Programming in Core Java

Faculty: Ajay Pooreti

L&T Technology Services

LTTS GLOBAL ENGINEERING ACADEMY

Agenda

- Introduction to JAVA
- JAVA Basic Fundamentals
- Control Flow Statements
- Object Oriented Programming Concepts
- Inheritance
- polymorphism
- String, String Buffer, StringBuilder

Agenda

Version

Version	Reviewed by	Approved by	Remarks
1.0	Salman Hamza Hussain	KNS Acharya	

Embed Syllabus of the Couse on this slide

Learning Outcome of the Course

At the completion of the Course participants will able to:

- Understand and apply basic Java programming concepts
- Write applications using general concepts of Java
- Use different in-built and third party packages of Java.
- Understand how to use Eclipse IDE for development and debugging of Java applications.

Pre-Requites for the Course

Having knowledge in C programming. (C++ preferable).

Introduction to JAVA

Java History

June 1991

Java was started as a project called "Oak" by James Gosling in June 1991.

JDK 1.0 released in

23 Jan. 1996

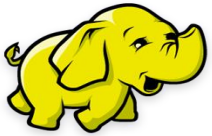
20 Oct. 2020

Java SE 15

What is Java?

- Java is a Programming Language.
- Java is a High-level Language.
- Java supports Object Oriented Programming concepts.
- Java is a case sensitive programming.
- Java can be called as functional programming(from Java 8)
- Java can be called as modular programming (from Java 9)

Where Java is mostly used?



Hadoop & Big Data

Android Apps

Web applications (Spring, Hibernate etc)

Mobile Games

Data Analytics

Java Features or Buzzwords

- Simple
- Object Oriented
- Platform Independent
- Portable
- Robust
- Security (bytecode)
- Distributed
- Multithreading
- Exception Handling(try, catch)

Simple

Syntax is Simple.

Easy to understand.

Removed many Complicated features.

Object – Oriented

- Object oriented is a methodology to solve the problems.
- It has various concepts like class, object, data abstraction, inheritance polymorphism.

Platform Independent

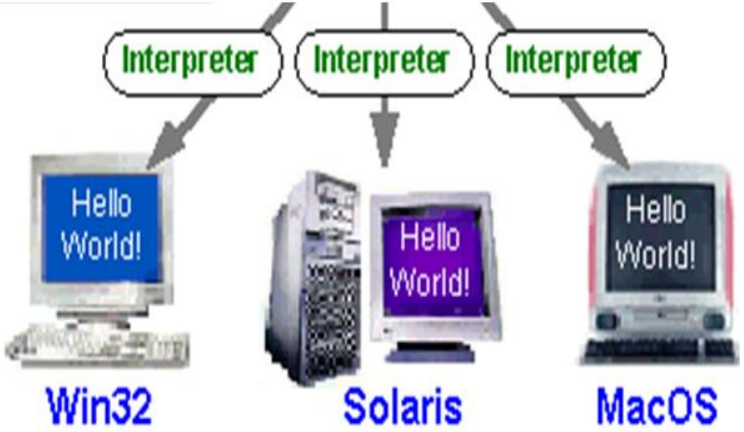
```
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

javac HelloWorld.java

HelloWorld.class

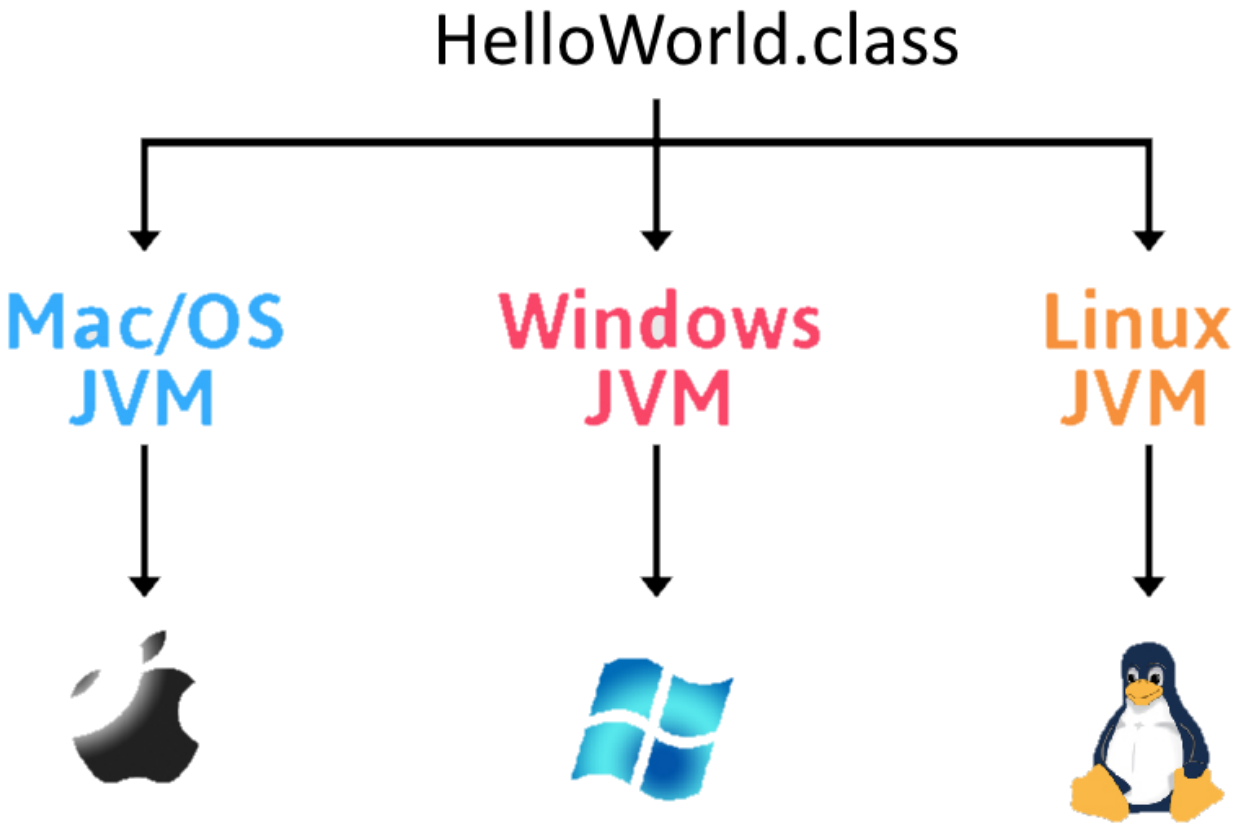
javac – JAVA compiler. Converts the .java files to .class files.

java - The Interpreter's referred to as the JAVA Virtual Machines (JVM)



java HelloWorld

Portable



Architectural Neutral

- If you upgrade your system with any hardware architecture, it will execute.
- In previously upgrade the 32bits to 64 bits operating system it works.
- It portable is a part of architectural neutral

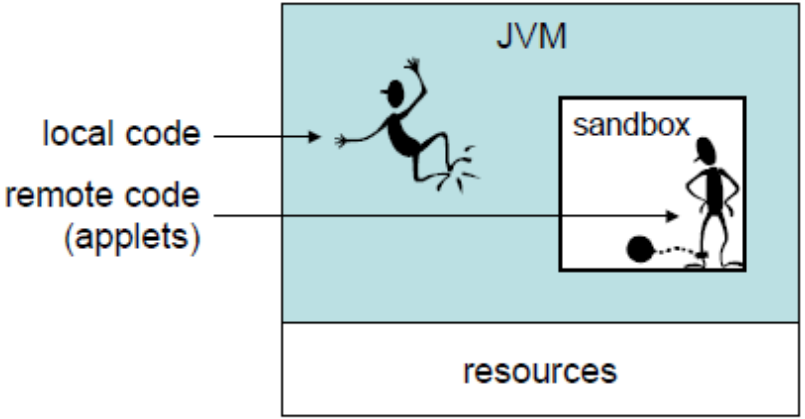
Robust

It uses the Memory management mechanism

It contains automatically garbage collection used by jvm.

Java Security

idea: limit the resources that can be accessed by applets



- introduced in Java 1.0
- local code had unrestricted access to resources
- downloaded code (applet) was restricted to the sandbox
  - cannot access the local file system
  - cannot access system resources,
  - can establish a network connection only with its originating web server

Distributed

Facilitates to create the Distributed application.

Concepts like RMI, to create the distributed application.

Access the method from outside of the system through internet.

Multithreaded

- If we want to execute the statements concurrently.
- Programmer to write the code to execute concurrently.

Java Editions

- J2SE – Java 2 Standard Edition (Core Java)
  - Able to create the Desktop Applications. Like calculator, games ..etc

- J2EE – Java 2 Enterprise Edition (Advance Java)
  - Able to create the Web Applications and Enterprise (Distributed) Applications

    Web Based or Distributed Application

    Ex: Servlet, Jsp or Ex: Spring framework
  - Technologies involved Servlets, JSP, EJP, JMS etc.

- J2ME – Java 2 Micro Edition
  - Able to create the Sensor / Embedded -based applications.

Note: If you want to learn J2EE or J2ME – You must need to learn J2SE.

JDK

- JDK – Java Development kit used to develop the code.
- It contains a private Java Virtual Machine (JVM).
- interpreter/loader (java) for execution.
- a compiler (javac) to compile the code.
- an archiver (jar) to archive the all codes.
- a documentation generator (Javadoc).

JRE

JRE – Java Runtime Environment

It provides the runtime environment.

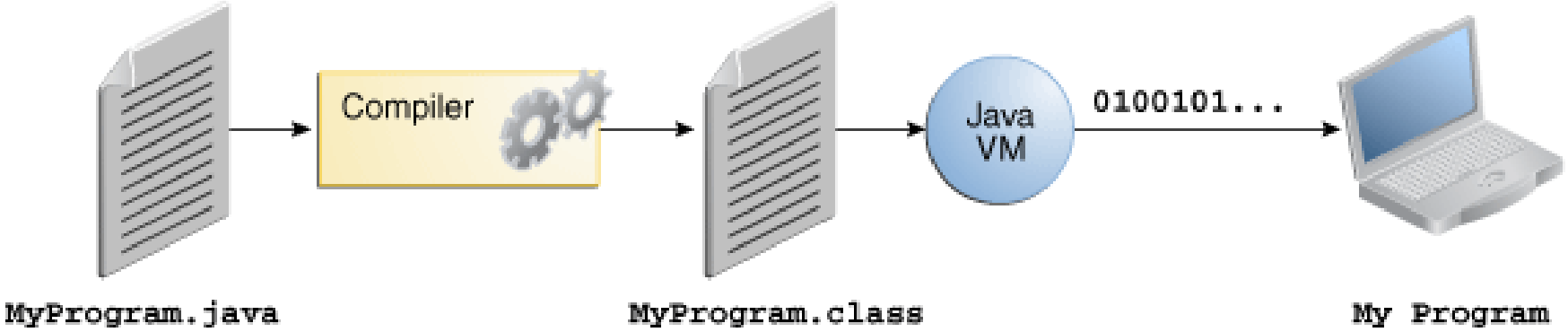
It contains a set of libraries + other files that JVM uses at runtime.

JVM

- JVM – Java Virtual Machine
- JVM is a specification that provides a runtime environment in which Java bytecode can be executed.
- It performs the following tasks
  - Load the code
  - Verifies the code
  - Executes the code
  - Provide the runtime environment.

About Java Program

- Java Programs written in plain text files.
- Save with .java extension.
- For example write the code and save as MyProgram.java
- Use javac command to compile code
- Converts into MyProgram.class file created by the compiler if no error.
- .class file contains the bytecode.
- Execute the code using java command then .class file will be executed.


Compiler
Java
VM
0100101...
MyProgram.java
MyProgram.class
My Program

Structure of Java Program

package statement

import statements….

class One

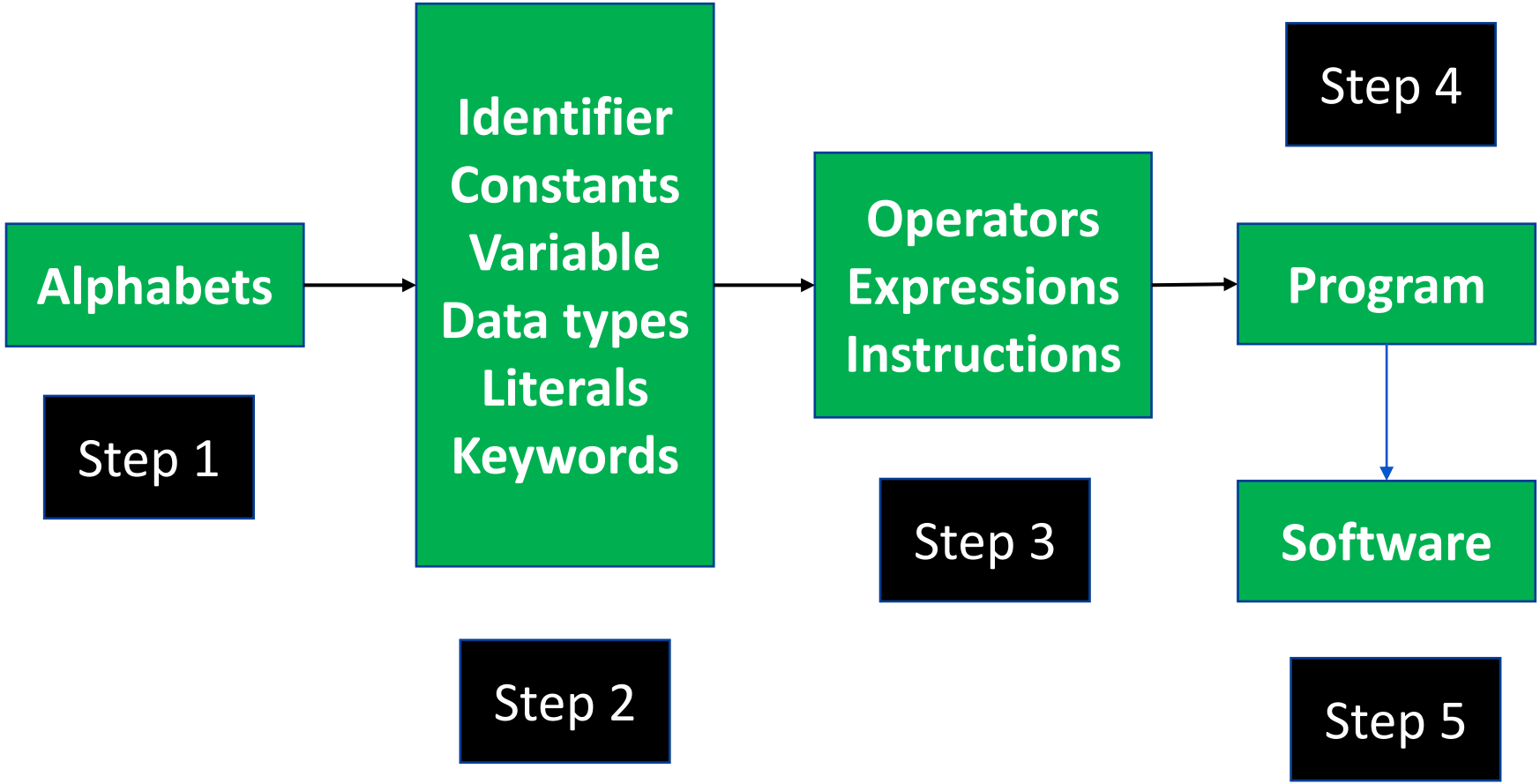
member1, member2 …..

method1

Method n

class n

Steps to Learn Programming Language



JAVA Basic Fundamentals

Fundamentals

- Identifier
- Keywords or Reserve words
- Data Types
- Variables
- Literals
- Expressions and operators
- Statements
- Arrays

Identifier

Identifier is just a name.

The name may be class, method, interface, variable, package etc.

Sample Program

```
package com.example.main;
import java.io.*;
class MyProgram{
  public static void main(String []args){
        System.out.println("Hello Welcome to LTTS");
  }
}
```

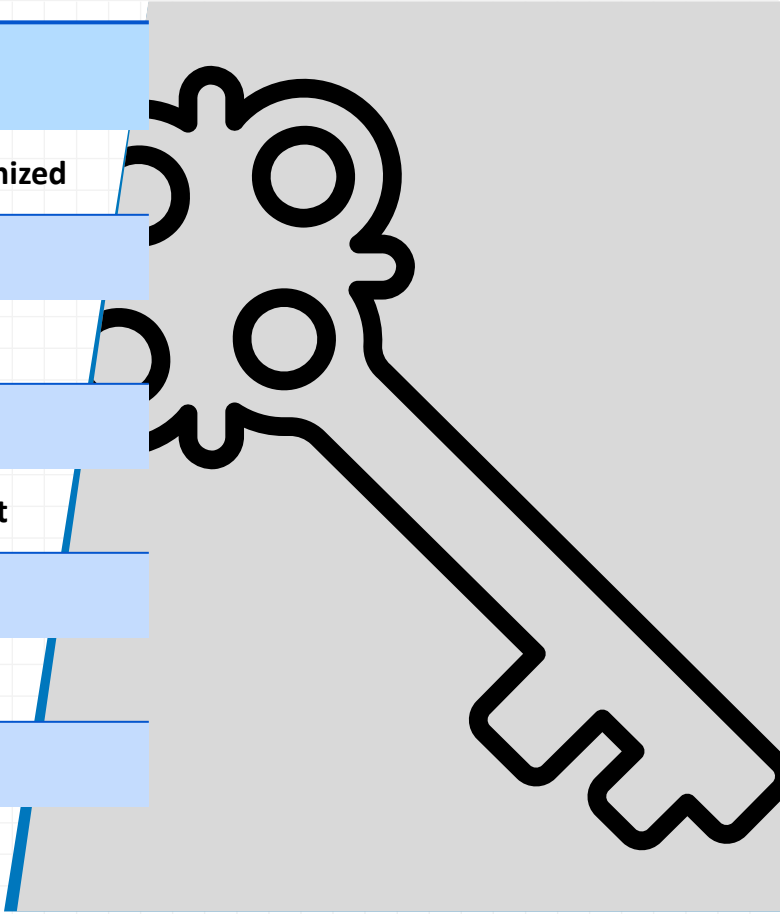
Can you Guess identifiers?

Identifiers are

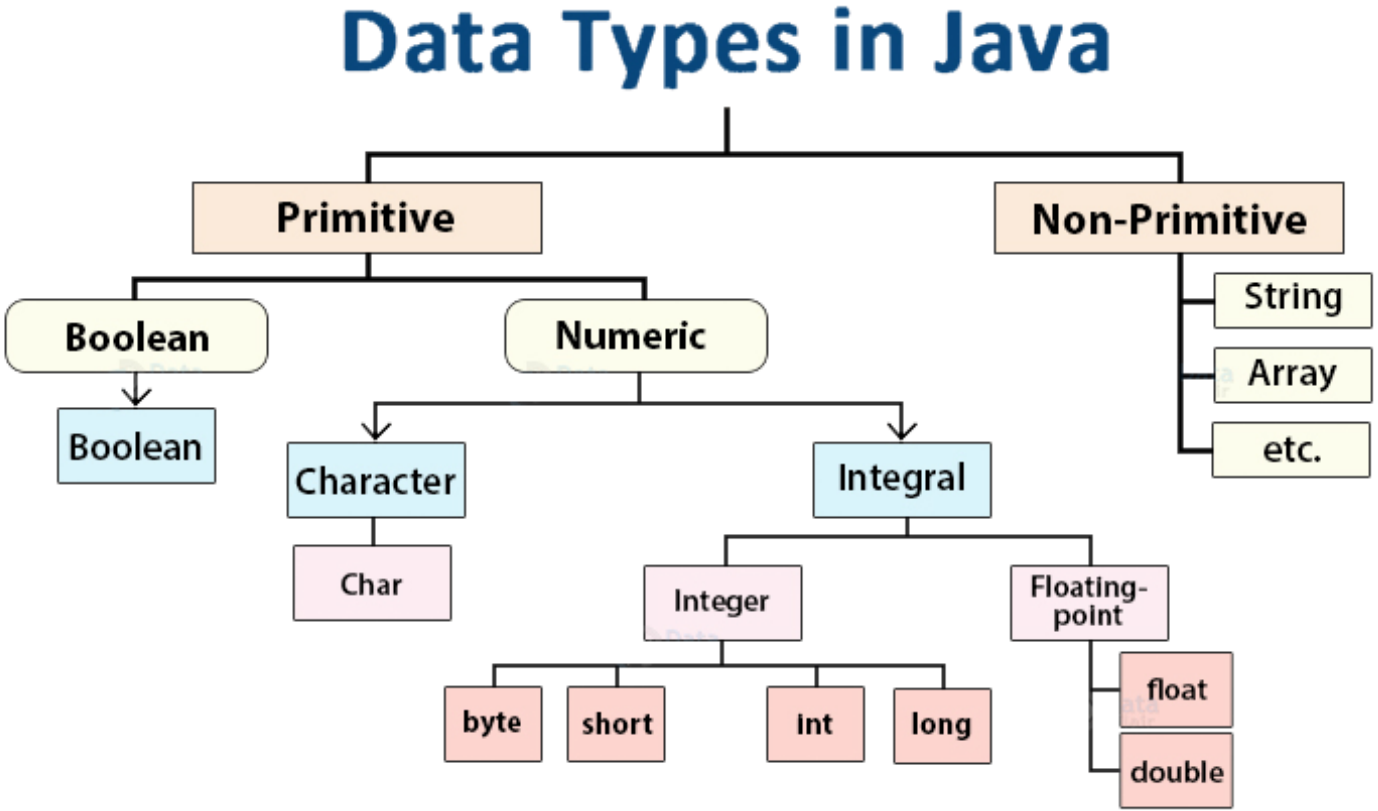
```
package com.example.main;
import java.io.*;
class MyProgram{
  public static void main(String []args){
      System.out.println("Hello Welcome to LTTS");
  }
}
```

Keywords or Reserve Words

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



Data Types


Data Types in Java
Primitive
Non-Primitive
Boolean
Numeric
String
Array
etc.
Boolean
Character
Integral
Char
Integer
Floating-point
byte
short
int
long
float
double

Data Type	Default Value	Default size
boolean	false	N/A
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Variable

- Variable is a name to store or hold a value in it.
- The name must start with alphabet, underscore(_) or dollor ($).
- Common Syntax

  <access specifier><special keyword>< type> <variable_name >;
- Access specifiers are public, private, protected
- Keywords will not be allowed for name.
- Types of Variables are
  - 1. Local Variable
  - 2. Instance Variable
  - 3. Reference Variable
  - 4. Static Variable
  - 5. Final Variable
  - 6. Transient Variable

1. Local Variable – variable which is declared inside the method.

- **Syntax – <><datatype> <variable name>;**
- No access specifier will not be allowed. Only final Allowed.

2. Instance Variable – variable which is declared outside the method.

- **Syntax - <access specifier> <datatype> <variable name>;**
- Special key words are not allowed.

3. Reference variable – variable which is declared inside or outside the method.

- **Syntax - <type> <variable_name>;**
- Here type refers the class name.
- This variable always holds the value as object only.

4. Other Variables - variable which is declared outside the method.

- **Syntax - <access specifier> <Special keyword> <datatype> <variable name>;**
- Special words are: **final, static, transient**

Example Local Variable

- The variable which is declared inside the class and inside the method.
- No Access specifier allowed.
- Create using the primitive data type.

```
class MyProgram{

  public static void main(String []args){
        int a; //local variable
        final int b; //local variable
        System.out.println("Hello Welcome to LTTS");
  }
}
```

Example Instance Variable

- The Variable declared inside the class and outside the method.
- It allows access specifier like public, private and protected.
- Create using the primitive data type.

```
Class MyProgram{
   public int a; //instance variable
   private int b; //instance variable

   public static void main(String []args){
         System.out.println("Hello Welcome to LTTS");
   }
}
```

Example Reference Variable

- Declare inside or outside the method.
- If you declare inside the method no access specifier allowed
- If you declare outside the method access specifier allowed.
- Created using the predefined or user defined class.

```
class MyProgram{
  public MyClass myc; // Reference variable
  public static void main(String []args){ // Predefined Reference array variable
        MyClass myc; //Reference Variable
        System.out.println("Hello Welcome to LTTS");
  }
}
```

Example – Static Variable

- The Variable declared inside the class and outside the method.
- It allows access specifier like public, private and protected.
- Object does not need to access this variable.
- Create using the primitive data type.

```
Class MyProgram{
   public  static int a; //instance variable
   private static int b; //instance variable

   public static void main(String []args){
         System.out.println("Hello Welcome to LTTS");
   }
}
```

Literals

Literals are the representation of digits.

Integer Literals are

- Decimal representation = (0-9) 123;
- Hexa decimal Representation = 0xCAB1 (0-9 a-f)
- Octal Representation = (0-7) 023;
- Binary Representation = 0b1101; (0&1)

Floating point Literals

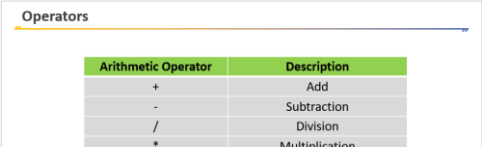
Character Literals

Example – Integer Literals

```
class MyProgram{
   public static void main(String []args){
         int a = 0b1101;  //Binary representation – Base 2
         int b = 023;  //Octal Representation – Base 8
         int c=123; // Decimal Representation – Base 10
         int d = 0xCAFE; // Hexa Decimal Representation – Base 16.
         System.out.println(a+” “+b+” +c+” “+d);
   }
}
```

Operators

Arithmetic Operator	Description
+	Add
-	Subtraction
/	Division
*	Multiplication
%	Remainder of
++	Prefix/Postfix Increment
--	Prefix/Postfix Decrement



Relational Operator	Description
==	Equals
!=	Not equal
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Bitwise Operator	Description
&	AND
|	OR
~	One’s compliment
^	Binary XOR
<<	Binary left
>>	Binary Right

Logical Operator	Example
&&	i > j && j < k
||	i < j or j < k
!	not j < k

Working with Operators

```
class MainDemo{
   public static void main(String []args){
         int a=10; //assignment operator
         int b = 20; //assignment operator
         int c = a+b; // assignment and arithmetic operator
         System.out.println("Result :"+c);
   }
}
```

Expression

```
class MainDemo{
   public static void main(String []args){
         int a = 2*3/6+4+4+4/4-2+5/8;
         System.out.println("Result: "+a);
   }
}
```

What will be the output?

Control Flow Statements

Control Flow - if statement

Syntax:

```
if (<condition>)
{
        statement(s);
}
```

Example:

```
i = 10;
if(i  > 0)
{
  System.out.println("i is positive");
}
```

Control Flow - if and else

Syntax:

```
if (<condition>)
{
   statement(s);
}
else
{
   statement(s);
}
```

If the condition expression is **True** then statement(s) in “if” block will be executed. Otherwise statement(s) in “else” block will be executed.

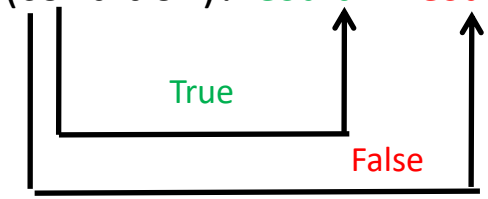
Control Flow – if and else if

Syntax:

```
if (<condition>)
{
    statement(s);
}
else if (<condition>)
{
   statement(s);
}
else
{
   statement(s);
}
```

Inline conditions

(condition)?result-1:result-2;



Control Flow – while loop

Syntax:

```
while (<condition>)
{
        statement(s);
}
```

If the condition expression is **True** then statement(s) in “while” block will be executed.

Note: The statement(s) will be executed continuously till condition expression is **False**.

for loop

Syntax:

```
for(<initialization> ; <condition> ; <inc/dec> )
{
  statement(s);
}
```

Constants

If a variable is declared as constant, it can not be changed any where in the program.

```
final int  i = 100;
i = i + 1;   → ERROR: "i" is constant, so it can not be updated.
```

Array

- An Array is a sequential indexed collection of Homogeneous elements with fixed in size.
- Array contains 3 phases
  - 1. Array Declaration
  - 2. Array Definition
  - 3. Array Initialization
- Array will be declared as with the help of primitive type and class with [ ].

  Ex: int[] a; Myclass []myc;
- Array will be defined as size as integer type.

  Ex: =new int[10]; =new Myclass[10];
- Array values will be initialized with its index value.

  Ex: =a[2] =10; myc[1]=new MyClass(1,”Thiru”);

Arrays

Syntax:

```
<datatype> [ ] variable = new <datatype> [array_length]
```

Example:

```
int [] marks = new int[10];

marks[0] = 79;
marks[1] = 87;
…
marks[9] = 91;
```

Object Oriented Programming Concepts

OOPs Features

- Object Oriented Programming that uses the object in the program.
- Is a paradigm for an approach to solve the problem in real time.
- Focuses on how these concepts relate to the real world.
- To bind the data and methods that operate.
- To increase the flexibility and maintainability of programs.

Concepts

- Classes and Objects
- Data Abstraction
- Data Encapsulation
- Inheritance
- Polymorphism
- Message passing
- Dynamic Binding
- Constructor
- Coupling
- Cohesion

Classes and Objects

For example, we want to develop a module to process employee information for an organization.

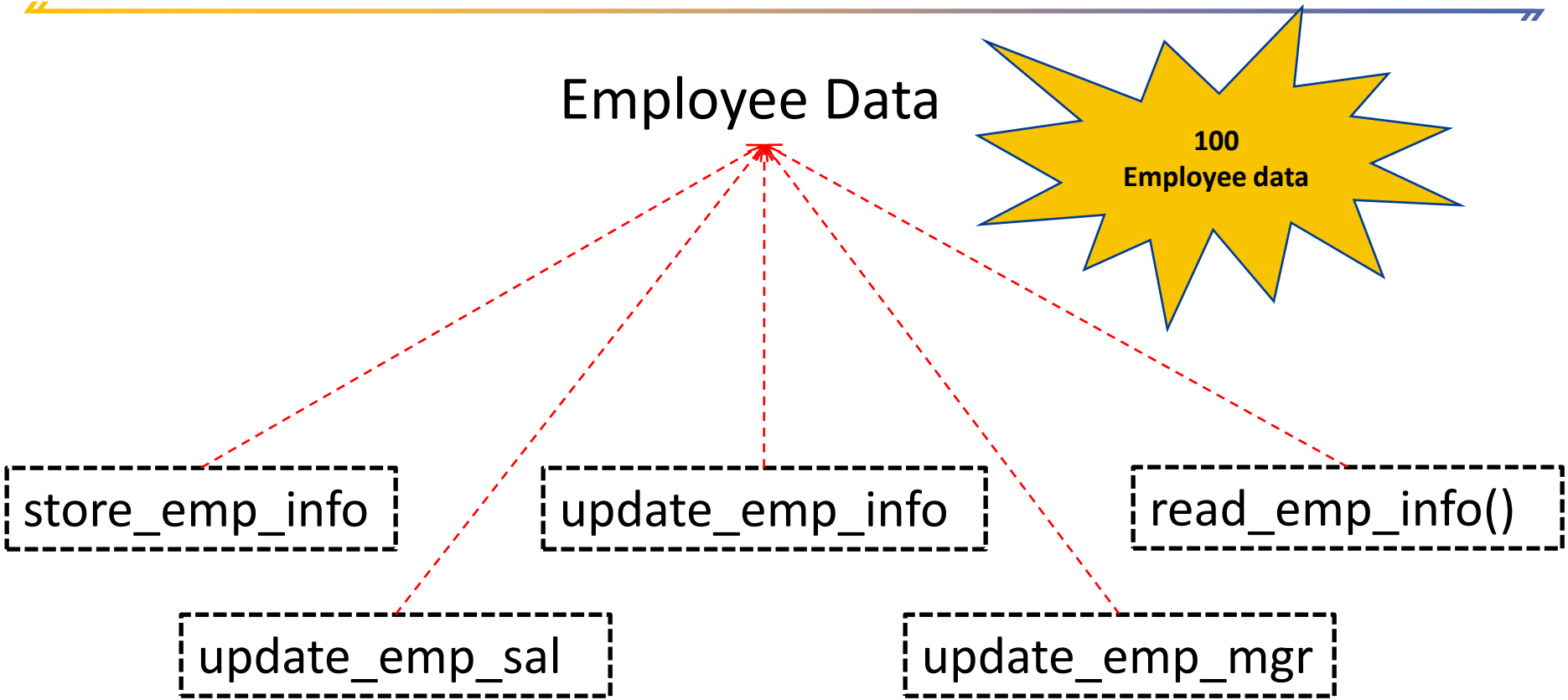
Solution (Using procedure language like C):

Let us consider the following data for 100 employees.

a. id [100]
b. name [100]
c. dept [100]
d. manager [100]
d. salary [100]

write functions to process the above information for 100 employees.

Disadvantages Solution



1. How to provide the security to employee data from modifications by different functions.
2. How to control the concurrent updates of employee data.

Class

- Blueprint of the program.
- Basic unit of program.
- Class contains members and methods.
- Class contains state and behavior.

- Syntax

```
<access specifier> <special key word> class <class name>
{
  member1 … member n
  method 1{} ……..
  method n{}
}
```

- Special key word like abstract, final used.

Class

Class is group of variables and methods which operates on the variables defined in the group.

Class contains state and behavior.

Syntax:

```
<access specifier> class <class_name>
{
   <access modifier>  <data type>  varaiable-1;
   <access modifier>  <data type>  varaiable-2;
   ……
}
```

<access modifier> can be one of public/private/protected/

Class Declarations

```
class MyClass{
  //member declaration
  //method definition
}
```

- The *class body* (the area between the curly braces) contains all the code.
- Provides for the life cycle of the objects created from the class: constructors for initializing new objects.
- Declarations for the fields that provide the state of the class and its objects.
- Methods to implement the behavior of the class and its objects.

Class Specification

We can declare and define any number of class inside the program.

You can save .java at any name.

But if you declare a class with public you provide same name to program.

You have only one class as public in one program.

Types of classes:

- Normal or default class
- Public class – you must save the program as class name only
- Abstract class – is an incomplete class
- Final class – is a complete class.

```
class Employee
{
    public    int     id;
    public    String  name;
    public    String  dept;
    public    int     manager;
    public    float   salary;
}
```

Access Modifier

Object

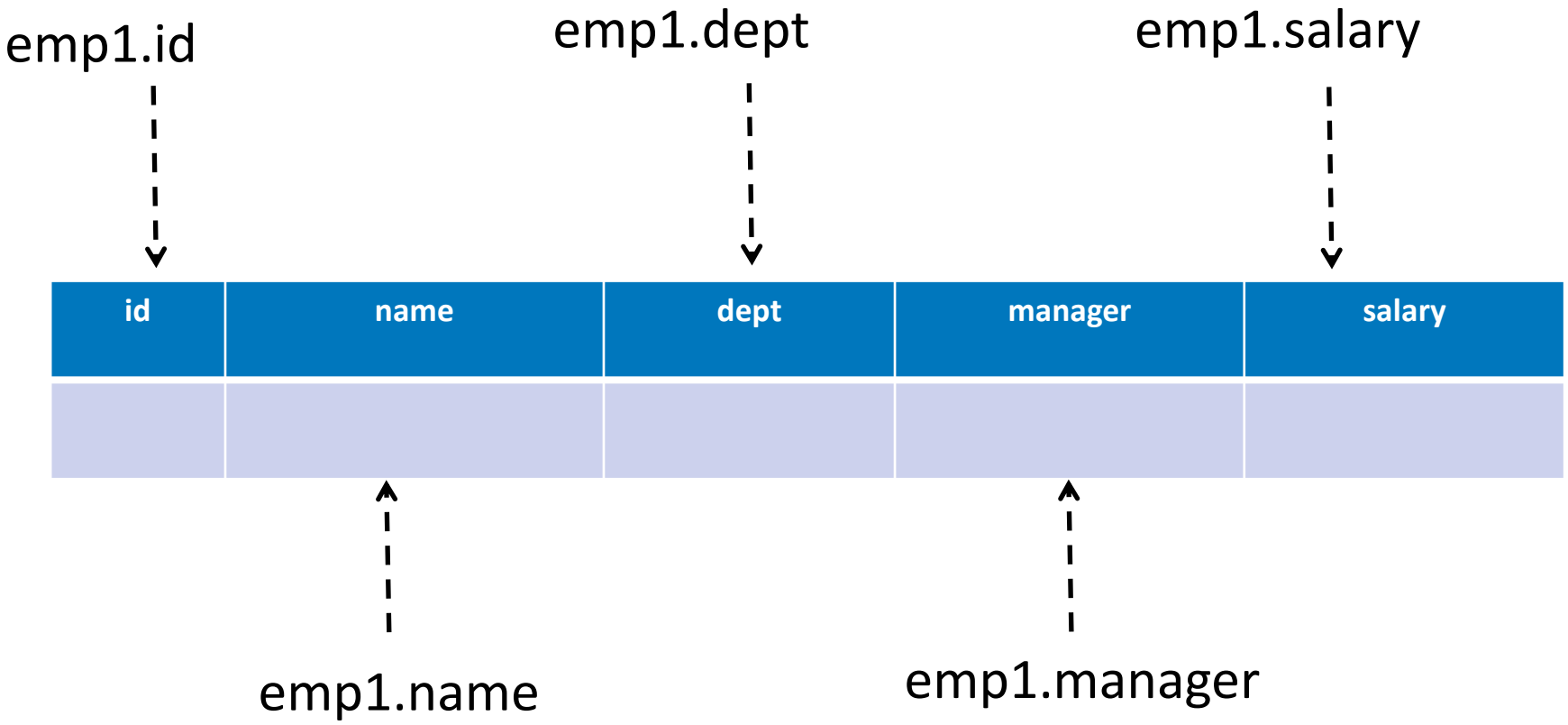
Object is an instance of a class.

1. *int count = 100;*

   // count is an object of "int"

2. *Employee emp1 = new Employee();*

   // "new Employee()" is an object (instance) of "Employee" class which is assigned to emp1.


emp1.id
emp1.dept
emp1.salary
id
name
dept
manager
salary
emp1.name
emp1.manager

emp1 ---->

id	name	dept	manager	salary
100	Name1	TESTER	200	9999.99

emp1.id = 100;

emp1.name = “Name1”;

emp1.dept = “TESTER”;

emp1.manager = 200;

emp1.salary = 9999.99;

Access Modifiers

Access modifier indicates whether member variables (variables defined in class) is accessible through object or not.

Access Modifier	Description
public	Visible to world
private	Visible to class only
protected	Visible to package and subclasses.

Example:

```
class Employee
{
    public      int     id;
    private     String  name;
    public      String  dept;
    protected   int     manager;
    private     float   salary;
}

Employee  emp1  =  new Employee();
```

emp1 ---->

id	name	dept	manager	salary
100		TESTER	222	

emp1.id = 100;

emp1.name = “Name1”; X Not accessible to world

emp1.dept = “TESTER”;

emp1.manager = 222;

emp1.salary = 111.12; X Not accessible to world

Method

- Method is a collection of statements or instructions.
- To declare the method, The minimum required elements of a method declaration are
  - *the method's return type,*
  - *name,*
  - *a pair of parentheses, (),*
  - *and a body between braces, {}.*
- The others are
  - *Modifiers – public , private, protected and default.*
  - *Special keywords – static, final, abstract, synchronized*
  - *Parameter list – in between the pair of parentheses.*
  - *Exception list.*

Method

Class can contain method which operates/modifies the variables (member variables) defined in the same class.

Syntax:

```
class <class_name>
{
    <access modifier>  <special word> <return_type> function_name ( [arguments])
    {
        // Function body.
    }
}
```

Example Method

```
public double calculateAnswer(double x, double y){
        // statements or instructions
        return double;
  }
```

- Public is access specifier.
- Double is a return type
- calculateAnswer – is the method's name.
- Double x, double y are the parameters.
- return double is the statement of the method.

Types of methods

- Instance Method
- Static Method
- Abstract Method
- Final Method
- Synchronized Method.
- constructor

Example

```
class Employee
{
   public int  id;
   public String name;
   public String dept;

   public void display()
   {
     System.out.println("Id:" + id);
     System.out.println("Name:" + name);
     System.out.println("Department:" + dept);
   }
}
```

Member variables

Method

Method Definition

```
class Employee
{
   public int  id;
   public String name;
   public String dept;

   public void display()
   {
      System.out.println("Id:" + id);
      System.out.println("Name:" + name);
      System.out.println("Department:" + dept);
   }
}
```

id	name	dept
200	Name2	TESTER-2

```
display()
{
    System.out.println("ID: " + id);
    System.out.println("Name: " + name);
    System.out.println("Department : " + dept);
}
```

emp2

```
emp1.id  = 200;

emp1.name  = "Name2";

emp1.dept  = "TESTER-2";

emp1.display();
```

Data Encapsulation

The Hiding up of data into a single unit.

Steps to implement the Encapsulation

- 1. Mark you instance variable as “private”
- 2. Use public get and set methods to access and the set the value for instance variable.

Setter and Getter Methods

According to the coding standards, all member variables in class must be "private". Then how to set/get the values to/from object of class?? Answer is getter and setter methods.

```
class Employee
{
   private int  id;

   public void setId(int id1)  ----------> Setter method for "id"
   {
      id = id1;
   }

   public int getId()  ----------> Getter method for "id"
   {
      return id;
   }
}
```

Overloading member functions

Function Overloading: Defining multiple functions with same name but different parameters.

```
public  void  setId()
{
   id = 100;
}
public void setId(int id1)
{
   id = id1;
}
```

Overloading member functions

```
public class Sample {
void setDisplay()
{
System.out.println("Method overloading");
}
void setDisplay(int a)
{
System.out.println("a value is"+a);
}
public static void main(String []args)
{
Sample s=new Sample();
s.setDisplay();
s.setDisplay(10);
}}
```

Constructors

```
class Employee
{
    public int  id;
    public String name;
    public String dept;
};

Employee emp1 = new Employee();
```

emp1 ----->

id	name	dept
0	null	null

When object created, some garbage values will be populated in member variables.

Constructors – cont'd

Constructor is a member function of class with

a. Function name same as class name.

b. No return type.

Example

```
class Employee
{
    public int  id;
    public String name;
    public String dept;

    public Employee()
    {
        id = 100;
        name = “Name1”;
        dept = “TESTER”;
    }
}
```

Employee emp1 = new Employee();

During object creation, constructor of the class AUTOMATICALLY invoked by JVM.

Constructors – Types

- Constructors with parameters
- Overloaded constructors
- Copy constructor

static variables , methods and blocks

Static declaration allows to access the variables of a class WITHOUT DECLARING the object.

Syntax: <class name>.<variable name>

```
class Test
{
    public static int id = 10;
}
System.out.println(Test.id);
```

Class name

static variable

```
package com.ltts;
public class Sample {
static int a=10;//static variable in java
public static void main(String []args)
{
System.out.println(Sample.a);
 }
}
```

static method

```
package com.ltts;
public class Sample
 {
static int a=10;
static void infoDisplay()//static method
{
System.out.println(a);
}
public static void main(String []args)
{
Sample s= new Sample();
Sample.infoDisplay();
}
}
```

static block

```
package com.ltts;
public class Sample
 {
static int a=10;
Static{
System.out.println("static block")
}
static void infoDisplay()//static method
{
System.out.println(a);
}
public static void main(String []args)
{
Sample s= new Sample();
Sample.infoDisplay();
}}
```

Inner classes

Syntax:

```
class <classname1>
{
    // Outer class definitions.
   class <classname1>
    {
        // Inner class definitions.
    }
}
```

Inheritance

Inheritance

Inheritance is the concept of a child class (sub class) automatically inheriting the variables and methods defined in its parent class (super class).

Deals with IS-A Relationship and HAS-A relationship.

Need for Inheritance

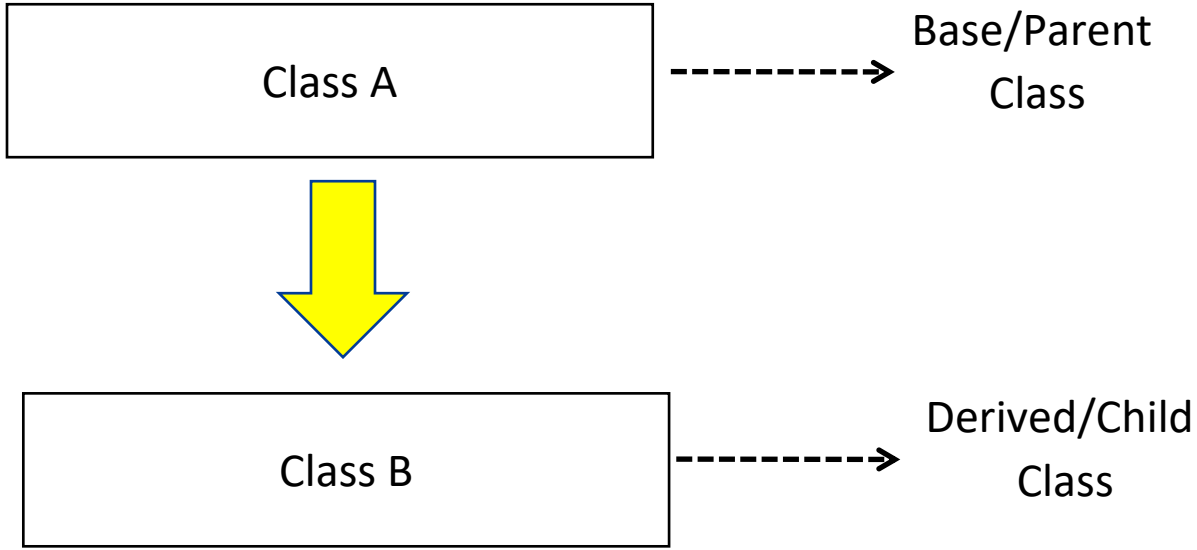
The benefit of inheritance in OOP is reusability. Once a behaviour (method) is defined in a super class, that behaviour is automatically inherited by all subclasses.

Write a method only once and it can be used by all subclasses:

- **Once a set of properties (fields) are defined in a super class, the same set of properties are inherited by all subclasses.**
- **A class and its children share common set of properties.**
- **A subclass only needs to implement the differences between itself and the parent.**

Inheritance

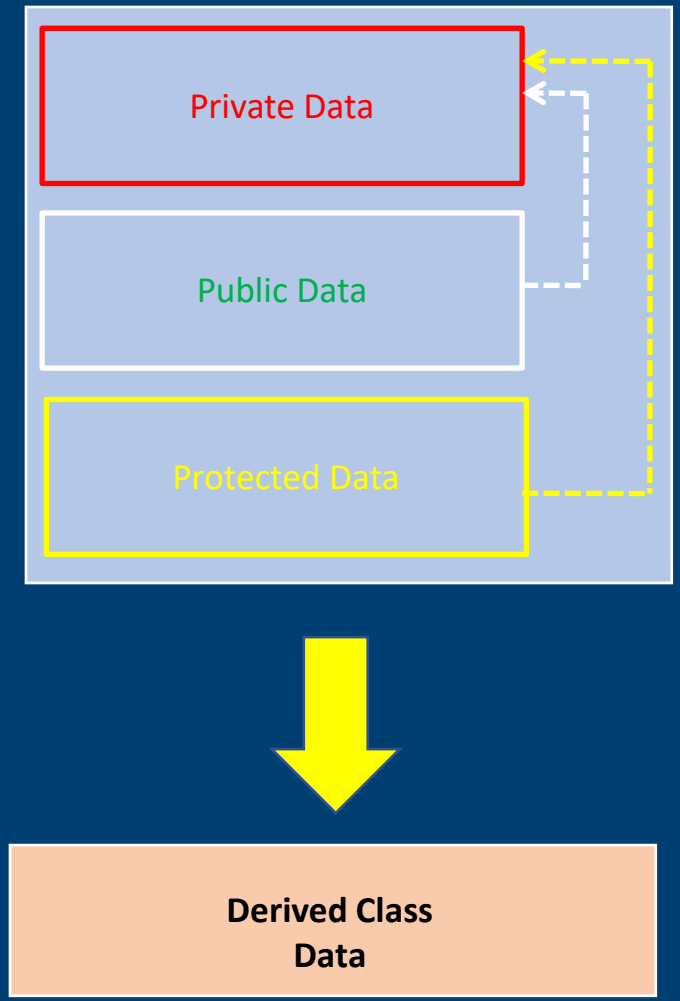
Inheritance is a mechanism of reusing or extending the functionality of one class by another class. Extended class is called "Parent/Base" class and extending class called "Child/Derived" class.

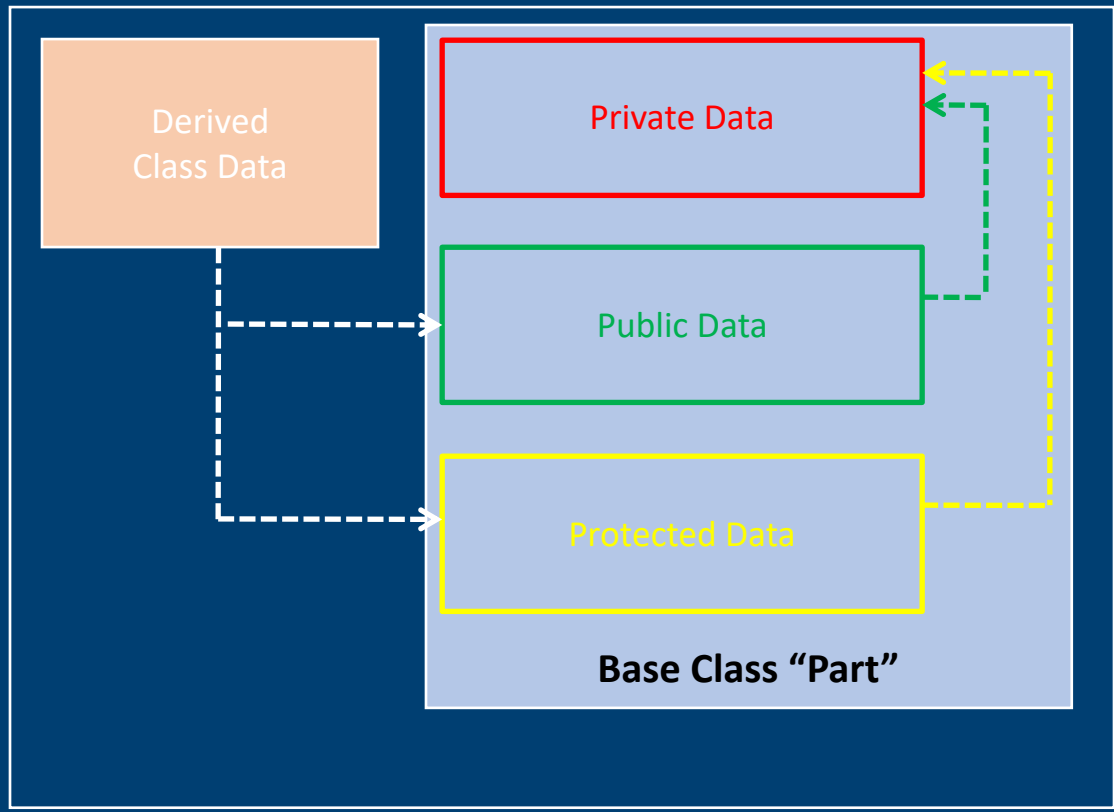


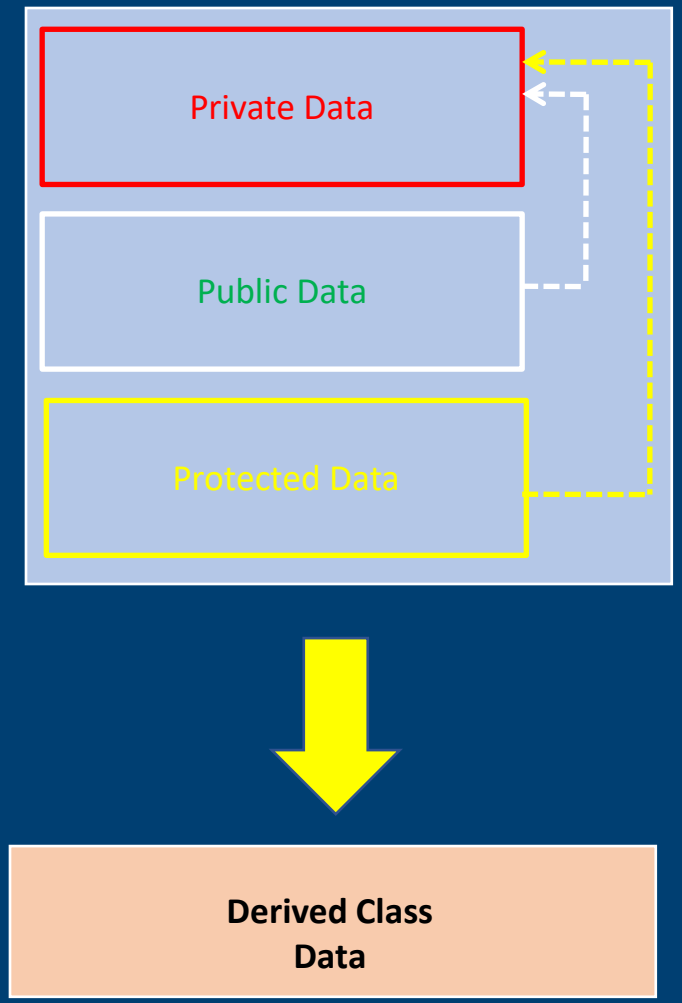
Example

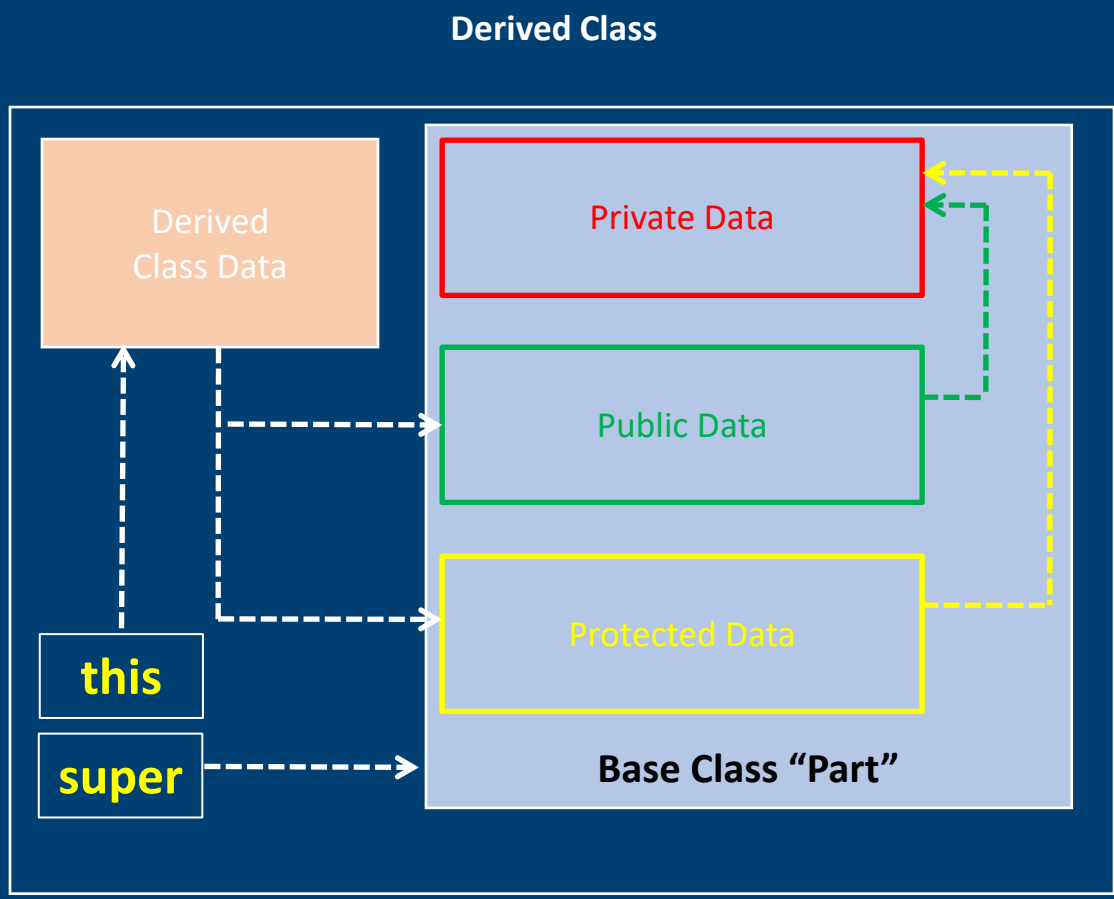
```
class A
{
   // Member variables of A.
   // Member functions of A.
}

class  B  extends  A
{
   // Member variables of B.
   // Member functions of B.
}
```

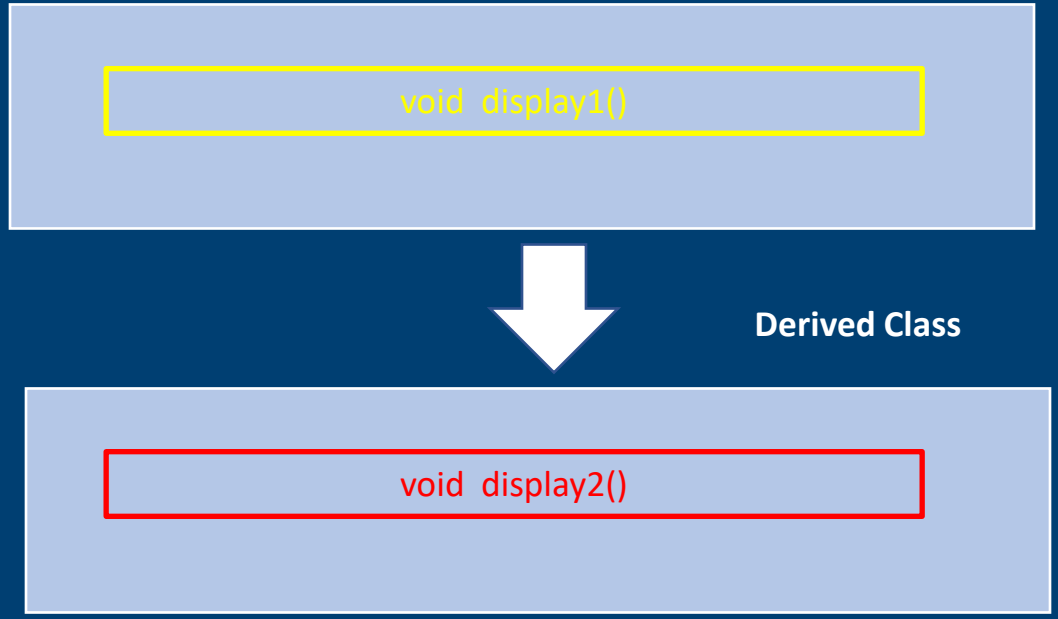
```
class B
{
   // Member variables of A
   // Member functions of A
   // Member variables of B
   // Member functions of B
}
```

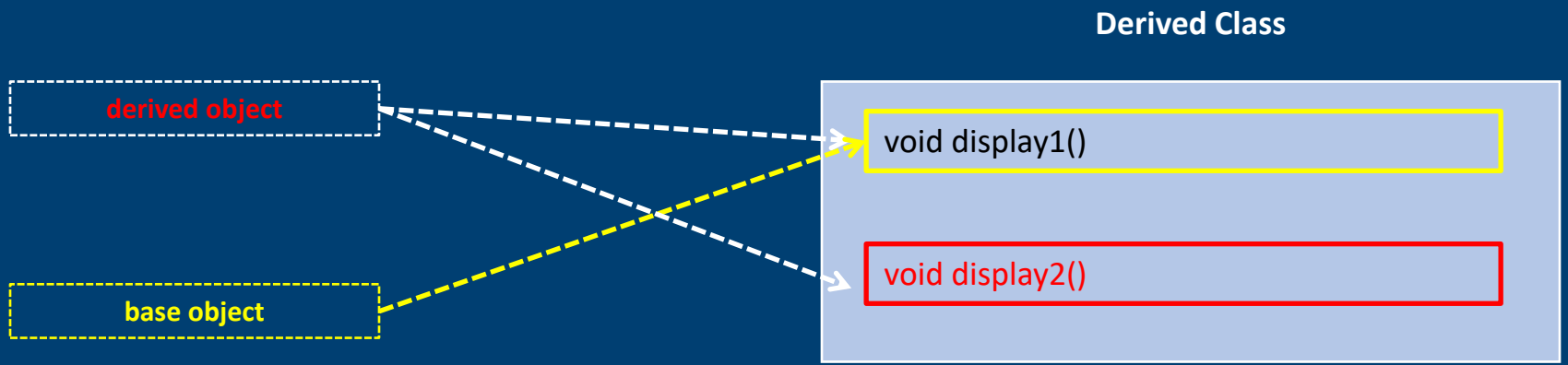

Private Data
Public Data
Protected Data
Derived Class
Data


Derived Class
Derived
Class Data
Private Data
Public Data
Protected Data
Base Class “Part”


Private Data
Public Data
Protected Data
Derived Class
Data


Derived Class
Derived
Class Data
Private Data
Public Data
Protected Data
this
super
Base Class “Part”


void display1()
Derived Class
void display2()


Derived Class
derived object
base object
void display1()
void display2()

Inheritance - concepts

1

Function Overriding

2

Preventing function overriding

3

Preventing class inheritance by another class.

4

Class slicing

Use Case in Inheritance

```
class Parent{
public void method1(){
System.out.println("Parent Method");
}
}
public static void main(String[] ar){
Parent p1=new Parent();
p1.method1();
p1.method2();
Parent p2=new Child();
p2.method1();
p2.method2();
```

```
class Child extends Parent {
public void method2(){
System.out.println("Child Method")
}
}
public static void main(String[] ar){
Child c1=new Child();
c1.method1();
c1.method2();
Child c2=new Parent();
```

Interfaces

- Multiple inheritance can be supported by using interfaces only.
- Up to java version 6.0 interface called as purest form of abstract class.
- Interface can contain final static members, abstract methods, default methods, static methods and private methods.
- Interface use to build the loosely coupled system.
- Syntax

```
interface <interface name>{
    public static final VALUE=3.14;
      public void displayArea();
    public default void getDetails(){ …….}
}
```

How to use interface

By implementing the functionality of interface by another class.

```
interface Human
getGender() {?????}
isPregnent(){?????}
```

```
class Male implements Human
getGender() {'M'}
isPregnent(){false}
```

Abstract class

Abstract class is class which is not fully defined its functionality. (Behavior).

```
abstract  class <class name>
{
   abstract <function name>([arguments]);
}
```

<u>Example:</u>

```
abstract  class  Human
{
   abstract  char getGender();  // No definition
   abstract  boolean isPregnent();   // No definition
}
```

```
Human human = new Human();
```

Error: Object can not be created from abstract class.

What is the use of Abstract class?

By extending the functionality of abstract class (base class) by another class (concrete class).

Abstract class Human

getGender() {**?????**}

isPregnent(){**?????**}

class Male extends Human

getGender() {**'M'**}

isPregnent(){**false**}

<u>Concrete Class</u>

Functionality is completely defined

Example

```
abstract  class  Human
{
   public char getGender();          // No definition
  public boolean  isPregnent();   // No definition
}


class  Male extends Human
{
   public char getGender()  { return ‘M’ }
   public boolean  isPregnent() { return false }
```

Final and Abstract

- final used to make the class instantiate.
- Final class cannot be inherited.

```
final class Person
  {
        String name;
        String number;
        ....
  }
```

- abstract used to make the class inherit.
- abstract class cannot be instantiated.

```
abstract class Person
        {
        String name;
        String number;
                ....
        }
```

Final and Abstract Implementation

```
class Customer extends Person
 {
        String name;
        String number;
        Person p=new Person();
        ....
 }
```

```
class Customer extends Person
{
String name;
String number;
Person p=new Person();

                ....
}
```

Polymorphism

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

If we have implemented Inheritance only we can use Polymorphism.

There are two types of polymorphism in Java

compile-time polymorphism	runtime polymorphism.

Compile Time Polymorphism

Overloading the method is called compile time polymorphism.

Method overloading means method having same name but different return type or different parameters.

This we can perform in same class or different class using inheritance.

Runtime Polymorphism

Overriding the method is possible only we inherit the class.

Method Overriding is not possible in same class.

Method overriding means two or more method having same name, same return type, same method parameter

If we do so, method invocation is determined by the JVM not compiler.

String, String Buffer, String Builder

Java: String class

Java String is a data type and built-in class t which is used to represent text.

1. String str = “text”; // “text” is string literal.

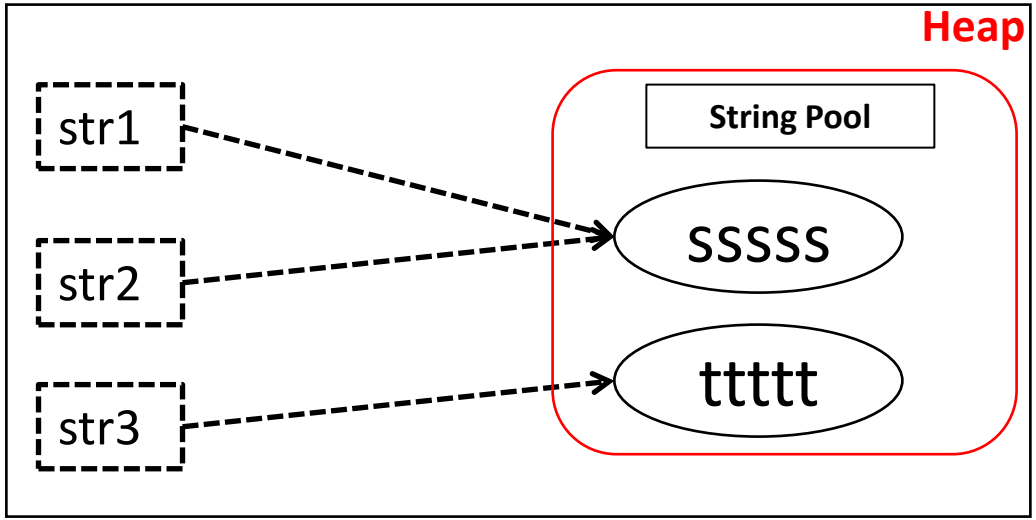
2. String str = new String(“text”);

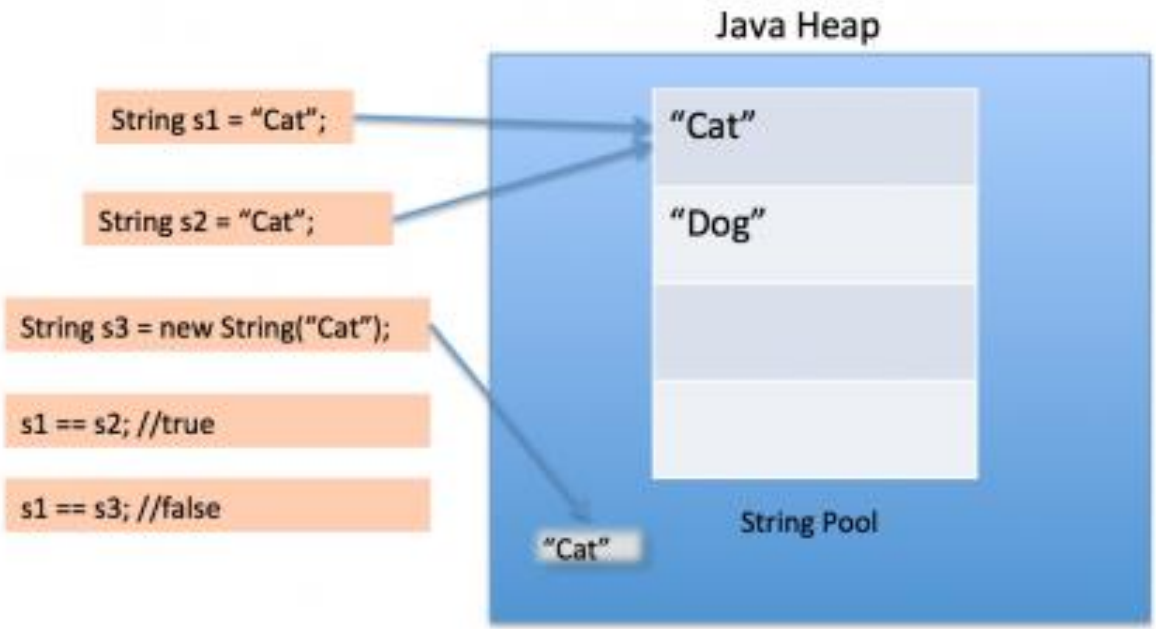
<u>String pool:</u> JVM creates a string pool with all string literals. If any two strings having same string literals then both points to same location.

String str1 = “sssss”;

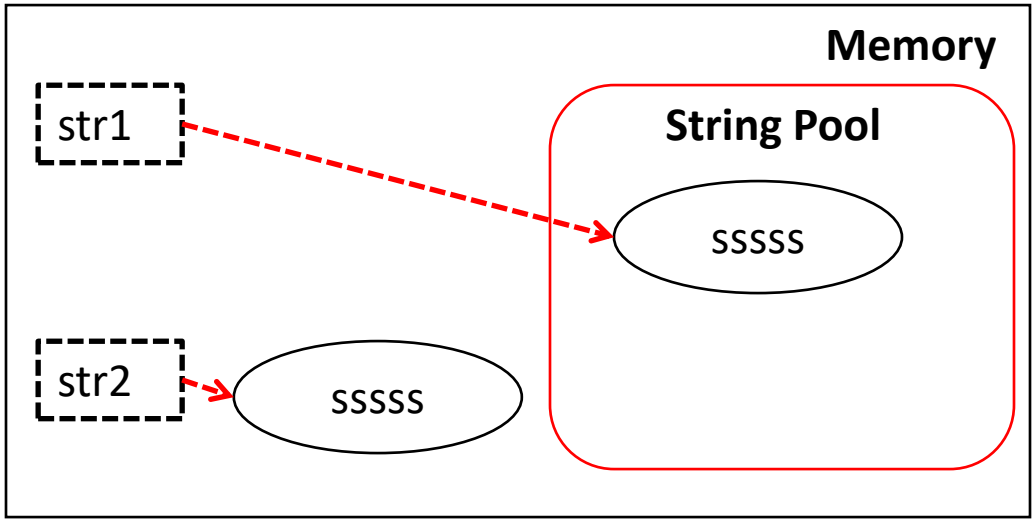
String str2 = “sssss”;

String str3 = “ttttt”;



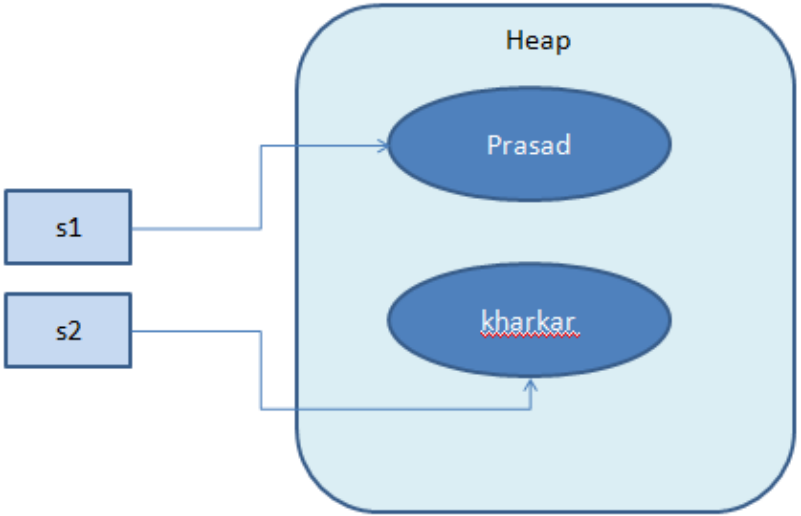

Java Heap
String s1 = "Cat";
String s2 = "Cat";
String s3 = new String("Cat");
s1 == s2; //true
s1 == s3; //false
"Cat"
"Dog"
String Pool
"Cat"

```
String  str1  = “sssss”;

String  str2  =  new String(“sssss”);
```

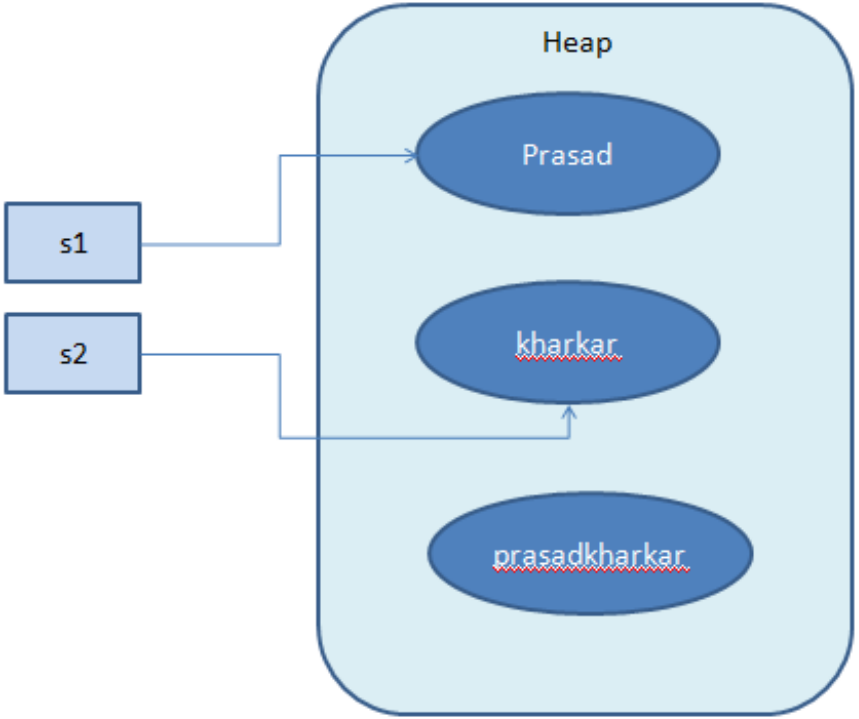


When doing any operations on string, a new object created instead of modifying the original string.

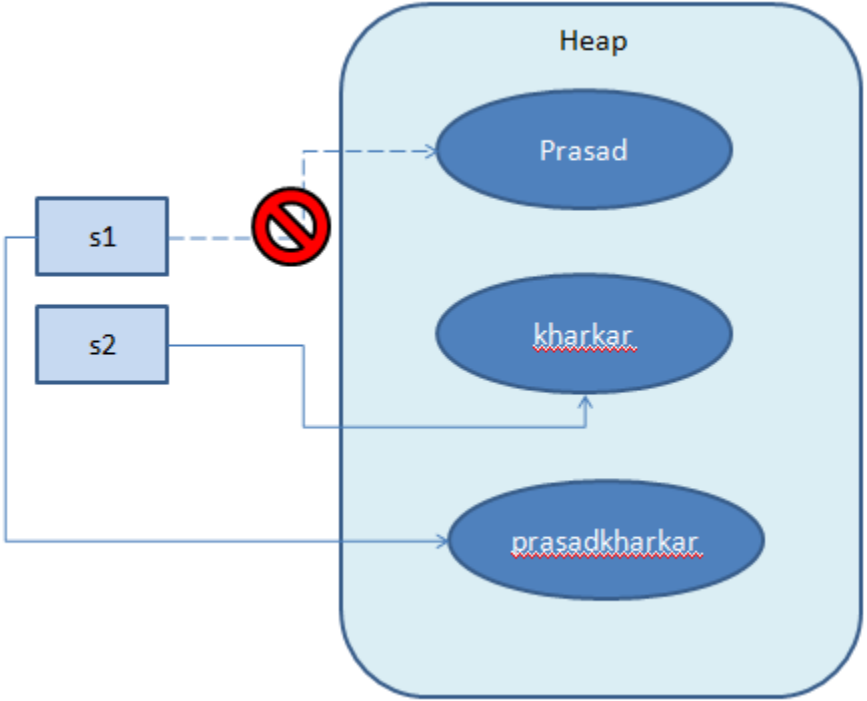
```
String s1 = new String("prasad");
String s2 = new String("kharkar");
```



s1.concat(s2)



s1 = s1.concat(s2)



Java: String functions

Function
length()
charAt()
equals()
compareTo()
concat()
getBytes()
indexOf()
replace()
trim()

Java: String Buffer

StringBuffer allows to append, replace, reverse, concatenate or any kind of manipulation on char data. It is mutable class means the contents can be updated.

Difference between String and StringBuffer is that, String represents fixed-length immutable characters where as StringBuffer represents growable and writable characters.

```
StringBuffer   strBuff  =  new StringBuffer();

StringBuffer   strBuff  =  new StringBuffer(<length>);

StringBuffer   strBuff  =  new StringBuffer(<data>);
```

Java: StringBuffer functions

Function
length()
capacity()
append()
insert(index, data)
reverse()
….

Java: String Builder

StringBuilder allows to append, replace, reverse, concatenate or any kind of manipulation on characters. It is mutable class means the contents can be updated.

Difference between StringBuilder and StringBuffer is that, both are same in all the excepts except that StringBuilder is not thread-safe.

```
StringBuilder  strBuff  =  new StringBuilder();

StringBuilder   strBuff  =  new StringBuilder(<capacity>);

StringBuilder   strBuff  =  new StringBuilder(String str);
```

Java: StringBuilder functions

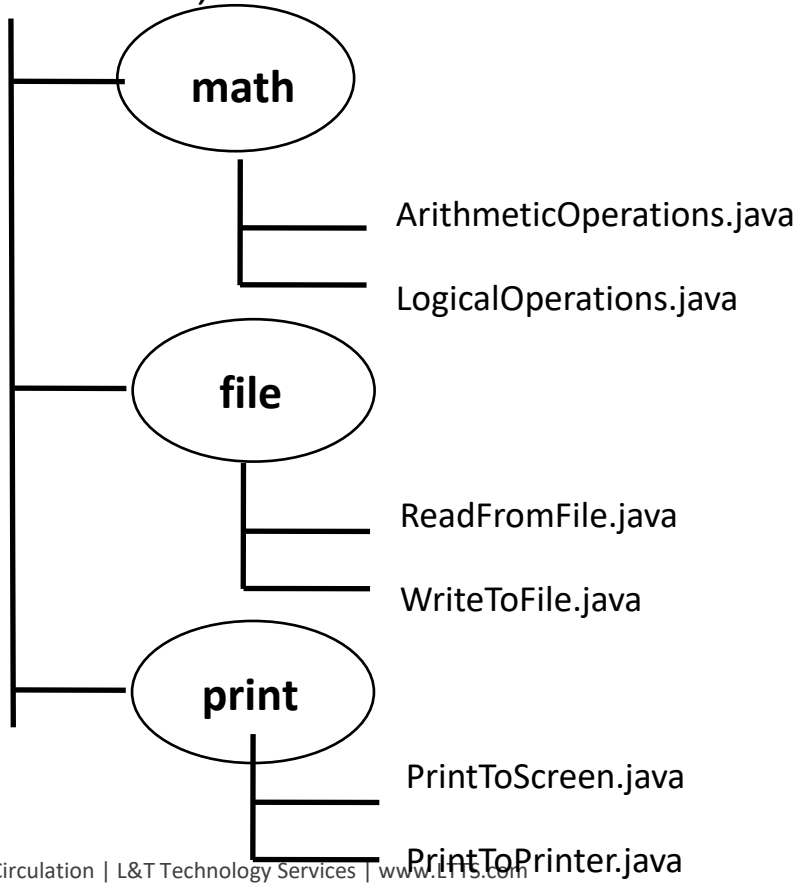
Function
length()
capacity()
append()
charAt(index)
reverse()
….

Packages

Packages and Sub Packages

A package in Java is a group of similar classes, interfaces etc.

List of Java Files
ArithmeticOperations.java
LogicalOperations.java
ReadFromFile.java
WriteToFile.java
PrintToScreen.java
PrintToPrinter.java



Importing classes from packages

Syntax:

```
import <package name>.<class name>
```

Example:

```
import math.ArithmeticOperations;
import file.WriteToFile;
import print.PrintToScreen;
```

Exception Handling

Exception Handling

Exception is an error which occurs at runtime.

Examples:

- √ Accessing objects without creating it.
- √ Accessing element array exceeding the limit.
- √ User entered invalid information.
- √ File not found during execution.

etc…..

If the exception is not handled properly in the program, then program will exit/terminate automatically.

Exception Types

Checked Exceptions

Exceptions which is identified by compiler during compile time.

- Calling function on null objects.
- Calling functions which may raise exception.

etc

Unchecked Exceptions

Exceptions which is raised during execution.

- Accessing array with invalid index.
- File missing during execution to read the data.

etc

Example

```
static void main(String   args[])
 {
     int a,b,c;
     a = 10;
     b = 0;
     System.out.println("Before ...");
     c = a/b;        ---------------------->  Program will terminate here.
     System.out.println("C=" + c);
 }
```

Syntax:

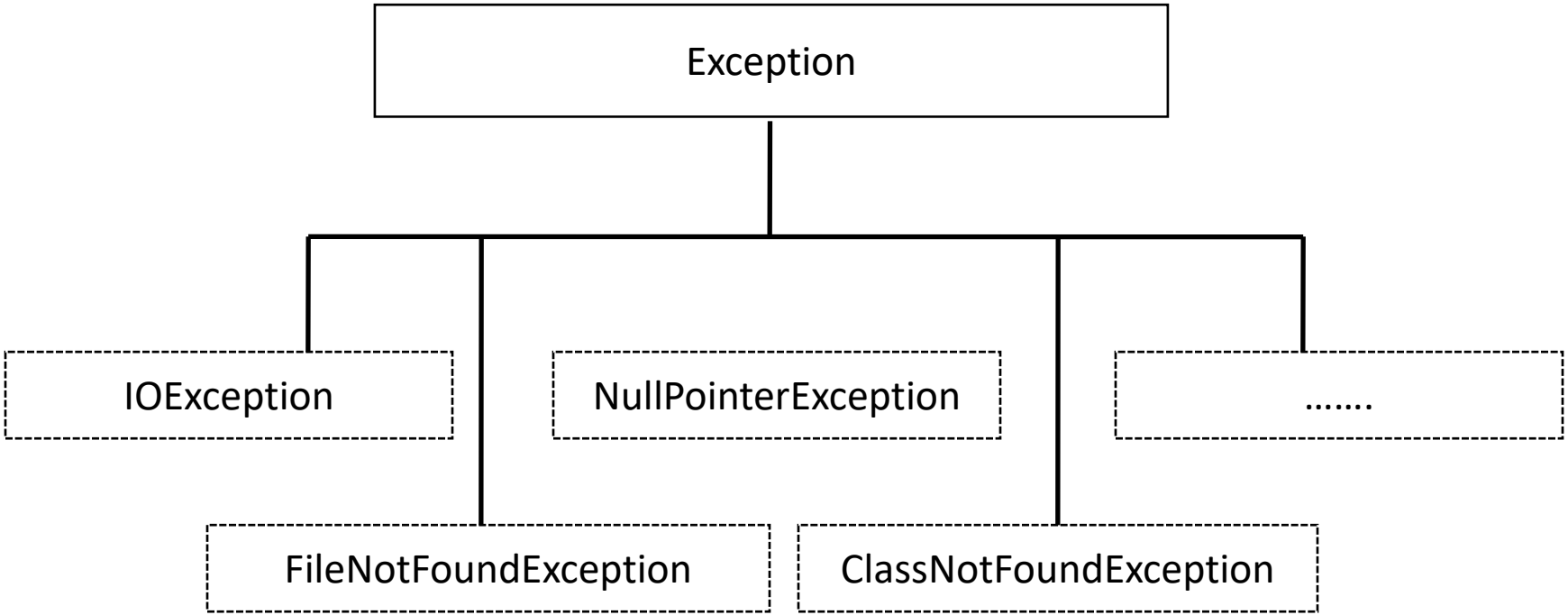
```
try
{
    statement(s);
}
catch(ExceptionType e)
{
   <exception handling statements>;
}
```

```
static void main(String   args[])
 {
    int a = 10 , b = 0, c;
   try
    {
       System.out.println("Before ...");
       c = a/b;
       System.out.println("C=" + c);
    }
    catch(ArithmeticException e)
    {
        System.out.println("Exception:" + e.getMessage());
    }
    System.out.println("After try/catch block");
 }
```

Not executed

Exception Types
ArithmeticException
ArrayIndexOutOfBoundsException
ClassNotFoundException
FileNotFoundException
IOException
NullPointerException
many more …….

Exception Hierarchy

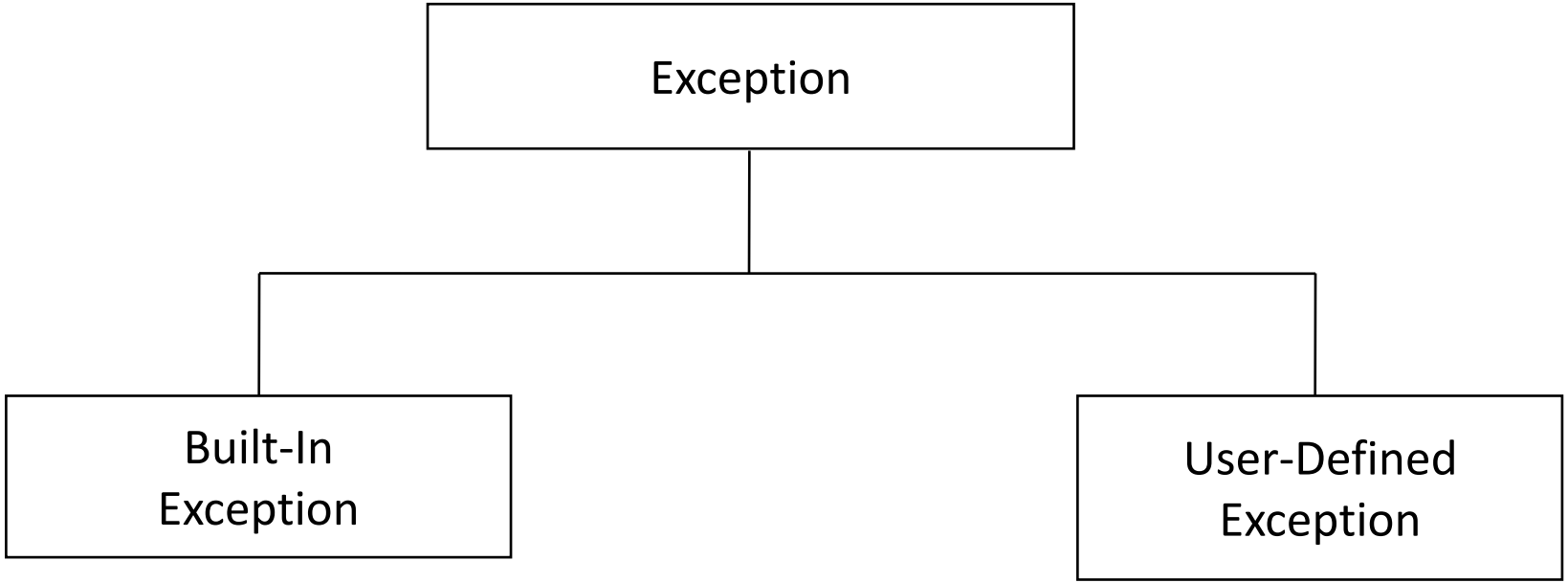


All exceptions types are sub-classes of (Derived from) Exception (Base class)

Multiple Exceptions

```
try
{
  // statement(s)
}
catch(ExceptionType-1    e)
{
   ........
}
catch(ExceptionType-2   e)
{
   <exception handling statements>;
}
catch(ExceptionType-N   e)
{
   <exception handling statements>;
}
```

Exception Hierarchy



User define exceptions

Syntax:

```
class <class_name> extends Exception
{
   // override getMessage() method.
}
```

Example:

```
class  MyException extends Exception
{
   public Sting getMessage()
   {
      // return error message.
   }
}
```

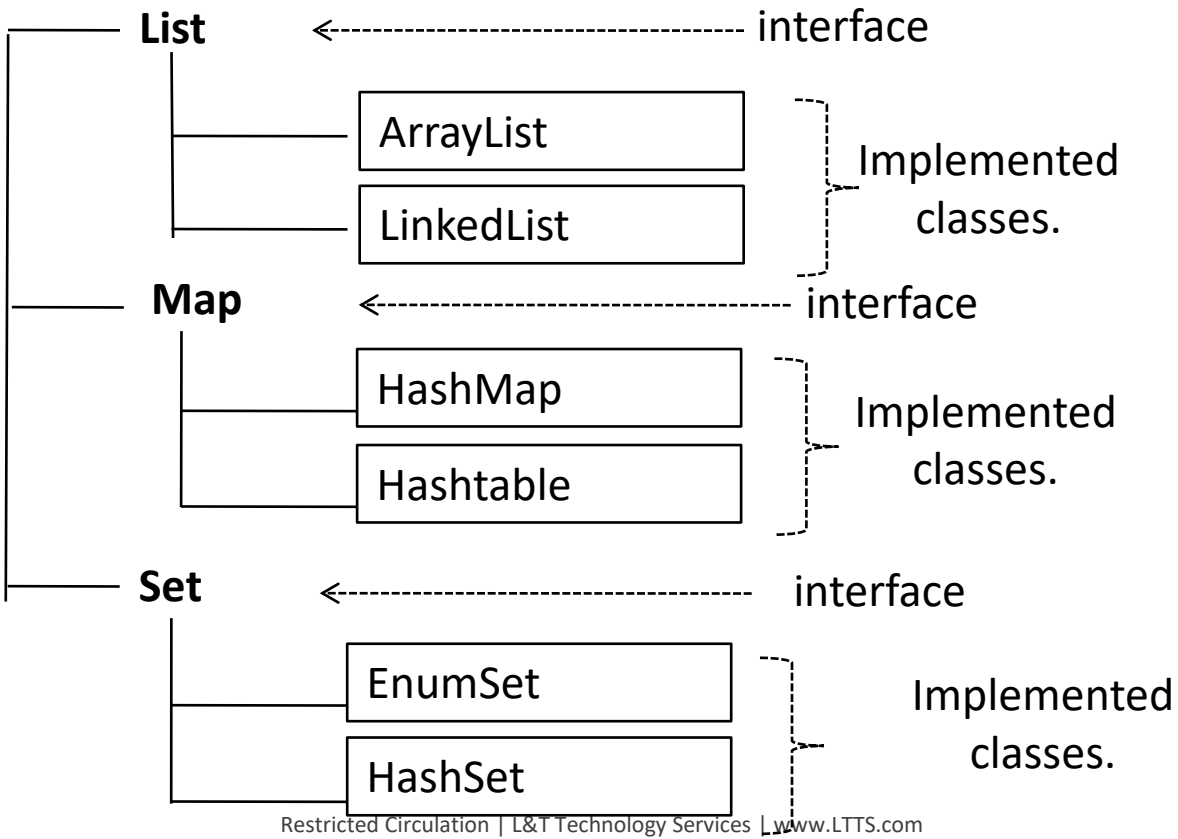
Throwing exceptions

```
try
{
    if(b == 0)
    {
      MyException myExp = new MyException();
      throw myExp;
    }
}
catch(MyException    e)
{
   System.out.println(e.getMessage());
}
```

Java Collections and Generics

Java Collections

Java collections are set of Java classes that allows to group the objects and manage them.



Java Collections - ArrayList

Syntax:

```
ArrrayList <type>  = new ArrayList<type>();
```

Example:

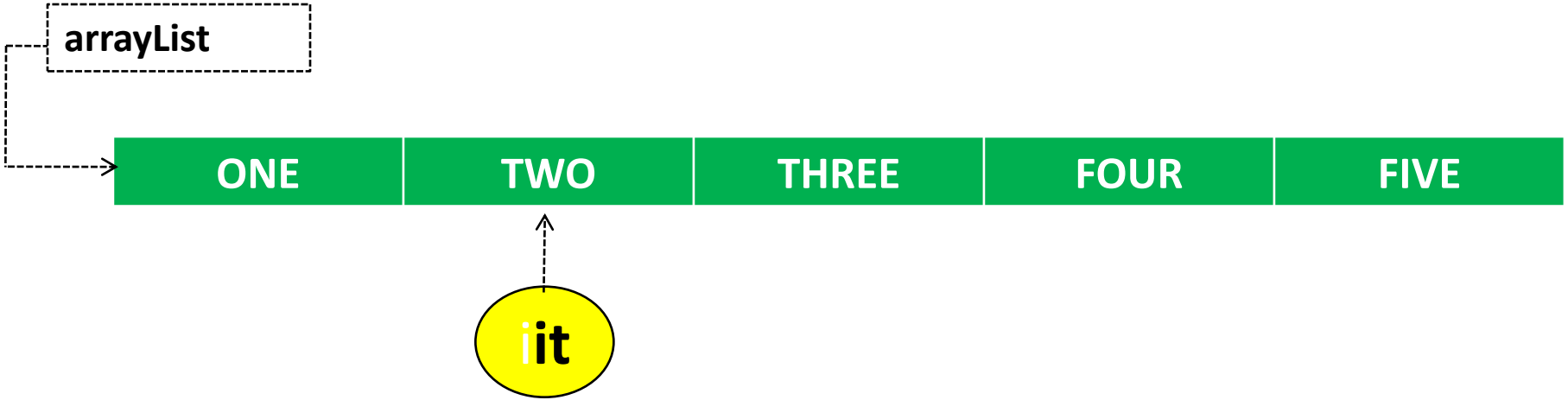
```
ArrrayList  <String> arrayList = new ArrayList<String>();
```

Function	
add(element)	add(index, element)
size()	remove(index)
get()	remove(element)
set()	clear

Concept of Iterator


```
Iterator  it  = arrayList.iterator();

it.next();          Returns  "ONE" and moves to next element.
```

Concept of Iterator

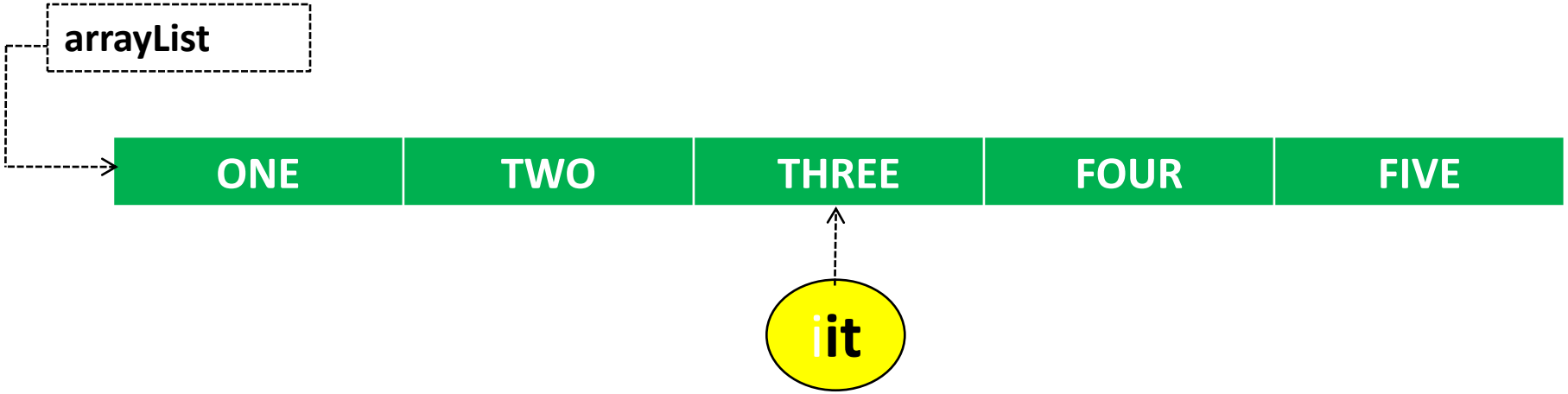



```
Iterator  it  = arrayList.iterator();
```

it.next(); Returns "ONE" and moves to next element.

it.next(); Returns "TWO" and moves to next element.

Concept of Iterator




```
Iterator  it  = arrayList.iterator();
```

it.next(); Returns "ONE" and moves to next element.

it.next(); Returns "TWO" and moves to next element.

User of Iterator

- Iterate (traverse) over the collection (List, Set, Maps).
- Remove the elements from collection.

```
ArrayList <String> list = new ArrayList <String>();
....
Iterator it = list.iterator();
while(it.hasNext())
{
  String temp = (String) it.next();
  if(temp.equals("NEW2"))
    it.remove(); // remove element using iterator.
}
```

Java Collections - LinkedList

Syntax:

```
LinkedList <type>  = new LinkedList <type>();
```

Example:

```
LinkedList  <String> myList = new LinkedList <String>();
```

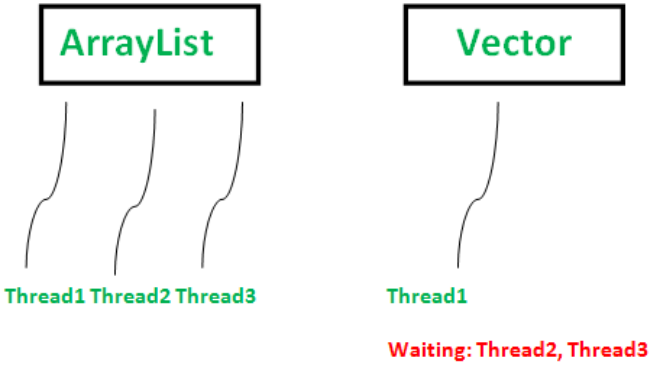
Function	
add(element)	add(index, element)
size()	remove(index)
get()	remove(element)
set()	clear

ArrayList Vs LinkedList

ArrayList	LinkedList
ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Java Collections - Vector

Java Vector class is similar to ArrayList (dynamic array) with difference is that it is synchronized (Thread-safe).



<u>Syntax:</u>

```
Vector <type>  = new Vector <type>();
```

<u>Example:</u>

```
Vector <String> myVector = new Vector <String>();
```

Java Collections - HashMap

HashMap store items in "key/value" pairs, and can access them by key value.

Key	Value
name	Badrinath
role	Faculty

- Contains values based on the key.
- Contains only unique keys.
- May have one null key and multiple null values.
- Non synchronized.
- Maintains no order.

Syntax:

```
HashMap <key, value>  =  new HashMap<key, value>();
```

Example:

```
HashMap<String, String> hm =  new HashMap<String, String>();
```

Function
put(key, value)
get(key)
remove(<element>)
containsKey()
keySet() , entrySet()

Java Collections - HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It implements Set interface.

- Contains unique elements only.
- Stores the elements by hashing.
- Allows null value.
- Non synchronized.
- No insertion order. Elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

Syntax:

```
HashSet <type>  = new HashSet <type>();
```

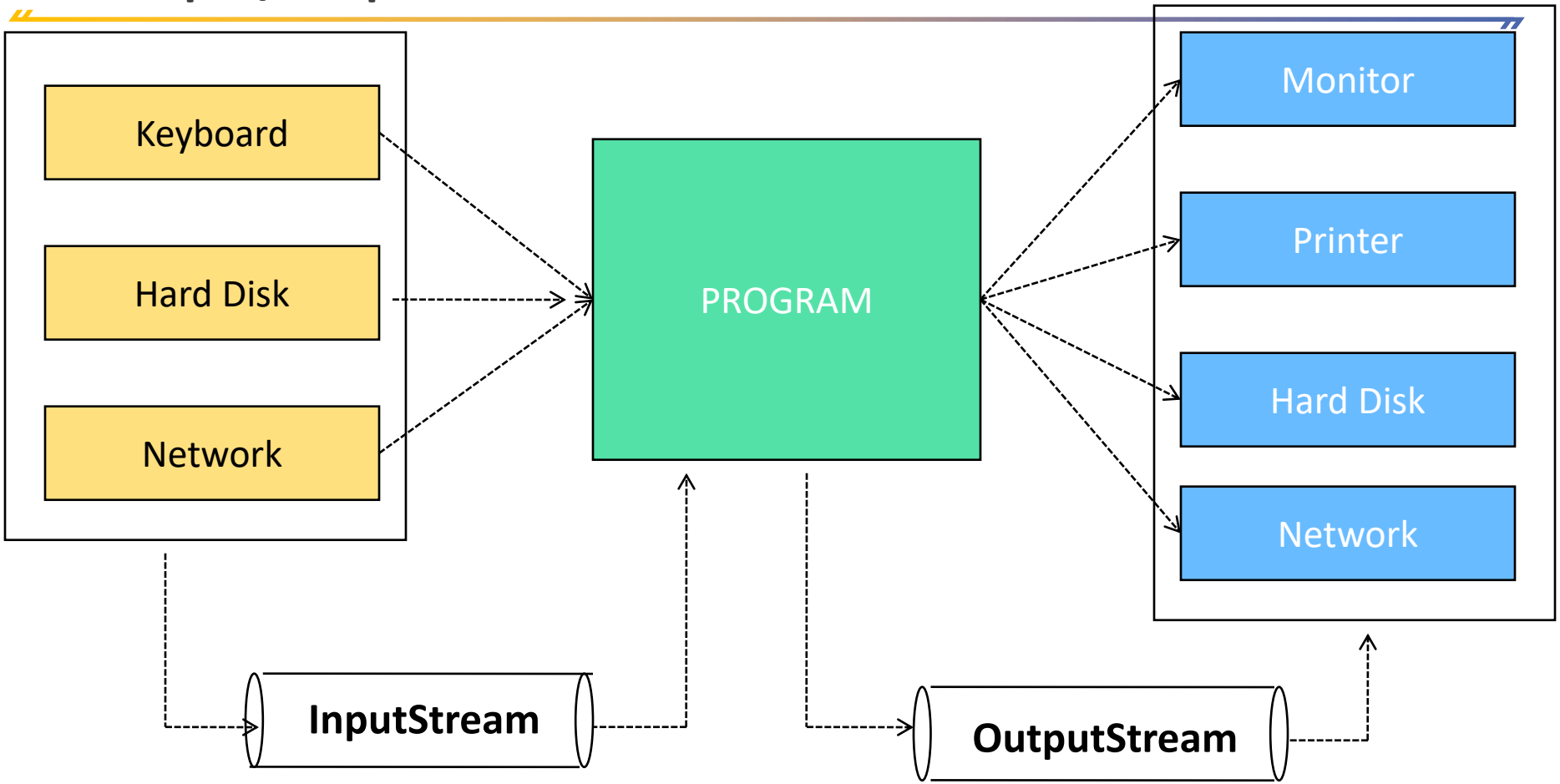
Example:

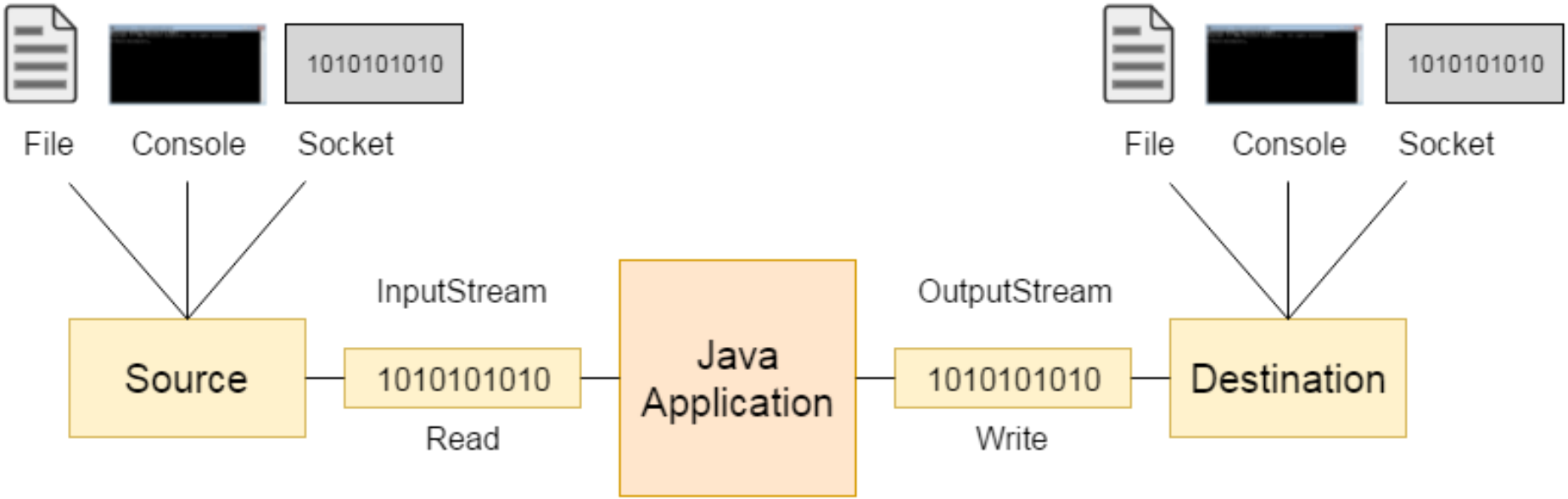
```
HashSet  <String> myList = new HashSet <String>();
```

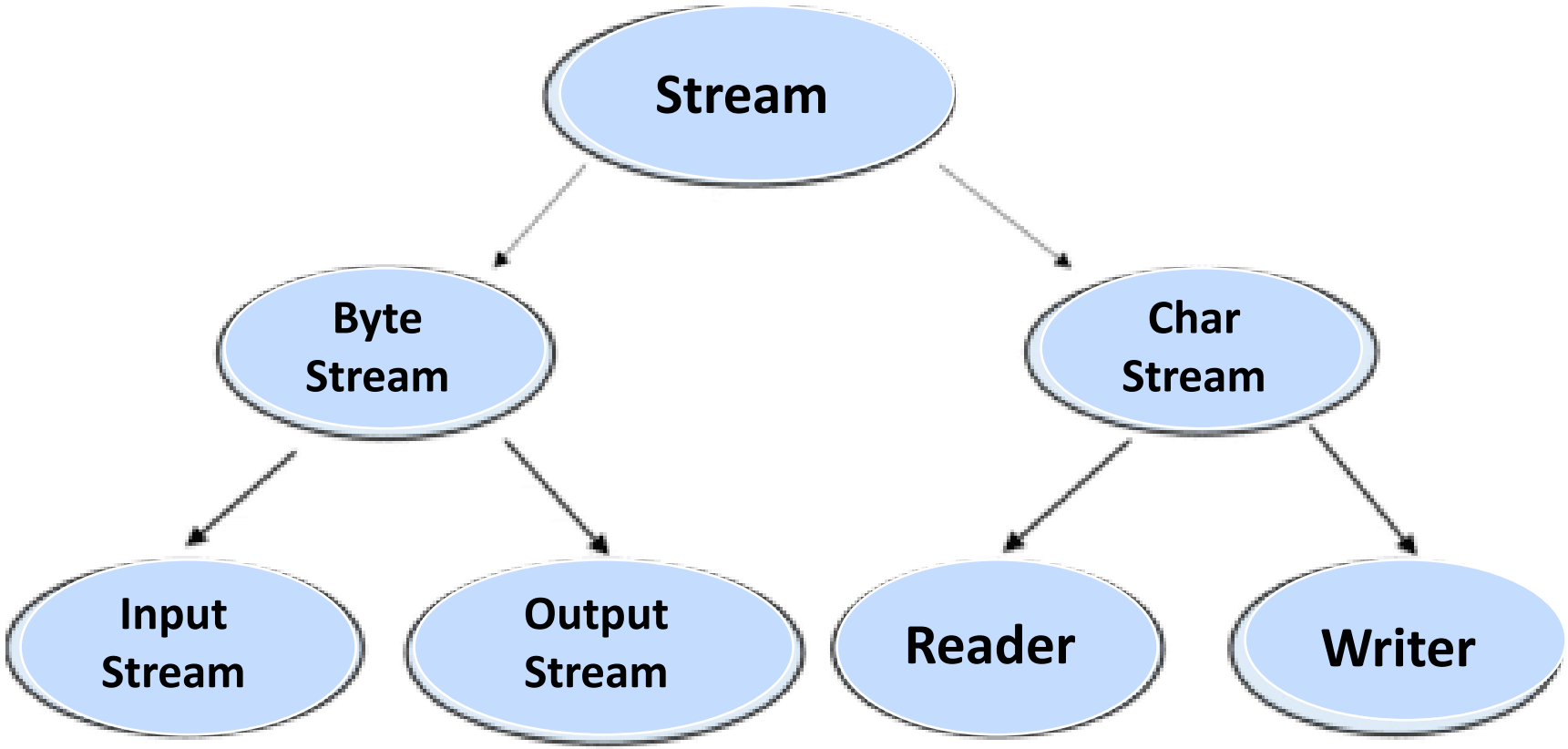
Function
add()
size()
remove(<element>)

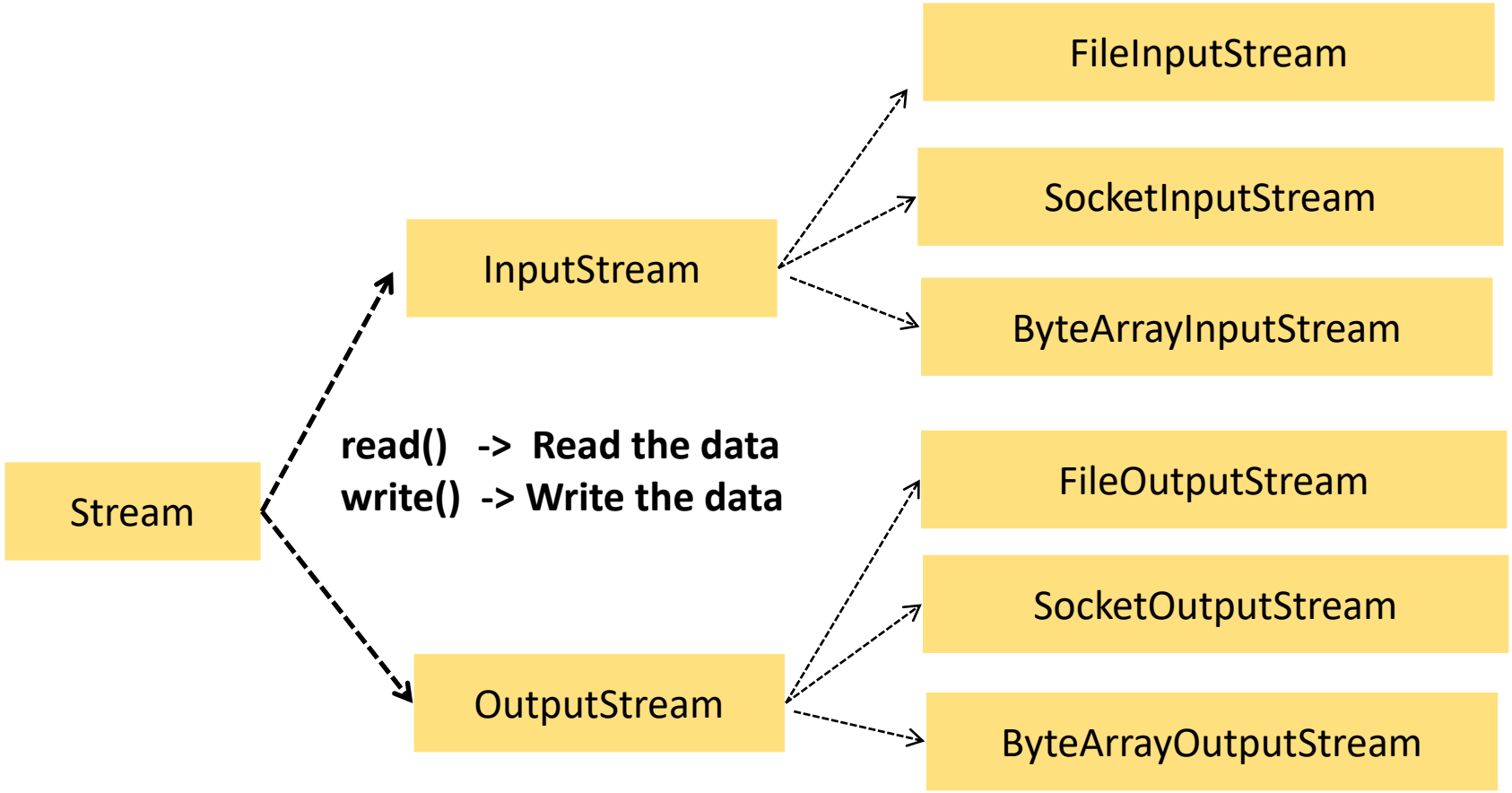
Java I/O

Java Input/Output




1010101010
File
Console
Socket
1010101010
File
Console
Socket
InputStream
OutputStream
Source
1010101010
Read
Java Application
1010101010
Write
Destination


Stream
Byte Stream
Char Stream
Input Stream
Output Stream
Reader
Writer


FileInputStream
SocketInputStream
InputStream
ByteArrayInputStream
read() -> Read the data
write() -> Write the data
Stream
FileOutputStream
SocketOutputStream
OutputStream
ByteArrayOutputStream

Read from File (FileInputStream)

FileInputStream in = new FileInputStream(<filename>);

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to **b.length** bytes of data from the input stream.
void close()	It is used to closes the stream.

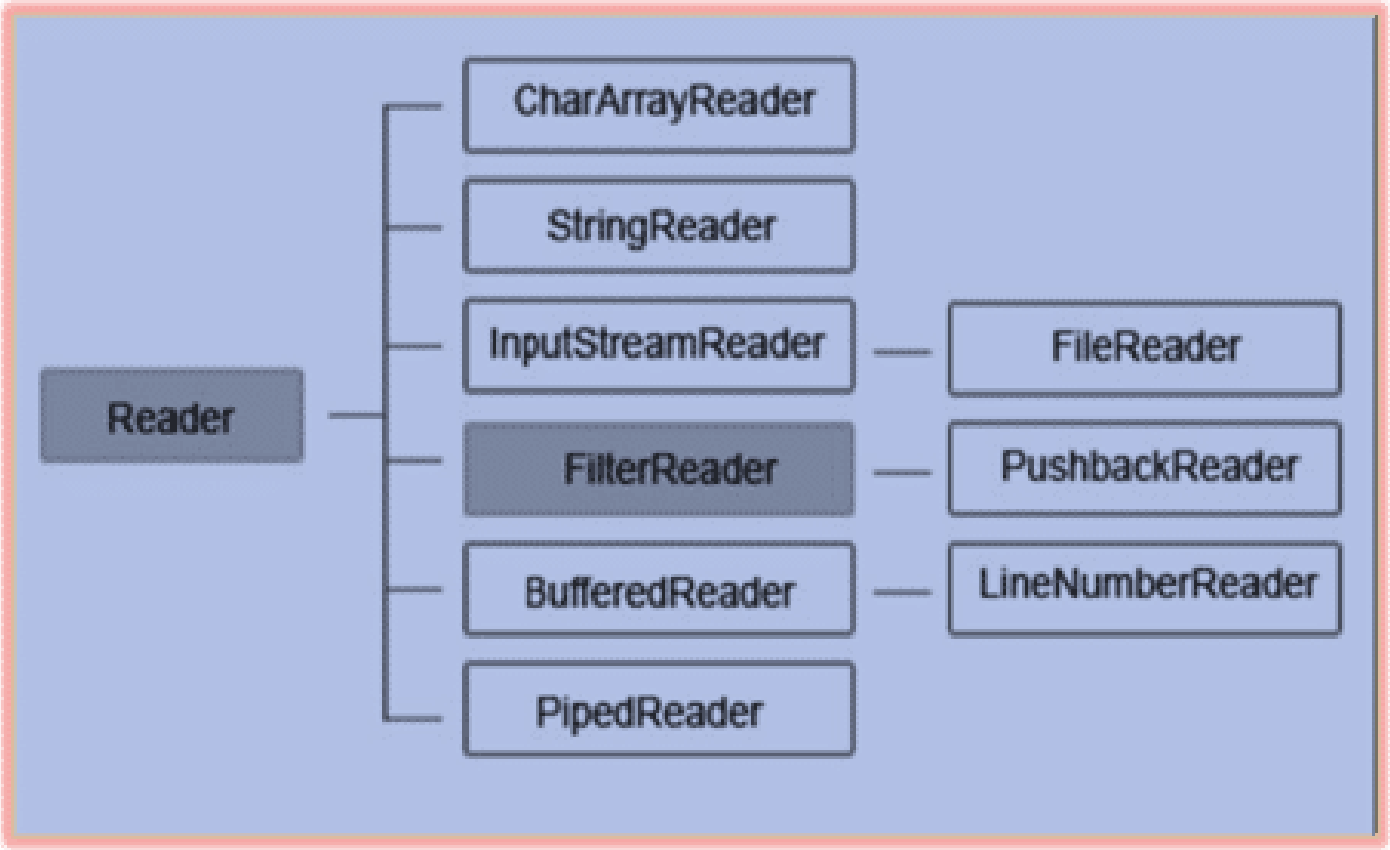
FileOutputStream in = new FileOutputStream(<filename>);

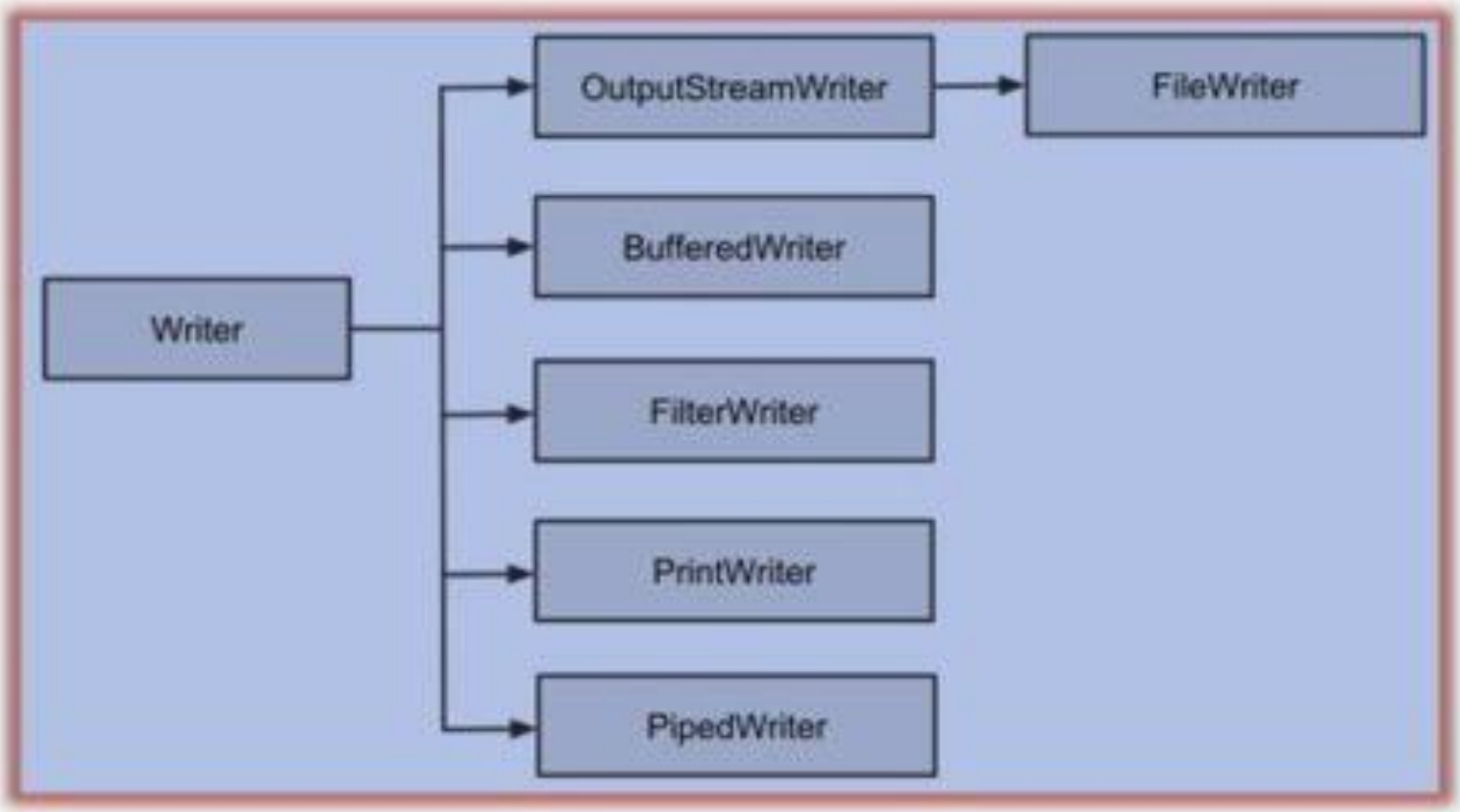
Write to File (FileOutputStream)

```
FileOutputStream  in = new FileOutputStream(<filename>);
```

Method	Description
void write(byte[] array)	It is used to write array**.length** bytes from the byte array to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
void close()	It is used to closes the file output stream.

Character Streams (Readers)




Writer
OutputStreamWriter
FileWriter
BufferedWriter
FilterWriter
PrintWriter
PipedWriter

Reading character from Keyboard (STDIN)

InputStreamReader is used to read the data from Keyboard.

Syntax:

```
InputStreamReader in = new InputStreamReader(System.in);
```

Methods:

read() : Only reads one character from the file associated with reader.

close(): Close the reader associated with file.

Reading lines from Keyboard (STDIN)

BufferedReader is used to read the data from Keyboard. It allows to read the lines from the keyboard.

Syntax:

```
BufferedReader br = new BufferedReader (<reader object>);
```

Methods:

Method	Description
int read()	It is used for reading a single character.
int read(char[] cbuf, int off, int len)	It is used for reading characters into a portion of an array.
String readLine()	It is used for reading a line of text.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

FileReader

FileReader read the data from file in character format.

Syntax:

```
FileReader reader  = new FileReader(<file name>);

FileReader reader  = new FileReader(File obj);
```

Methods:

read() : Only reads one character from the file associated with reader.

close(): Close the reader associated with file.

BufferedReader

BufferedReader read the data from ANY Reader object in character by character or entire line.

Syntax:

```
BufferedReader br  = new BufferedReader(<reader object>);

BufferedWriter bw  =  new BufferedWriter(<reader object>);
```

Method	Description
int read()	It is used for reading a single character.
String readLine()	It is used for reading a line of text.
void close()	It closes the input stream and releases any of the system resources associated with the stream.

FileWriter

FileWriter used to write the data to file character by character or by strings.

Syntax:

```
FileWriter writer = new FileWriter (<file name>);

FileWriter writer = new FileWriter (File obj);
```

Methods:

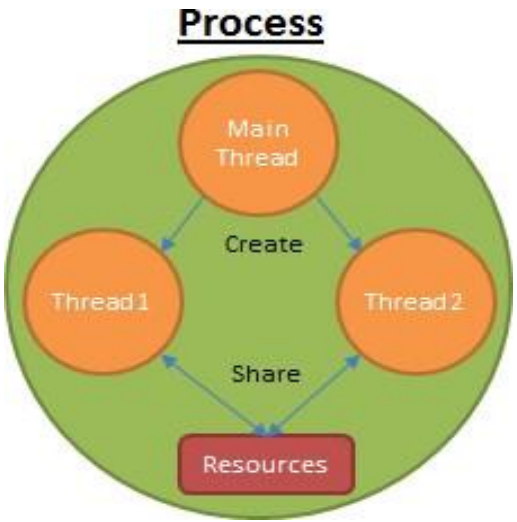
Method	Description
void write(String text)	It is used to write the string into FileWriter.
void write(char c)	It is used to write the char into FileWriter.
void flush()	It is used to flushes the data of FileWriter.
void close()	It is used to close the FileWriter.

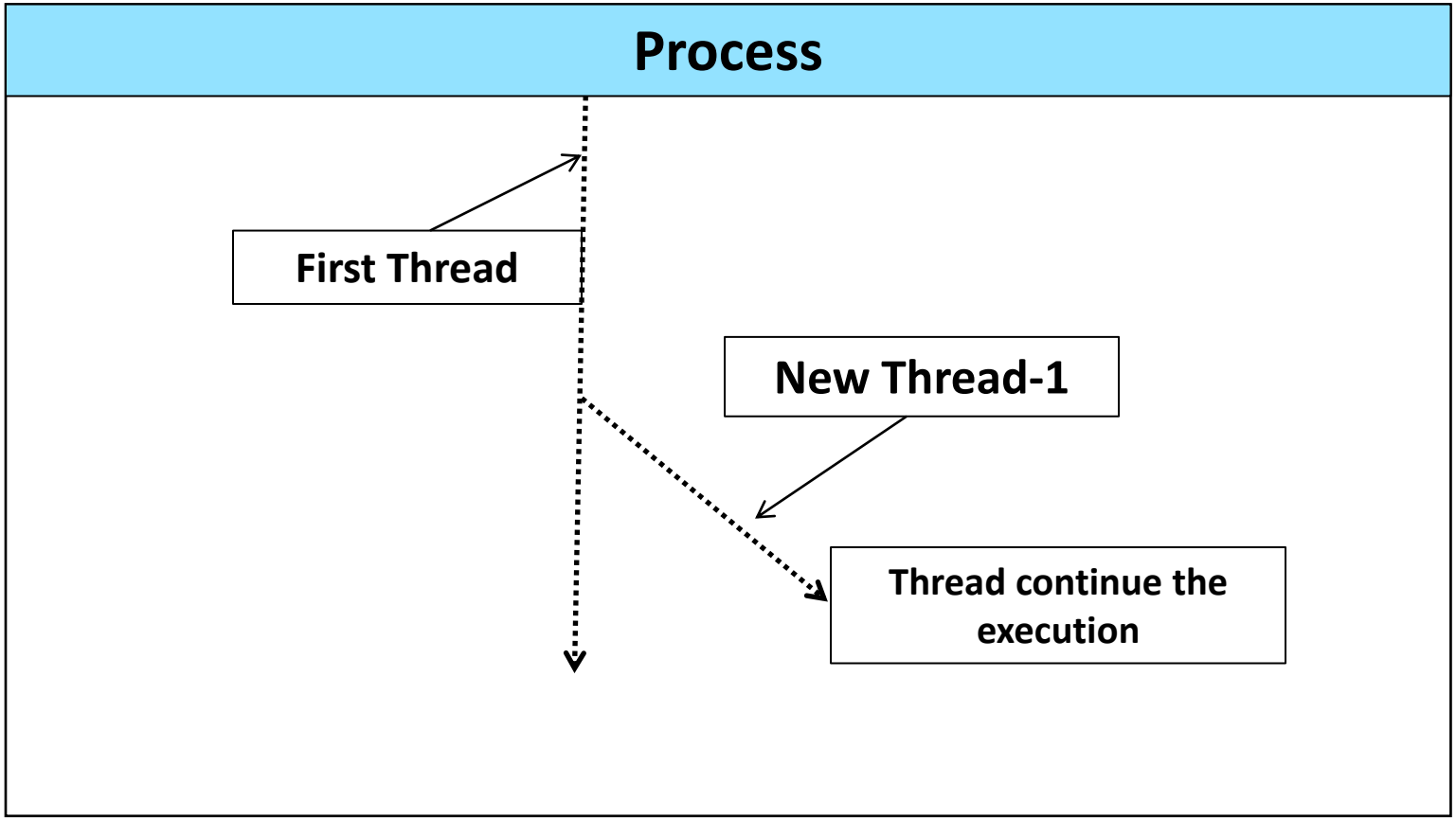
Multi-Thread Programming

Introduction to Threads

- Threads
  - A thread is a task in a process.
  - Shares resources of process.
  - Single process can have multiple threads threads.
  - Useful when an application wants to perform many concurrent tasks.
  - Example:

    Browser loading pages, animations, etc in parallel.

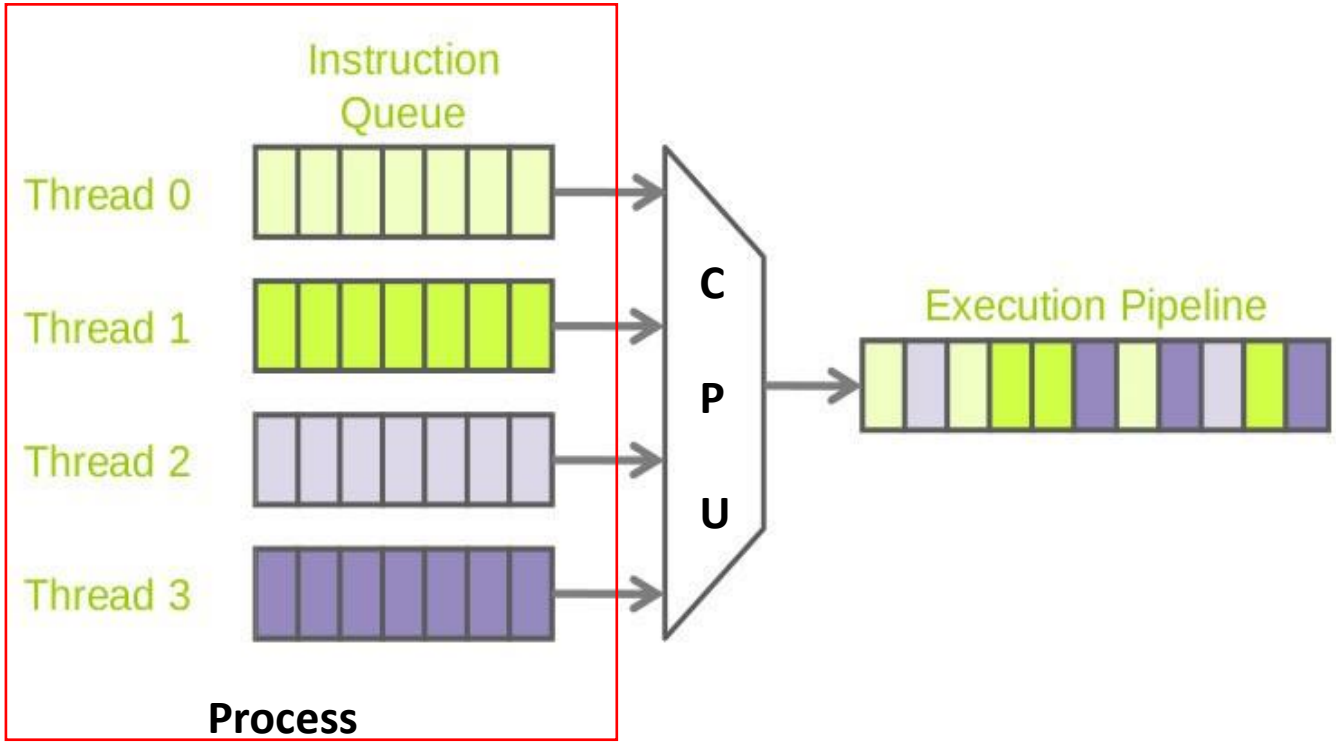



Process
First Thread
New Thread-1
Thread continue the execution

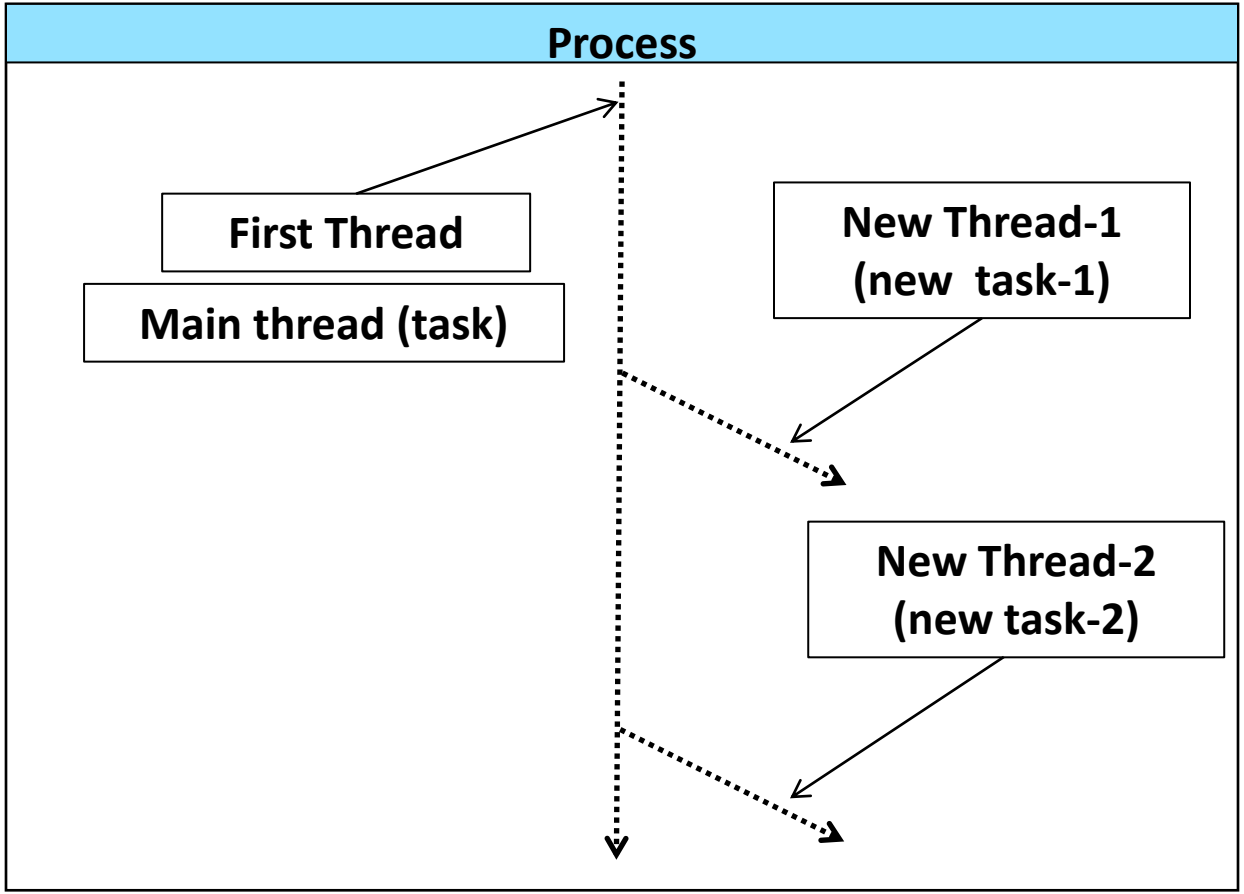
Advantages of Threads

- Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be truly concurrent.
- Allows to do something else while one thread is waiting for an I/O task (disk, network) to complete.
- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

Thread scheduling



Multithread Programming



Creating threads

Java provides two different ways to create the threads.

- Extending the “Thread” class.
- Implementing the “Runnable” interface.

Extending Thread

Example:

```
class   extends Thread
{
    public void run()
    {
        // Write the code which run as thread.
    }
}
```

Example

```
class  MyThread extends Thread
{
    public void run()
    {
        System.out.println("In MyThread….");
    }
}

public static void main(String arg[])
{
   MyThread myThread  = new MyThread();
   myThread.start();
}
```

Joining the thread

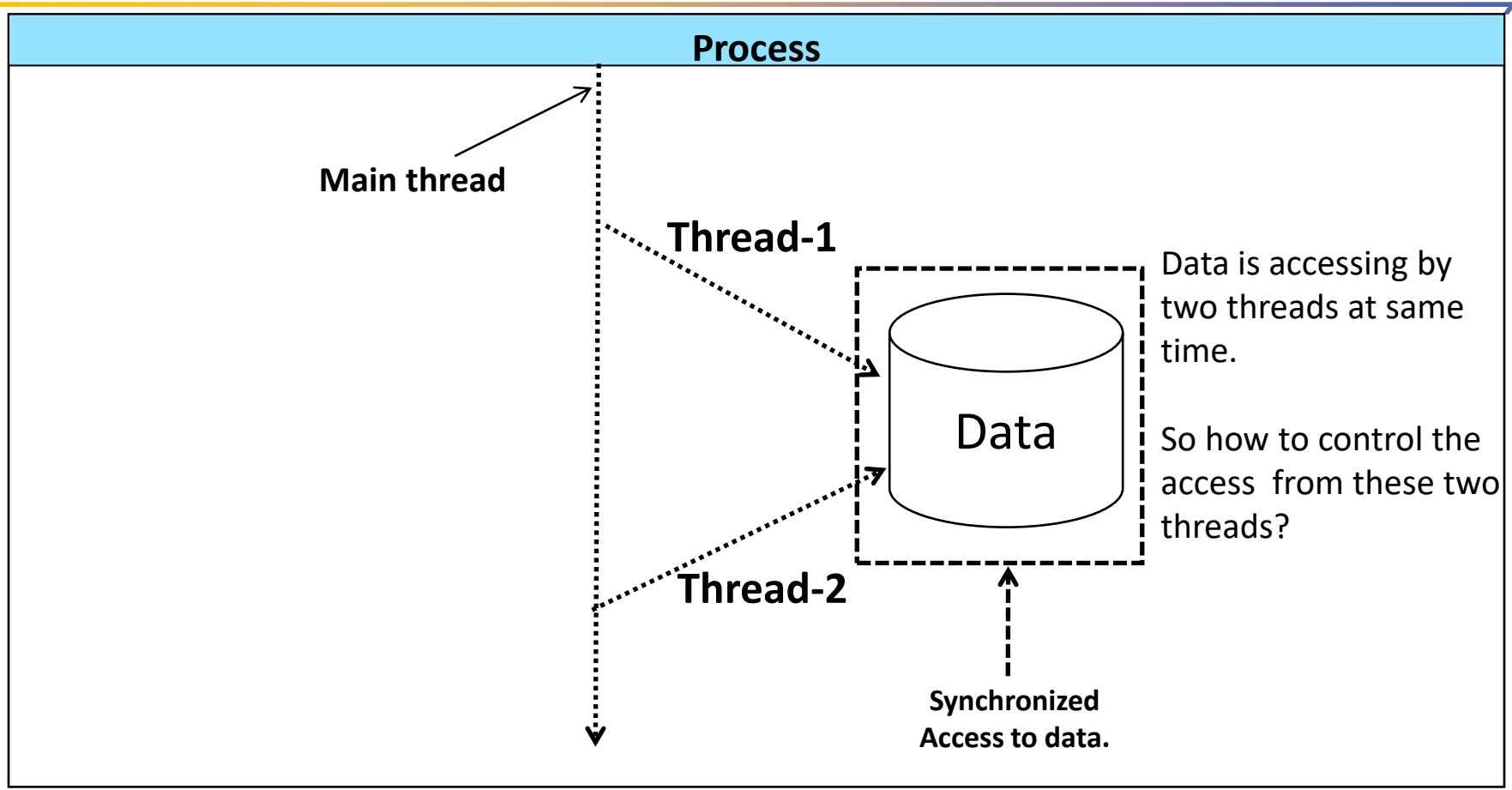
```
Public static void main(String arg[])
{
   MyThread myThread  = new MyThread();
   System.out.println("main thread started…");
   myThread.start();
   myThread.join(); // Wait main thread till myThread is over.
   System.out.println("main thread started…");
}
```

Implementing Runnable interface.

Example:

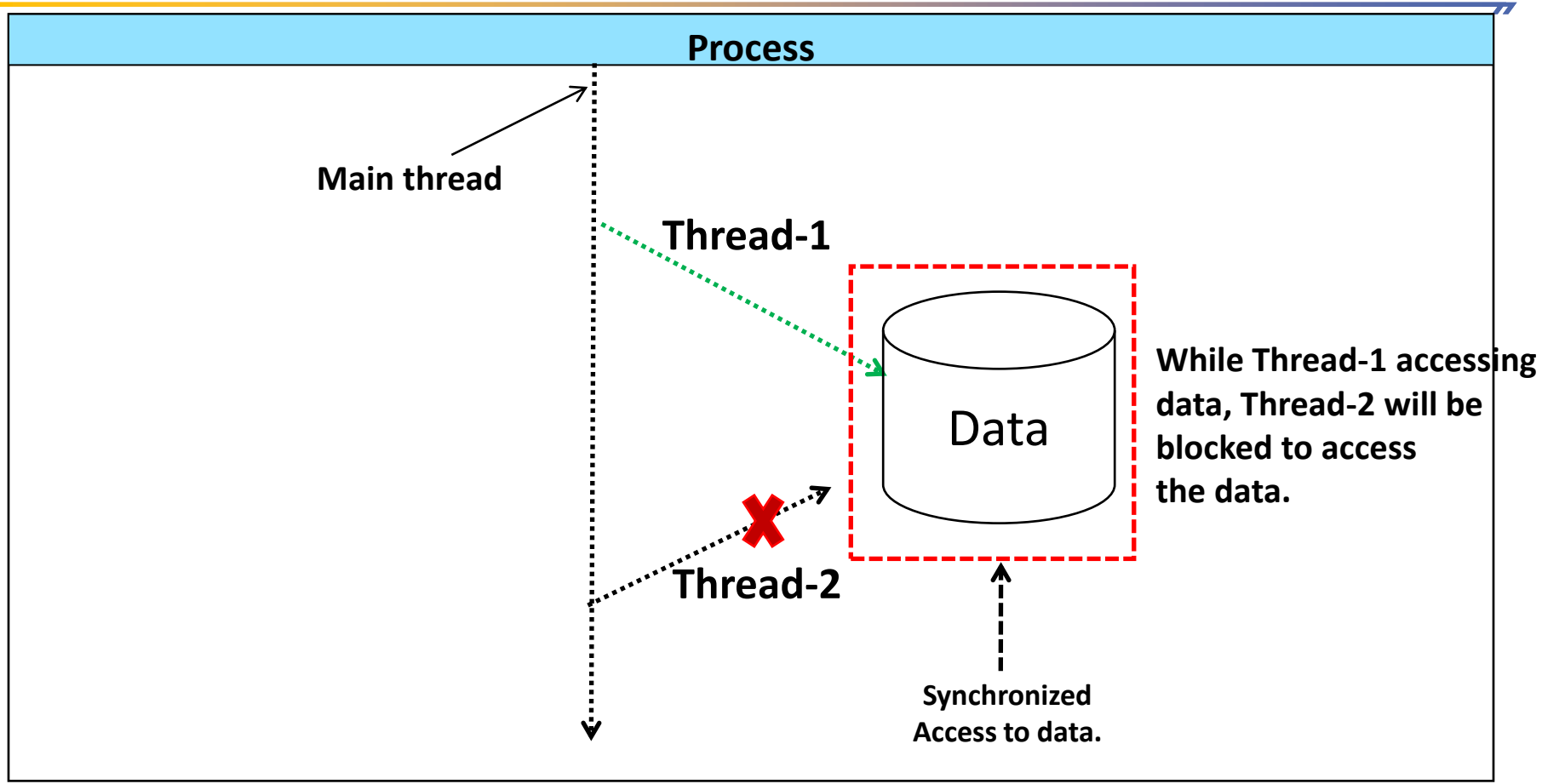
```
class  MyRunnable implements Runnable
{
    public void run()
    {
        // Write the code which run as thread.
    }
}

MyRunnable myRunnable  = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

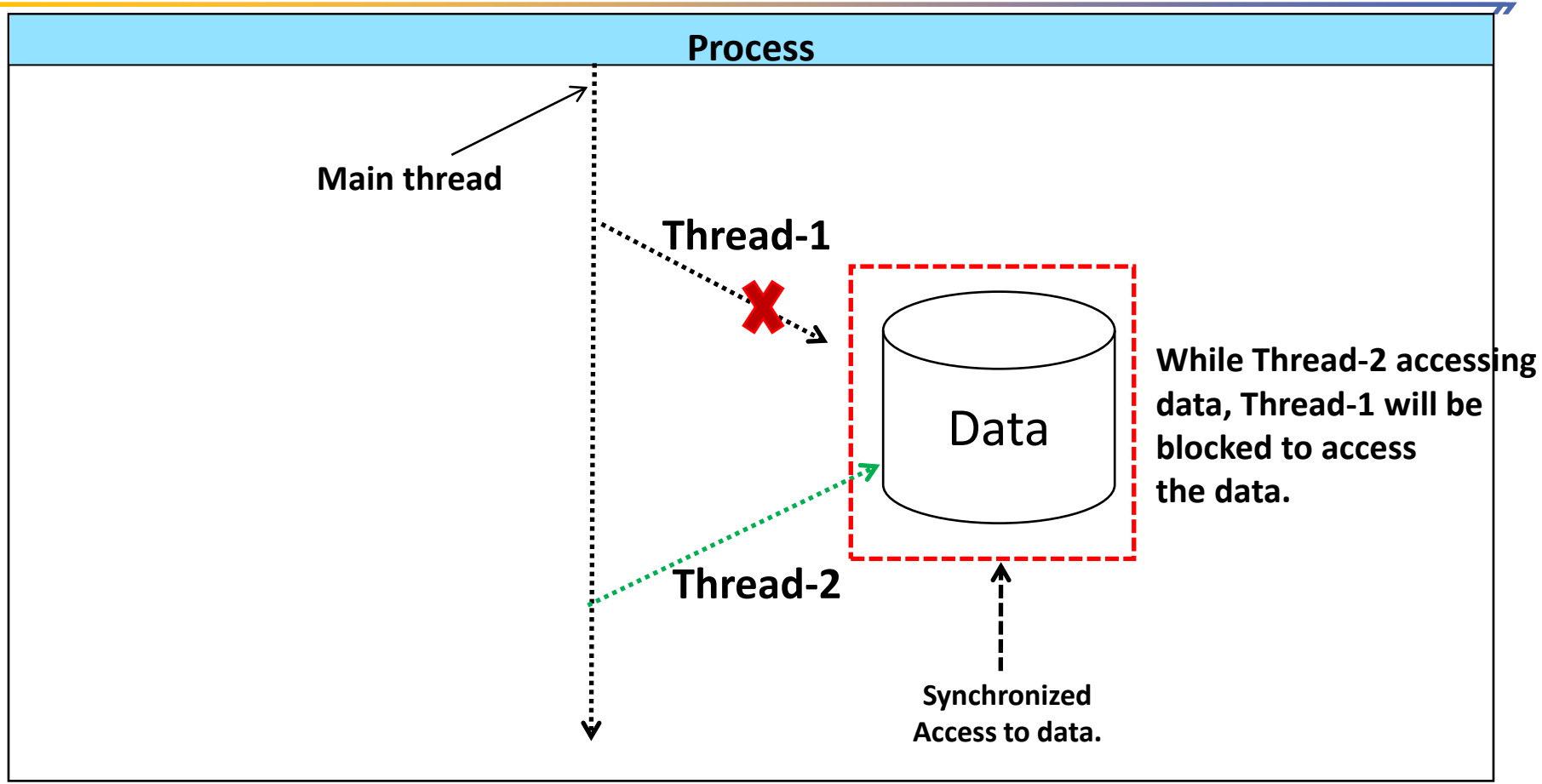
Thread synchronization



Data is accessing by two threads at same time.

So how to control the access from these two threads?


Process
Main thread
Thread-1
Data
While Thread-1 accessing data, Thread-2 will be blocked to access the data.
Thread-2
Synchronized Access to data.


Process
Main thread
Thread-1
Data
While Thread-2 accessing data, Thread-1 will be blocked to access the data.
Thread-2
Synchronized Access to data.

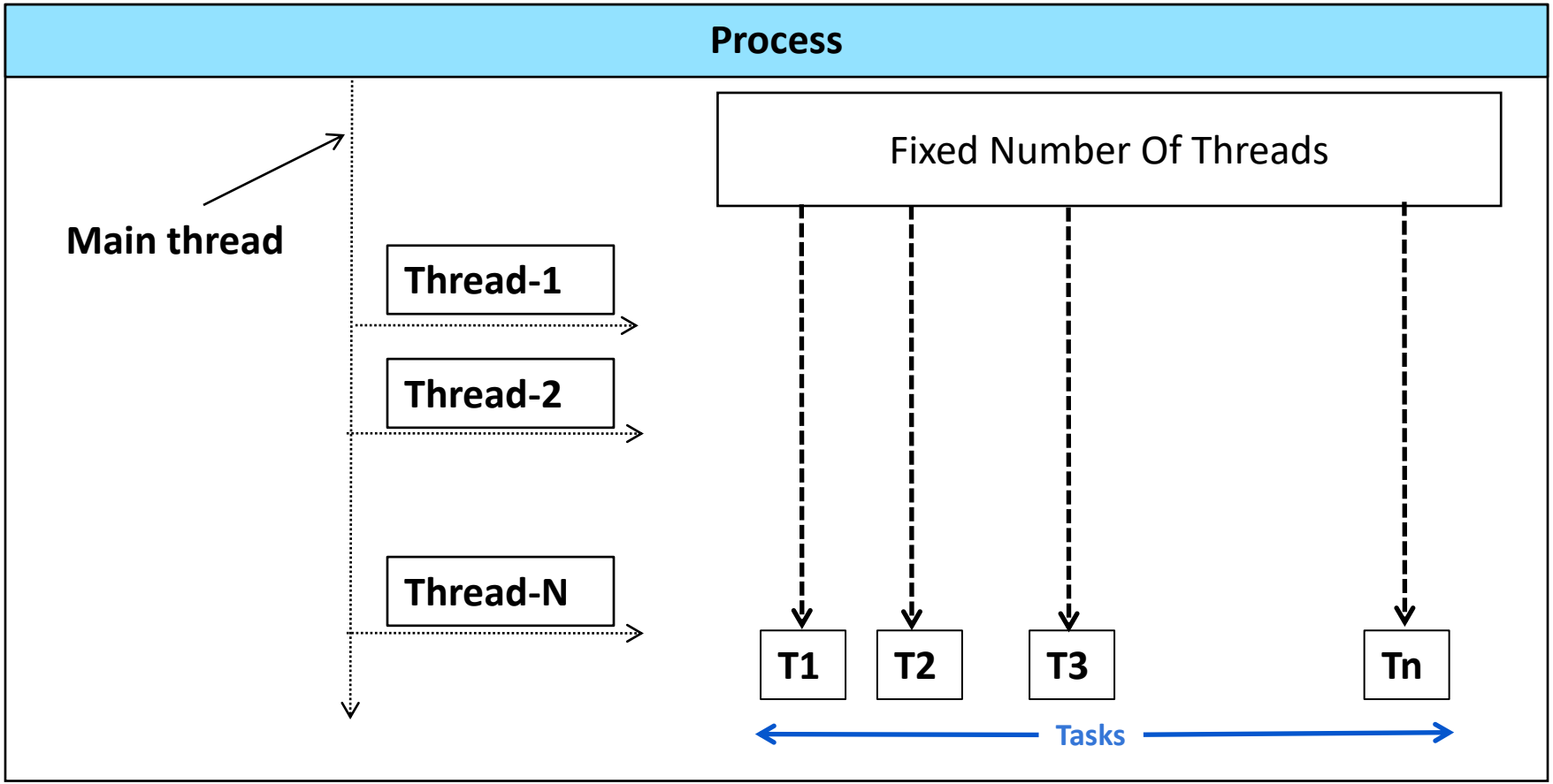
Thread Synchronization

Syntax

```
synchronized (object)
{
   // execute the block.
}
```

```
synchronized  member_function()
{
   // execute the block.
}
```

Java concurrent package



Thread Pools

```
ExecutorService service = Executors.newFixedThreadPool(N);
```

```
ExecutorService service = Executors.newCachedThreadPool();
```

```
service.execute(<Runnable Interface>)
```

```
service.shutdown()
```

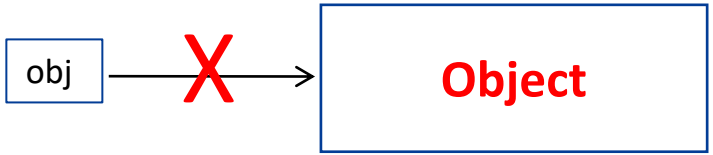
Garbage Collector

Garbage collection is processing removing unused/orphan objects from memory.

Note: Calling garbage collection to JVM is just a request and execution is depends on JVM.

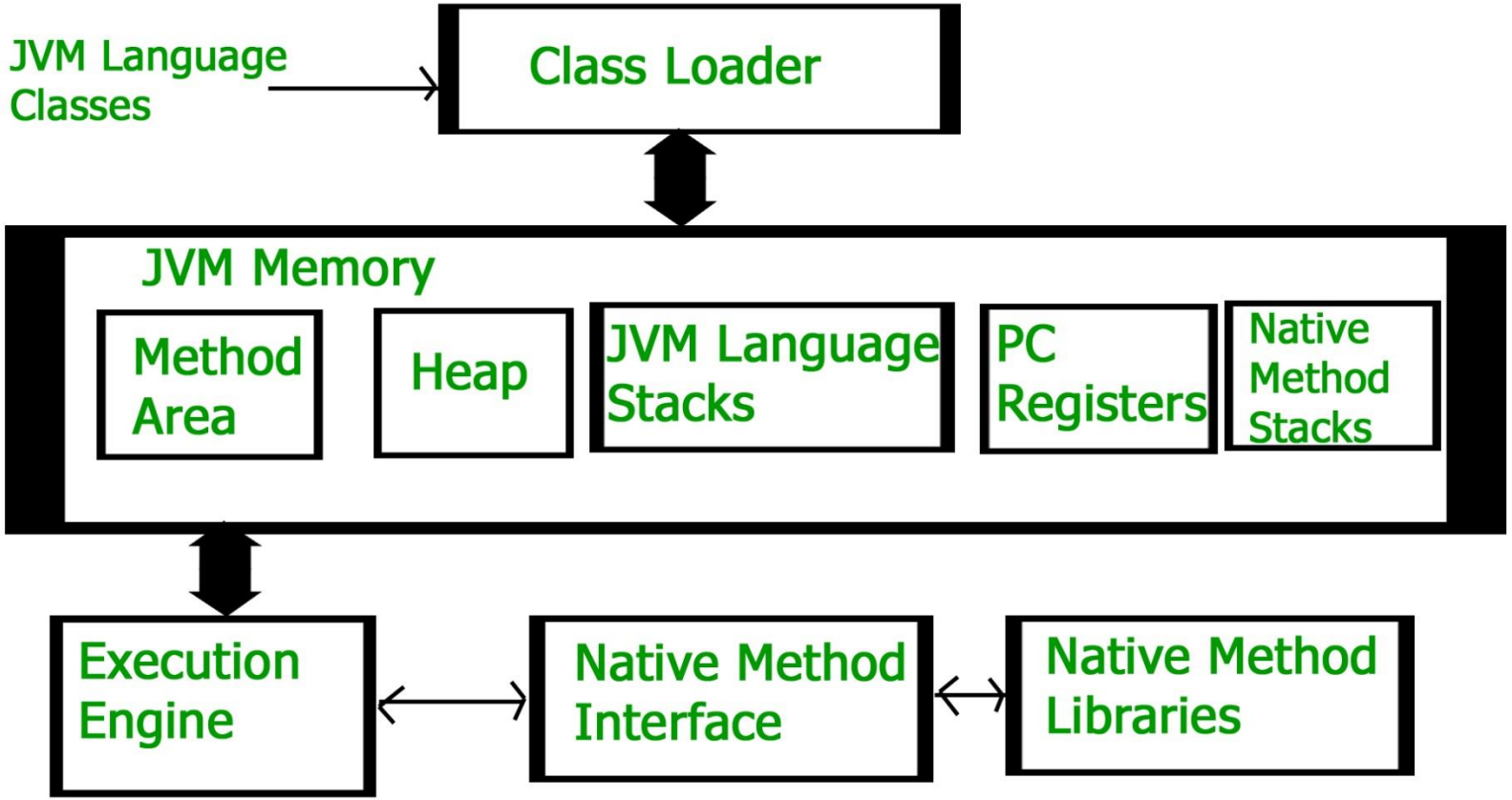
Syntax: System.gc()

- Garbage collector removes the object from memory only when
  - Reference count reachers to zero.
  - When object created in function and function execution completed.

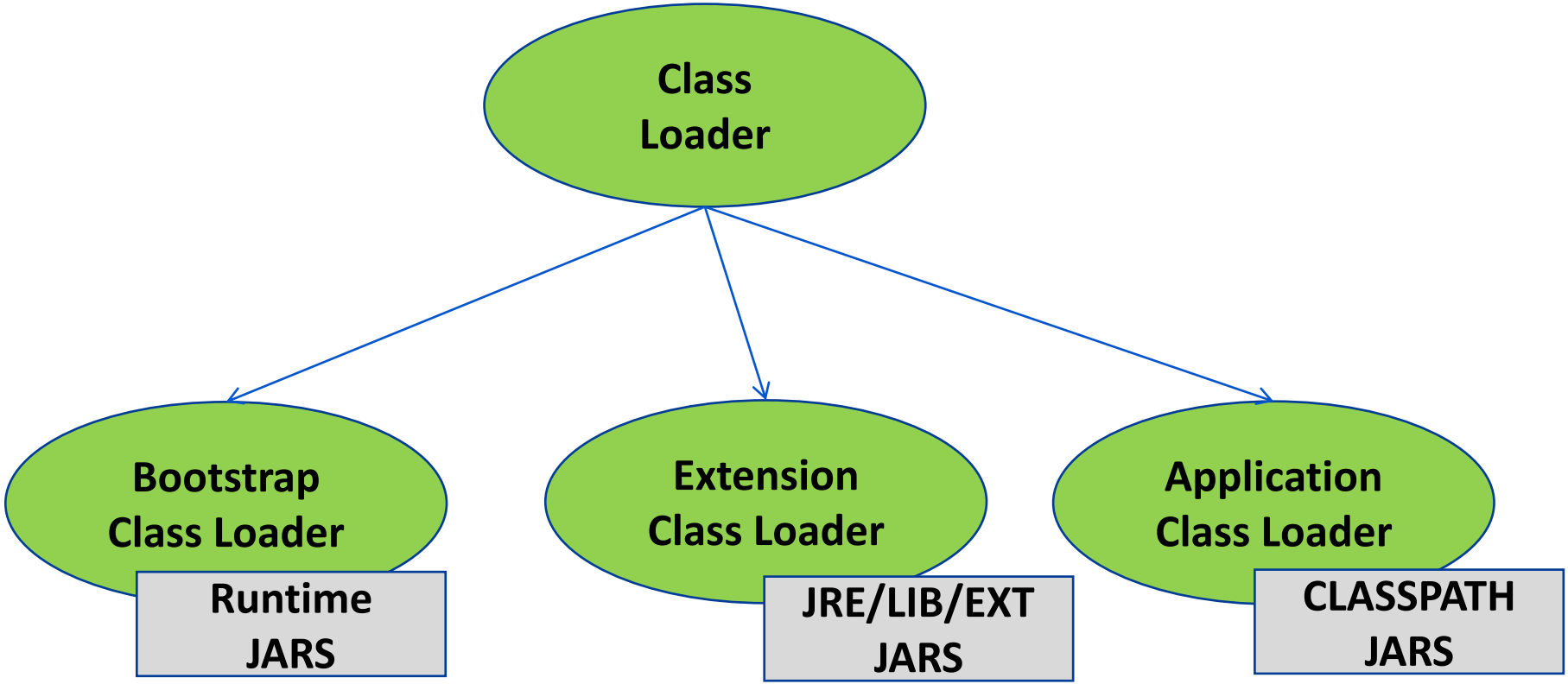


Garbage Collection

JVM Architecture



JVM Architecture – Class Loader



JVM Architecture – JVM Memory

All Class level data (Static) will be loaded into this area.

All objects, arrays and object variables are stored in this area.

Session Summary

- By the end of this session, participants are able to
  - Understand and apply basic Java programming concepts
  - Write applications using general concepts of Java
  - Use different in-built and third party packages of Java.
  - Understand how to use Eclipse IDE for development and debugging of Java applications.

Thank You !

L&T Technology Services