools in JDK

Basic Tools in Java



IDE

- Tools specifically designed for writing Java code.
- Tools offer a GUI environment to compile and debug your Java program easily from the editor environment, as well as browse through your classes etc.
- Popular IDE's
  - Eclipse
  - Netbeans
  - Kawa
  - JCreator

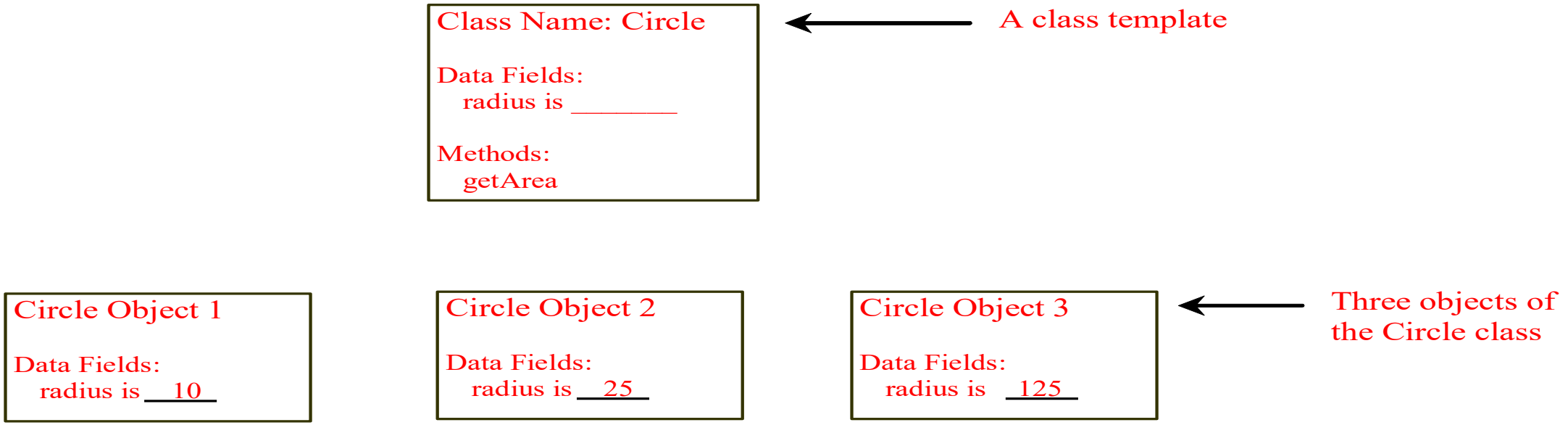
Objects and Classes

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values
- Example:
  - A circle object has a data field **radius**, which characterizes a circle
  - A rectangle object has the data fields **width** and **height**, which characterize a rectangle
- The *behavior* of an object (also known as its *actions*) is defined by methods
- To invoke a method on an object is to ask the object to perform an action
- Example:
  - **getArea()** and **getPerimeter()** maybe defined
  - **getArea()** maybe invoked by circle object to return its area and **getPerimeter()** to return its perimeter

Objects and Classes

- Objects of the same type are defined using a common class
- A *class* is a template, blueprint, or *contract* that defines what an object's data fields and methods will be
- An object is an instance of a class, many instances can be created
- Creating an instance is referred to as *instantiation*.
- The terms *object* and *instance* are often interchangeable

Objects



- An object has both a *state* and *behavior*
- The state defines the object, and the behavior defines what the object does

Classes

- *Classes* are constructs that define objects of the same type
- A Java class uses variables to define data fields and methods to define behaviors
- Additionally, a class provides a special type of methods, known as constructors
- Constructors are invoked to construct objects from the class

Classes

```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

Constructors

Method

OOP Concepts

- Key OOP Concepts
  - Object, Class
  - Data Abstraction
  - Encapsulation
  - Inheritance and Subclasses
  - Polymorphism
    - Run-time polymorphism
    - Compile-time polymorphism

Object

- Defined as instance of a class
- Example: table, chair are all instances of the class Furniture.
- Objects have unique identity, state and behavior
- State is defined by the attributes of the object.
- Different objects have different attributes (characteristics)
- Example: the attributes of student are name, roll number etc
- Behavior actually determines the way an object interacts with other objects.
- Synonym to functions

Object

- Definition: a thing that has identity, state, and behavior
  - identity: a distinguished instance of a class
  - state: collection of values for its variables
  - behavior: capability to execute methods

  - variables and methods are defined in a class

Class

- Blueprint for an object, a plan, or template
- Description of a number of similar objects is also called class
- A class is also defined as a new data type; a user defined type
- Defining a class doesn't create an object
- Classes are logical in nature.
- Example: Furniture does not have any existence but tables and chairs do exist

Class

- Definition: a collection of data (fields/ variables) and methods that operate on that data

  – define the contents/capabilities of the instances (objects) of the class
  – a class can be viewed as a factory for objects
  – a class defines a recipe for its objects

Instantiation

- Object creation
- Memory is allocated for the object's fields as defined in the class
- Initialization is specified through a constructor
  - a special method invoked when objects are created

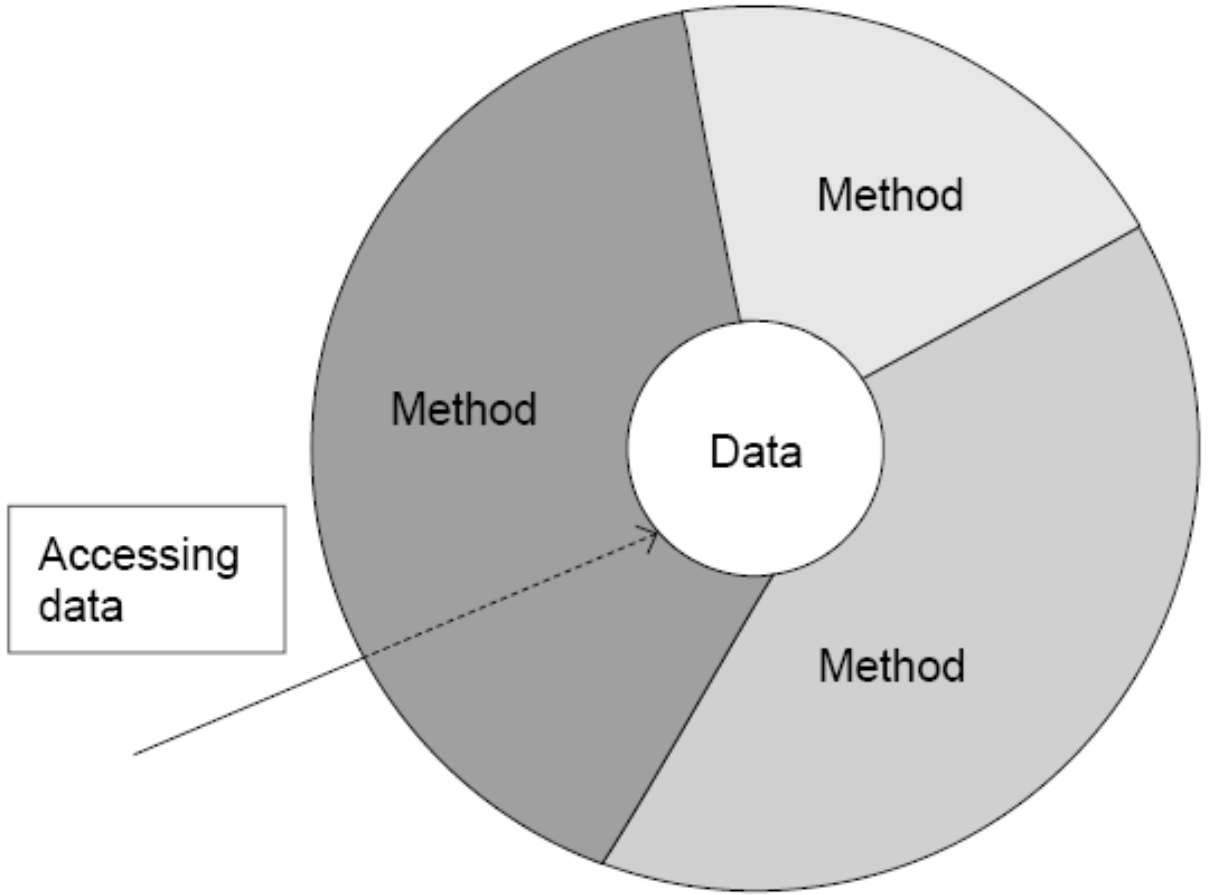
Abstraction

- OOP is about abstraction
- In real life, Humans manage complexity by abstracting details away
- In programming, we manage complexity
  - by concentrating only on the essential characteristics and
  - suppressing implementation details
- Encapsulation and Inheritance are examples of abstraction
  - What does the verb “abstract” mean?

Encapsulation

- A key OO concept: "Information Hiding"
- Key points
  - The user of an object should have access only to those methods (or data) that are essential
  - Unnecessary implementation details should be hidden from the user
  - Binding of data and procedure
  - In Java/C++, use classes and access modifiers (public, private, protected)

Encapsulation



Inheritance

- Inheritance:
  - programming language feature that allows for the implicit definition of variables/methods for a class through an existing class
- Subclass relationship
  - B is a subclass of A
  - B inherits all definitions (variables/methods) in A

Reuse

- Inheritance encourages software reuse
- Existing code need not be rewritten
- Successful reuse occurs only through careful planning and design
  - when defining classes, anticipate future modifications and extensions

Polymorphism

- "Many forms"
  - allow several definitions under a single method name
- Example:
  - "move" means something for a person object but means something else for a car object
- Dynamic binding:
  - capability of an implementation to distinguish between the different forms during run-time

Static and Dynamic Binding

- Association of method call to the method body is known as binding
- There are two types of binding:
  - **Static Binding** that happens at compile time
  - **Dynamic Binding** that happens at runtime

Static and Dynamic Binding

- If you have more than one method of same name (method overriding) or two variable of same name in same class hierarchy it gets tricky to find out which one is used during runtime as a result of there reference in code.
- This problem is resolved using static and dynamic binding in Java.
- Binding is the process used to link which method or variable to be called as result of the reference in code.
  - **Most of the references is resolved during compile time but some references which depends upon Object and polymorphism in Java is resolved during runtime when actual object is available.**
  - **When a method call is resolved at compile time, it is known as static binding, while if method invocation is resolved at runtime, it is known as Dynamic binding or Late binding.**

Static and Dynamic Binding

- private, final and static methods and variables uses static binding and resolved by compiler because compiler knows that they can't be overridden and only possible methods are those, which are defined inside a class, whose reference variable is used to call this method.
- Static binding uses Type information for binding while Dynamic binding uses Object to resolve binding.
- Static Binding
  - **Variables – static binding**

Variables are resolved using static binding which makes there execution fast because no time is wasted to find correct method during runtime.

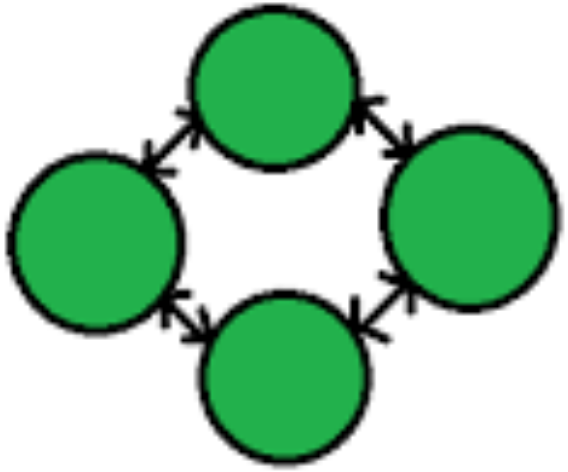
  - **Private, final, static methods – static binding**
  - **Overloaded methods are resolved using static binding**

- Dynamic Binding
  - **Overridden methods are resolved using dynamic binding at runtime.**

Coupling

Tight coupling:

1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

1. Less Interdependency
2. Less coordination
3. Less information flow

Coupling *refers to the extent to which a class knows about the other class.*

There are two types of coupling

- **Tight Coupling**(*a bad programming design*)
- **Loose Coupling**(*a good programming design*)

Tight Coupling

If a class A has some ***public*** **data members** and another class B is accessing these data members ***directly using the dot operator*** (which is possible because data members were declared **public**), the two classes are said to be **tightly coupled**

Let's say, the same class A has a String data member, *name*, which is declared **public** and this class also has *getter and setter* methods that have implemented some checks to make sure -

- A valid access of the data member, *name* i.e. *it is only accessed when its value is* ***not null***, and
- A valid setting of the data member, *name* i.e. *it cannot be set to a* ***null*** *value*

But these checks implemented in the methods of class A to ensure a *valid access* and *valid setting* of its data member, *name*, are *bypassed by its* ***direct access*** by class B, due to ***tight coupling*** between two classes.

Tight Coupling

Program Analysis

- Class **A** has an instance variable, *name*, which is declared **public**
- Class **A** has two **public** *getter and setter* methods which check for a valid access and valid setting of data member - *name*
- Class **B** creates an object of class **A** and directly sets the value of its data member, *name*, to **null** and directly accesses its value because it was declared **public**
- Thus, the validity checks for the data member, *name*, which were implemented within **getName()** and **setName()** methods of class A are *bypassed*, which shows that class **A** is *tightly coupled* to class **B**, and it is a *bad design*

Loose Coupling

- A good application designing is creating an application with loosely coupled classes by following **encapsulation**
- i.e. by declaring data members of a class with the **private** access modifier, which disallows other classes to directly access these data members, and forcing them to call the ***public*** **getter, setter methods** to access these *private* data members

Loose Coupling

Program Analysis

- Class **A** has an instance variable, *name*, which is declared **private**
- Class **A** has two **public** *getter and setter methods* which check for a valid access and valid setting of the data member, *name*
- Class **B** creates an object of class A, calls the **getName()** and **setName()** methods and their ***implemented checks are properly executed*** before the value of instance member, *name*, is accessed or set
- It shows that class **A** is *loosely coupled* to class **B**, which is a *good programming design*

Cohesion

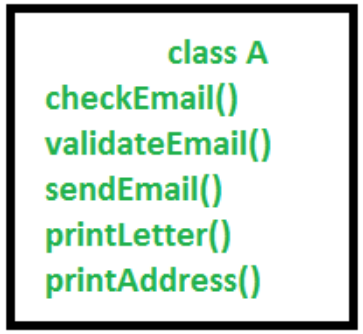


Fig: Low cohesion

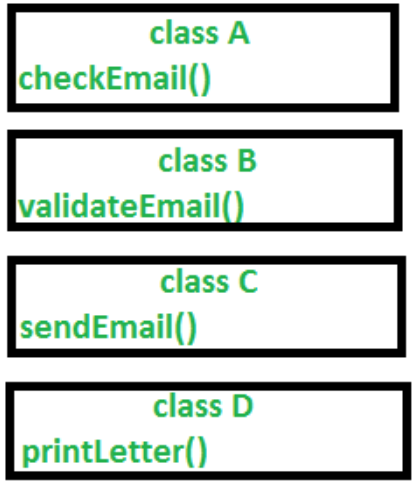


Fig: High cohesion

- **Cohesion** refers to the extent to which a class is defined to do a **specific specialized task**
- A class created with high cohesion is targeted towards a single specific purpose, rather than performing many different purposes

- There are two types of cohesion
  - **Low cohesion**(*a bad programming design*)
  - **High Cohesion**(*a good programming design*)

Low Cohesion

- When a class is designed to do **many different tasks** rather than focus on a *single specialized task*, this class is said to be a "*low cohesive*" class
- Low cohesive classes are said to be *badly designed*, as it requires a lot of work at creating, maintaining and updating them

Program

Example of a low cohesion class

```
class PlayerDatabase
{ public void connectDatabase();
public void printAllPlayersInfo();
public void printSinglePlayerInfo();
public void printRankings();
public void printEvents();
public void closeDatabase(); }
```

Low Cohesion

Program Analysis

- Here, we have a class **PlayerDatabase** which is performing **many different tasks** like connecting to a database, printing the information of all the players, printing information of a single player, printing all the events, printing all the rankings and finally closing all opened database connections

  Now, such a class is not easy to create, maintain and update, as it is involved in performing many different tasks i.e. *a programming design to avoid*

High Cohesion

- A good application design is creating an application with *high cohesive* classes, which are targeted towards a ***specific specialized task*** and such classes are not only easy to create but also easy to maintain and update

Program

```
//Example of high cohesion classes
class PlayerDatabase {
ConnectDatabase connectD= new connectDatabase();
PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();
PrintRankings rankings = new PrintRankings();
CloseDatabase closeD= new CloseDatabase();
PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo(); }
class ConnectDatabase { //connecting to database. }
class CloseDatabase { //closing the database connection. }
class PrintRankings { //printing the players current rankings. }
class PrintAllPlayersInfo { //printing all the players information. } class
PrintSinglePlayerInfo { //printing a single player information. }
```

High Cohesion

Program Analysis

- Here we have created several different classes, where each class is performing a specific specialized task, which leads to an easy creation, maintenance and modification of these classes
- Classes created by following this programming design are said to performing a cohesive role and are termed as *high cohesion classes*, which is an appropriate programming design to follow while creating an application

Thank You