

Chapter 3:

Object Orientation in Java

Classes and Objects

Class

A Class in object oriented programming is a blueprint or prototype that defines **the variables and the methods** (functions) **common to all Java Objects** of a certain kind. Class specifies **what** the object is and **how** it behaves

A class is the specification of **user defined type**.

A class allow us to create **abstract data type** representing **real world object**.

A class is **composed** of **data(attributes or fields)** and **code(behavior or methods)**

Class...

In general a class is specified with the following syntax:

```
class ClassName [extends  
SuperClassName]{  
    [fields declaration]  
    [methods declaration]  
    [Constructors]  
    [Inner Classes]  
}
```

The Empty Class

The smallest possible class (the bare bone class) is a class contains no fields and no methods

```
classClassName  class ClassName extends  
{               Object  
               == {  
               }  
               }  
}
```

Although this empty class doesn't contain any field and methods of its own, **it is not empty**.

Every class in Java is **directly or indirectly derived** from the java **Object class**. The Object class is the parent class of all the classes in java.

So the empty class has some fields and method inherited from the **java Object class**. After all not

General Form of a Class

The general form of the class is:

class ClsName [**extends** SuperClassName]

{

// instance variable declaration

type1 varName1 [= value1];

type2 varName2 [= value2];

:

:

typeN varNameN [= valueN];

Attributes or fields

// Constructors

ClsName(cParType1 cParam1, cParType2 cParam2)

{

//local variables declaration

// body of constructor

Constructors

}

:

:

```
ClsName(cParTypeN cParamN)
```

```
{
```

```
    //local variables declaration
```

```
    // body of constructor
```

```
}
```

```
:
```

```
// Methods
```

```
rType1 methodName1(mParType3 mParam1, mParType4 mParam2)
```

```
{
```

```
    //local variables declaration
```

```
    // body of method
```

```
}
```

```
:
```

```
rTypeN methodNameN(mParTypeN mParamN)
```

```
{
```

```
    //local variables declaration
```

```
    // body of method
```

```
}
```

```
}
```

Constructors

**The order of
placement of
members in
java is
immaterial**

Methods

General Form ...

Where

class is a java keyword used to define a class

clsName is the name or identifier of the class

type1, type2... are the data type of the member variable(field or attribute)

varName1, varName2 ... are identifiers of the member variables

cParType1, cParType2,...are types of the constructor's parameters

cParam1, cParam2,...are parameter identifiers of the constructors

methodName1, methodName2...are method identifiers

rType1, rType2 are return data type of the

General form of attribute definition

Variables defined **in a class method** are called **local variables**. They are **accessible only in the method they are defined**.

Variables defined **within a class** but not inside a method declaration or body are called **fields(or attributes)**.

They are data stored inside a class or instance and **represent the state** or quality of the class or instance.

The genal syntax for declaring attributes is:

[Modifiers] type identifier;

Where:

Modifiers : one or more java's attribute modifiers that modifies the **runtime behavior of the attribute** such as private, public, static..

Scope of Objects in a class

The Scope of an object is the part of the program where the object is accessible.

Methods(Class Level Scope)

These methods must be declared **inside class**. They can be directly accessed anywhere in class

Member Variables or fields(Class Level Scope)

These variables must be declared **inside class (outside any method)**. They can be directly accessed anywhere in class

Local Variables (Method Level Scope)

Variables declared inside a method have method level scop(accessible throughout the method body starting from point of declaration) and can't be accessed outside the method.

Block Variables (Block Scope)

A variable declared inside pair of brackets “{” and

General form of method definition

A method is a function defined within a class. The general format of a method is:

```
[Modifiers] return_type  
methodName([type1 param1, type2  
param2, ..., typeN paramN]) [throws  
exceptionType]{  
    // local variables  
    // statements  
}
```

General form of method...

where

- **Modifiers:** one or more java's attribute modifiers that modifies the **runtime behavior** of the method
- **MethodName** – name that identifies the method;
- **return_type** – the type returned by the method. This may be a **primitive (simple) type, or a reference type (class)** declared additionally in the program;
- **param1, param2, paramN** – parameter names that have **type1, type2, typeN(primitive type, or a reference type (class))** types respectively. The number of parameters in the methods can be arbitrary. In addition, the parameters can be omitted
- **exceptionType:** exception might be thrown by the method

Case Problem

Suppose we want to develop a menu and console based matrix processor application.

Solution

1. Identify the objects in the problem
2. For each objects design a class
 1. Specify the attribute
 2. Specify the Constructors
 3. Specify the methods

Identification of the Objects

There are two objects in the problem domain:

1. matrix
2. Program(Matrix driver)

Defining the Matrix Class

Question you must ask

What are the attributes of a matrix?

What are the service provided by a matrix object(or methods)?

These questions must be answered **in the context of the problem** we want to solve. We must take all features relevant to the problem(Abstraction)

Specifying Attributes of the Vector Class

Fields of Matrix

Fields	Data Type
Symbol	char
elements	int [][]

Specifying Methods of the Matrix Class

Methods of Matrix

Method Names	Description	Method Interface
rowSize	Returns row size	int rowSize()
colSize	Returns column size	int colSize()
setValue	Set element at index i and j	setValue(i, j, val)
getValue	Get element at index i and j	getValue(l, j)
isSquare	Determines if matrix is square	boolean isSquare()
isIdentity	Determines if matrix is identity	boolean isIdentity()
transpose	Transpose a matrix	Matrix transpose(Matrix m)
Add	Subtract one matrix from another	Matrix add(Matrix m)
subtract	Add one matrix from another	Matrix subtract(Matrix ₁₈ m);

```

class Matrix{
    //attributes
    public char symbol;
    public int [][] elements;
    //methods
    public int rowSize()
    {
        //statements;
    }
    public int colSize()
    {
        //statements;
    }
    public void setValue(int i, int j, int
val)
    {
        //statements;
    }
    public int getValue(int I, int j)
    {
        //statements;
    }

    public boolean isSquare()
    {
        //statements;
    }

    public Matrix transpose()
    {
        //statements;
    }
    public boolean isIdentity()
    {
        //statements;
    }
    public Matrix add(Matrix m)
    {
        //statements;
    }

    public Matrix subtract(Matrix m)
    {
        //statements;
    }
    public Matrix multiply(Matrix m)
    {
        //statements;
    }
    public int determinant(Matrix v)
    {
        //statements;
    }
}

```

The Matrix Class...

The matrix class definition contains only **method stub**. The body of each of the methods must be defined

```
public int rowSize()  
{  
    return elements.length;  
}
```

The Matrix Class...

The matrix class definition contains only **method stub**. The body of each of the methods must be defined

```
public int colSize()  
{  
    return elements[0].length;  
}
```

The Matrix Class...

```
public boolean isSquare()  
{  
    return elements.length ==  
    elements[0].length;  
}
```

The Matrix Class...

```
public boolean isIdentity()
{
    if (elements.length != elements[0].length) return
false;
    int size=elements.length;
    for(int i=0; i<size; i++ ){
        for(int j=0; j<size; j++ ){
            if(i==j && elements[i][j] != 1 || i!=j &&
elements[i][j] != 0){
                return false;
            }
        }
    }
    return true;
}
```

Exercise

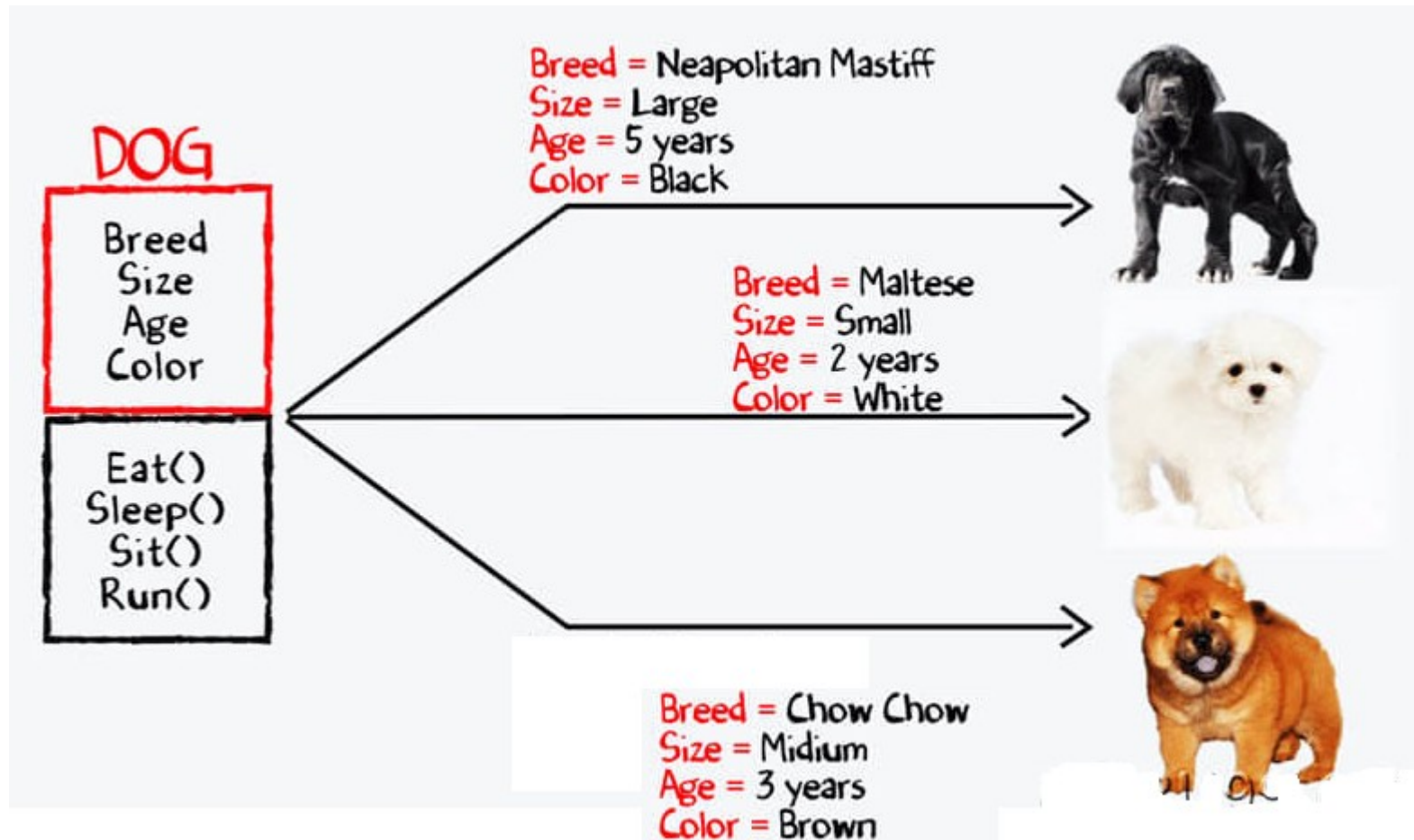
Define the setValue and getValue methods

Object

An object is an **instance of a class**. It has both **state** and **behavior** defined by the class

Object vs Class	
Class	Object
A class defines a collection of objects	Object is an instance of a class
Class is conceptual	Object is real
No memory is allocated for a class	Each objects has its own memory
A class can exist without objects	Objects can't exist without a class
Class doesn't have values associated with fields	Every object has its own value associated with the fields
Example: Car.	Example: Jaguar, BMW, Tesla, etc.

Object



Creating Objects

Objects are created **from already defined classes**.

When you create an object, you are creating an instance of a class, therefore "**instantiating**" a class.

In Java, **the new keyword** is used to create new objects. It is a two step process

1. Creating Reference(Declaration)

Syntax

<class idn> <ref idn>[, <ref idn>...]>;

Example

Matrix m1,m2,m3;//m1 and m2 are references on the stack

- **No object is created by creating reference**

Creating Objects...

2. Instantiating and initialization: Using the new keyword to instantiate and calling the constructor to initialize

Syntax

<ref identifier> = new <Class Name>();

Example

m1=new Matrix(); //m1 now refer to an object created on the heap

m2=new Matrix(); // m2 now refer to another object created heap

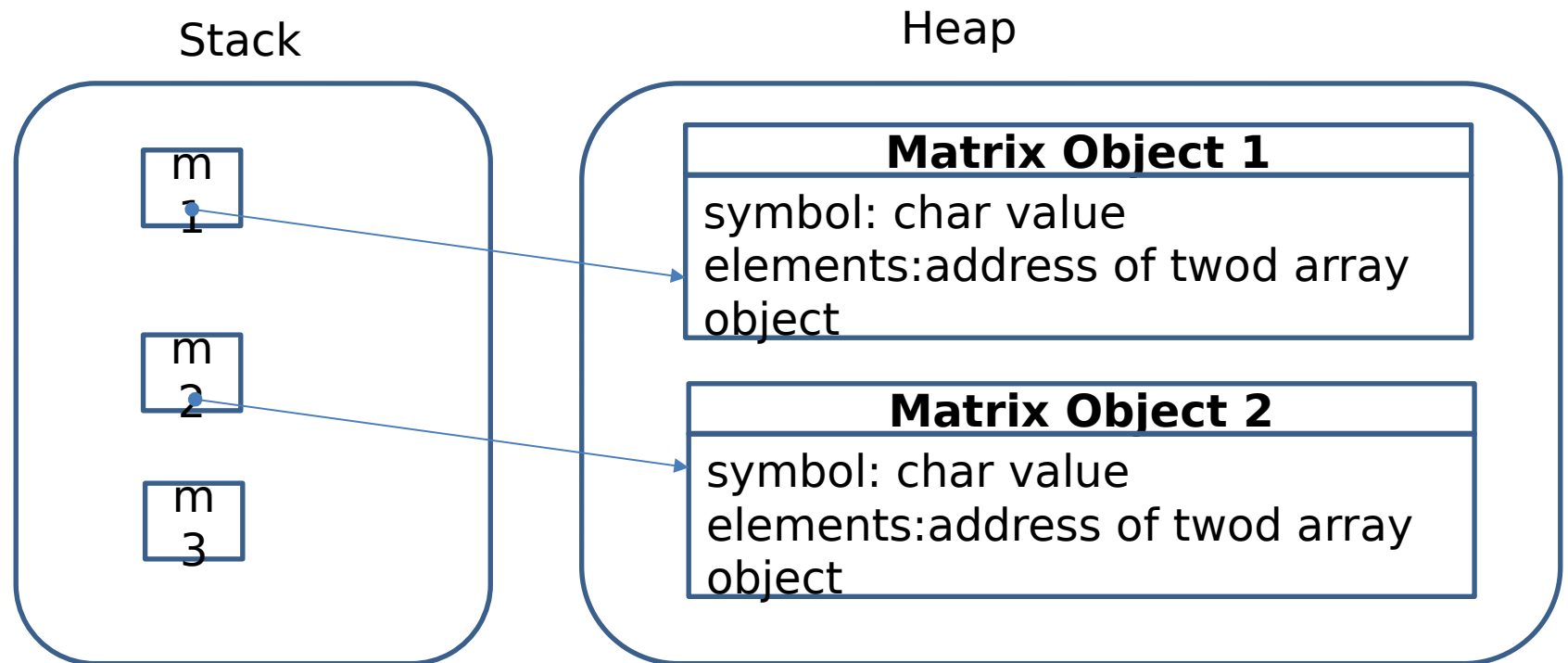
The two steps can be combined into a single syntax

<class idf> <ref identifier> = new < class idf >();

Creating Objects...

An object is uniquely identified by

- its reference name
- defined state(represented by the values of its attributes in a particular time)

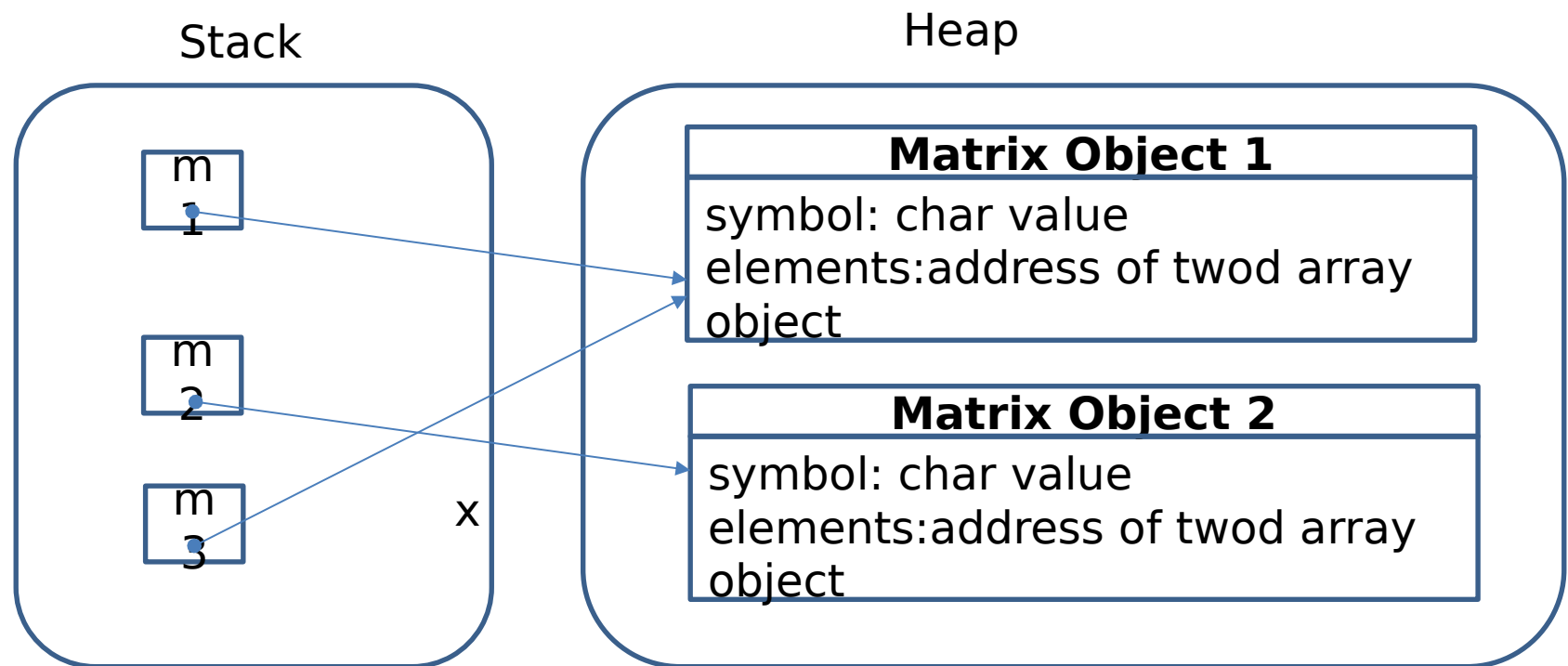


m3 pointes no where in memory. Contains null

Reference Assignment

Assigning one object to another just assigns the object reference. the assignment operator duplicates the reference, not the object.

`m3=m1; //both m1 and m3 refer matrix object 1`

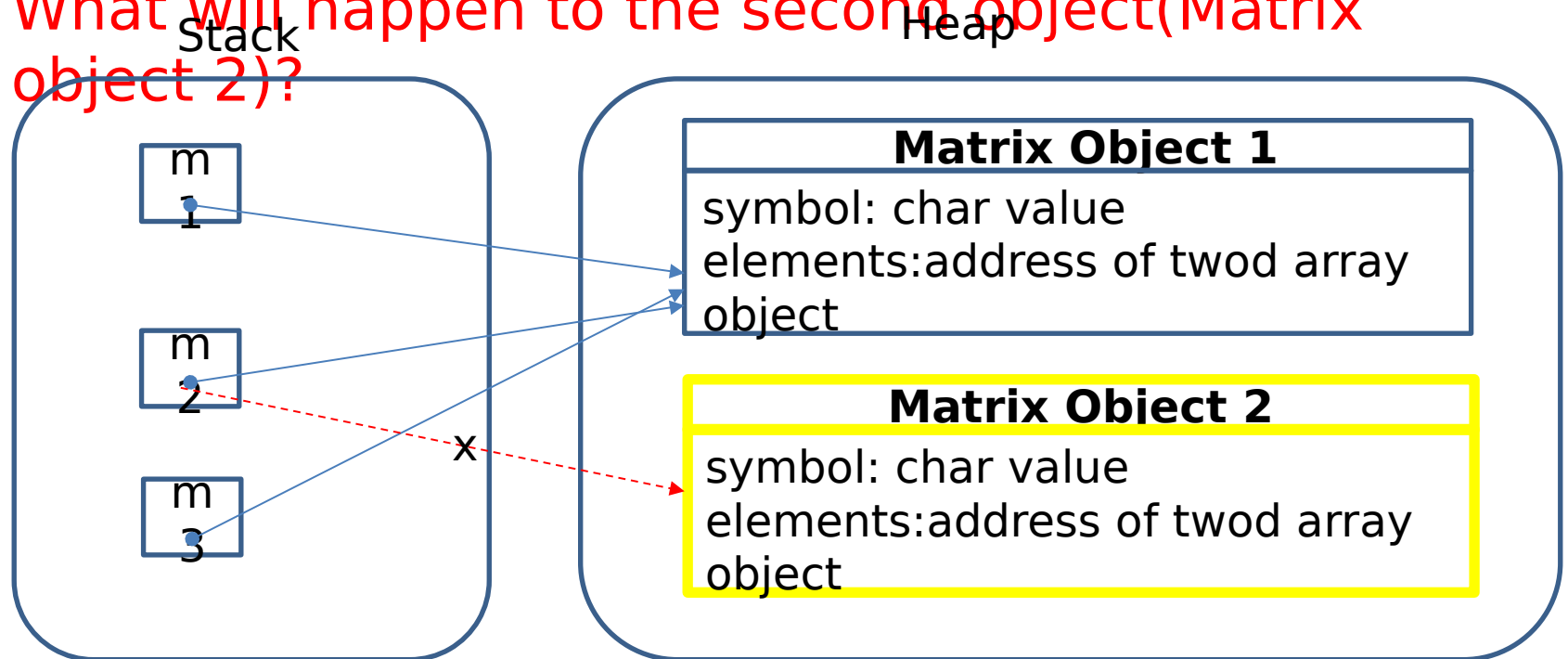


both m1 and m3 points matrix object 1

Reference Assignment...

`m2=m1;`//m1 and m2 now refer to the same object

What will happen to the second object(Matrix object 2)?



An Instance member is a member that belongs to an instance of an object. They are **the non-static member**.

Every time you create an instance of a class, the runtime system creates one copy of each the class's instance variables for the instance.

To access an object's member, simply append the member name to an object reference with an intervening **'.'** (**dot**).

For instance variables:

objectReference.fieldName

For instance methods:

objectReference.methodName([parameters

Accessing Class...

```
public class MatrixDriver{
    public static void main(String [] args){
        Matrix m=new Matrix();
        m.symbol='A';
        m.elements= new int[2][2];
        m. elements[0][0]=1;
        m. elements[1][1]=1;
        System.out.println("Row Size:\t" + m.rowSize());
        System.out.println("Column Size:\t" + m.colSize());
        System.out.println("Square?:\t" +
            m.isSquare()?"Yes":"No");
        System.out.println("Identity?:\t" +
            m.isIdentity()?"Yes":"No");
    }
}
```

Returning Reference(Class Type)

A function can return a primitive or reference to an object.

Let us add a transpose method to our matrix class

```
public Matrix transpose(){  
    int rSize=colSize(); cSize=rowSize();  
    Matrix tm=new Matrix();  
    tm.symbol=symbol;  
    tm.elements= new int[rSize][cSize];  
    for(int i=0; i<rSize; i++ ){  
        for(int j=0; j<cSize; j++ ){  
            tm.elements[i][j]=elements[j][i];  
        }  
    }  
    return tm;  
}
```

Returns reference

Parameter Passing

A method parameters can be primitive and a class type

When a function is called, the arguments in a function can be passed by value or passed by reference.

Pass by value means that a copy of the actual parameter's value is made in memory, i.e. the caller and callee have two independent variables with the same value. If the callee modifies the parameter value, the effect is not visible to the caller.

Pass by reference (also called pass by address) means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function so that a copy of the address of the actual parameter is made in memory, i.e. the caller and the callee use the same variable for the parameter. If the callee modifies the parameter variable, the effect is visible to the caller's variable.

Parameter Passing

Note:

When a method with reference type parameter is invoked, it is the reference of the object not the object itself is passed. If the reference variable is modified in the callee method, the change is not reflected in the caller reference variable but if the referenced object modified, the actual object is also modified.

Parameter Passing

```
public class DogTest{
    public static void main (String []
args) {
        Dog d=new Dog(),d1;
        d.name="Lucy";
        d.age=8;
        d1=d;
        System.out.println("Bofore
Modify");
        System.out.println("%s is %d
years old.",d.name,d.age);
        modify(d1);
        System.out.println("After
Modify");
        System.out.printf("%s is %d
years old.%n",d.name,d.age);
        System.out.println("Equal:
",d==d1);
    }
    public static void modify(Dog
dm){
        dm.name="Boby";
```

```
        System.out.println("In
Modify);
        System.out.printf("%s is %d
years old.
%n",dm.name,dm.age);
    }
}
class Dog{
    public String name;
    public int age;
}
```

What is the output of the program?

Passing Reference (Class Type) Parameters

```
public Matrix add(Matrix m) throws ArithmeticException
{
    int rSize=rowSize(), cSize= colSize();
    if(m.rowSize()!=rSize || m.colSize()!=cSize) {
        throw new ArithmeticException("Can't add matrix.
Size      mismatch!!!");
    }else{
        Matrix rm=new Matrix();
        rm.elements= new int[rSize][cSize];
        rm.symbol='R';
        for(int i=0; i<rSize; i++ ){
            for(int j=0; j<cSize; j++ ){
                rm.elements[i][j]=elements[i][j] +
m.elements[i][j] ;
            }
        }
        return rm;
    }
}
```

Class Type Parameters

```
public Matrix multiply(Matrix m) throws ArithmeticException
{
    int rSize=rowSize(), cSize= colSize();
    int mrSize=m.rowSize(), mcSize=m.colSize() ;
    if(mrSize!=cSize) {
        throw new ArithmeticException("Can't multiply matrix. Size
mismatch!!!");
    }else{
        Matrix rm=new Matrix();
        rm.elements= new int[rSize][cSize];
        rm.name='R';
        for(int i=0; i<rSize; i++ ){
            for(int j=0; j<mcSize; j++ ){
                rm.elements[i][j]=0
                for(int k=0; k<cSize; k++){
                    rm.elements[i][j] +=elements[i][k]*m.elements[k][j]
                }
            }
        }
        return rm;
    }
}
```

Exercise

Add a copy method that copy and return a matrix to our Matrix class

Overloading is a feature that allows a class to have **more than one method having the same name** if their argument lists are different in numbers, type or order.

Its purpose is for name reusing.

Example

```
int myMethod(int x)
```

```
float myMethod(float x)
```

```
double myMethod(double x, double y)
```

Rules for Method Overloading in Java

- The first rule is to **change the method signature**. Method signature refers to the number of arguments, the sequence of parameters, and the data type of parameters.
- If we only change the return type of the method keeping the method name and arguments the

Method Overloading...

```
public class OverloadingDemo{
    public static void main(String []
args){
        OverloadedPrint m=new
OverloadedPrint();
        m.print();
        m.print(3);
        m.print(4.5f);
        m.print(3, 4.5f);
        m.print(4.5f, 3);
    }
}
class OverloadedPrint{
    public void print(){
        System.out.println("I am
Empty!!!");
    }
    public void print(int x, float y){
```

```
        System.out.println("We are
```

```
    public void print(int x){
        System.out.println("I
am one:      int(" + x  + ")");
    }
    public void print(float x, int y){
        System.out.println("We
are two:      float(" + x  + ")
and int(" + y  + ")");
    }
    public void print(float x){
        System.out.println("I
am one:      float(" + x  +
")");
    }
}
```

What is the output of the program?

Method Overloading...

Our matrix class can have additional method that returns a scalar multiplied by a matrix. We can reuse the already used name multiply

```
public Matrix multiply(int scalar)
{
    int rSize=colSize(); cSize=rowSize()
    Matrix tm=new Matrix();
    tm.symbol=symbol;
    tm.elements=new int[rSize][cSize];
    for(int i=0; i<rSize; i++ ){
        for(int j=0; j<cSize; j++ ){
            tm.elements[i][j]=scalar*elements[i][j];
        }
    }
    return tm;
}
```

Constructor

A constructor is a special function that is used to **initialize objects**. It can be used to set **initial values** for object attributes when the object is created.

Characteristics of Constructor

- Constructors **name must be similar to that of the class name** inside which it resides.
- Constructors are automatically **called when an object is created**.
- Constructors **do not have a return type**; not even void
- Constructors **cannot return a value**.
- Constructors **can be private**.
- A constructor **cannot be abstract, static, final, native, strictfp, or synchronized**
- **A constructor can be overloaded..**

Constructor: General format

The general format of a constructor is:

```
public ClassName([parameters declaration]) [throws  
exceptionType]{  
    <[super([parameter list]);]>  
    <[this([parameter list]);]>  
    <statements>  
}
```

Where

ClassName: name of the class

exceptionType: exception that may be thrown by the method

Parameters declaration: optional declaration of parameters of the constructor

super([parameter list]): optional call to the base class constructor. **super** is a keyword

this([parameter list]): optional call to constructors in the same class. **This** is a keyword

Note: It is possible to make the constructor private however

Types of Constructor

There are three types of constructor in java:

1. Default Constructor

If you don't implement any constructor in your class, the Java compiler inserts default constructor into your code on your behalf during compilation. The attributes will be initialized by their default value. **Once we define our own constructor for the class, the default constructor is no longer used.**

2. No-arg Constructor

Constructor with no arguments is known as no-arg constructor. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

3. Parameterized Constructor

Constructor with arguments is known as Parameterized constructor.

A specialized type of parameterized constructor that

Constructor...

Example

Let us create **a no-arg** constructor for our Matrix class.

```
public Matrix() {  
}
```

Now we can create a new vector this way: A null matrix, A matrix that doesn't have a name and elements

```
Matrix v=new Matrix();
```

The this keyword

Within **an instance method or a constructor**, **this** is a reference to the **current object — the object whose method or constructor is being called**. You can refer to any member of the current object from within **an instance method or a constructor** by using this.

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter

Constructor...

Create another parameterized **overloaded** constructor for our Matrix class that take symbol, row and column as parameter

Example: Let us create another parameterized **overloaded** constructor for our Matrix class:

```
public Matrix(char symbol, int row, int col) throws
IllegalArgumentException {
    String errMsg="";
    if(Character.isAlphabetic(symbol)){
        errMsg="Invalid symbol value!!!";
    }else if(row<1 || col<1){
        errMsg="Invalid matrix dimension!!!";
    }
    if(!errMsg.equals("")){
        this.symbol='A';
        elements=new int[row] [col];
    }else{
        throw new IllegalArgumentException(errMsg);
    }
}
```

**The field symbol
is shadowed by
the argument
symbol**

Now we can create a new Matrix this way: a 3x2 matrix all elements having a zero value with a name A
Matrix v=new Matrix('A',3,2);

Constructor Chaining or Cascading

Constructor chaining is the process of **calling one constructor from another constructor** with respect to current object.

Constructor chaining can be done in two ways:

- **Within same class:** It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class

Constructor...

Create another parameterized **overloaded** constructor for our Matrix class that take symbol, and one dimension as parameter and create a square matrix

Constructor Chaining Within the Same Class

Suppose we want to create an overloaded constructor which creates **square matrix**

```
public Matrix(char symbol, int dimension) throws  
IllegalArgumentException {  
    this(symbol, dimension, dimension);  
}
```

Now we can create a new Square Matrix this way: a 3x3 square matrix all elements having a zero value with a name A

```
Matrix m=new Matrix('A',3);
```

Exercise

1. Create a constructor that create an identity matrix
2. Create a constructor that create a matrix and initialize the matrix with elements taken from one dimensionalarray. Your matrix accepts a one dimensional array that contains the values for the elements

Example

Let us create another parameterized overloaded **copy constructor** for our vector class:

```
public Matrix(Matrix mat) {  
    symbol=mat.symbol;  
    elements=new int[mat.rowSize()] [mat.colSize()];  
    for(int i=0; i< mat.rowSize(); i++)  
    {  
        for(int j=0; j< mat.colSize(); j++){  
            elements[i][j]=mat.getValue(i,j);  
        }  
    }  
}
```

Now we can create a new matrix this way:

Matrix **m=new** Matrix('B', 3);

Matrix **m1=new** Matrix(**m**); **//we create a copy of Matrix m**

Access Specifier

Access level modifiers determine whether other classes can **use a particular field or invoke a particular method**. It helps **to restrict the scope** of a class, constructor, method, or data member.

There are four types of access modifiers available in java:

1. Default - No keyword required

If you don't specify any modifier, then java provides a modifier by default and it is called default access modifier. A default class and members(fields, methods or constructors) can be accessed within the package declared.

2. Private

The private access specifier in java is specified by private keyword. The scope of the private access modifier is within the class.

We can't declare a top-level class as private. Java

Access Specifier

3. Protected

The protected access specifier in java is specified by a protected keyword. The scope of the protected access modifier is within the package and outside the package but through inheritance only.

1. You can't use the protected keyword with class.
2. It is more flexible than default and private modifiers.

4. Public

The public access modifier in java is specified by the public keyword. The scope of public access modifier is everywhere.

1. It is the most flexible modifier in java.
2. It does not provide any restrictions.

Access Specifier Summary

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Access Specifier

In our Matrix class specification, we make all members public leaving the class open to the outside. We don't have any control on how the member data are manipulated. One of the principle of OOP is violated, in particular encapsulation.

In encapsulation, **the fields of a class will be hidden from other classes**, and can be accessed only through well defined methods of their current class.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

```

class Matrix{
    //attributes
    private char symbol;
    private double [][] elements;
    //methods
    public int rowSize()
    {
        //statements;
    }
    public int colSize()
    {
        //statements;
    }
    public void setValue(int i, int j, int
val)
    {
        //statements;
    }
    public int getValue(int I, int j)
    {
        //statements;
    }
    public boolean isSquare(Matrix
m)
    {
        //statements;
    }
    public Matrix transpose(Matrix m)
    {
        //statements;
    }
    public boolean isIdentity(Matrix m)
    {
        //statements;
    }
    public Matrix add(Matrix m)
    {
        //statements;
    }
    public Matrix subtract(Matrix m)
    {
        //statements;
    }
    public Matrix multiply(Matrix m)
    {
        //statements;
    }
    public int determinant(Matrix v)
    {
        //statements;
    }
}

```

Getter and Setter methods

We made the fields in our Matrix class private hiding them from other class. We made them inaccessible to other class. We don't want to close the door totally, what we want is a guard that controls how the field is accessed so the getter and setter method for each fields.

The get method returns the value of the field. General syntax:

```
public fieldType getField(){  
    return field;  
}
```

The set method takes a parameter (fieldVal) and assigns it to the name variable. General syntax

```
public void setField(fieldType fieldVal){  
    field=fieldVal;  
}
```

Exercise

1. Create a setter and getter for the Matrix class for the symbol field. A symbol must be alphabetic character
2. Create a setter and getter for the Matrix class for the elements field.

Exercise

Define a class called Point that defines a point in a cartesian plane. A point is specified with the x and y coordinates

Your class should have

1. Three constructors

- No arg constructor that initialize the members by zero

- Constructor that takes x and y

- Copy constructor

2. Setter and Getter for the two instance variable

3. Methods

- void translateX(double x)

- void translateY(double y)

- double distance(Point p)

Exercise

Define a class called Line that defines a line in a cartesian plane. A line is specified $ax + by = c$

Your class should have

1. Five constructor

Constructor that takes a, b and c

Constructor that takes slope and yIntercept

Constructor that takes slope and one point

Constructor that takes two points

Copy constructor

2. Setter and Getter for the three instance variable

3. Methods

double YIntercept()	void translateX(double x)
double xIntercept()	void translateY(double y)
double slope()	void scaleX()
double yValue(double x)	void scaleY()
double xValue(double y)	Line perpendicular()
boolean online(Point p)	Point intersection(Line l)
	void rotate(double degree)

Garbage Collection And Finalize Method

Garbage Collection is process of **reclaiming the runtime unused memory** automatically. In other words, it is a way **to destroy the unused objects**. It is **automatically done** by garbage collector (gc) thread in java.

When there are no references to an object, it is assumed to be dead and no longer needed. So the memory occupied by the object can be reclaimed.

There are various ways in which the references to an object can be released to make it a candidate for Garbage Collection.

By making a reference null

```
Student student = new Student();  
student = null;
```

By assigning a reference to another

```
Student studentOne = new Student();  
Student studentTwo = new Student();  
studentOne = studentTwo;
```

Finalize is a method that the Garbage Collector always **calls just before the deletion/destroying the object** which is eligible for Garbage Collection, so as **to perform clean-up activity**. Clean-up activity means closing the resources associated with that object like Database Connection, Network Connection or we can say resource de-allocation.

- To add a finalizer to a class, you simply define the finalize() method.
- The Java run time calls that method whenever it is about to recycle an object of that class. That can be at any time after it has become eligible for garbage collection. Note that it's entirely possible that an object never gets garbage collected (**and thus finalize is never called**)
- Specify those actions that must be performed before an object is destroyed

The syntax for finalize is:

```
protected void finalize(){  
//your code here
```

Instance vs Static Variables

There are three types of variables in Java:

- Local Variables
- Instance Variables
- Static or Class Variables

1. Local Variables:

A variable defined **within a block or method or constructor** is called local variable. These variable are **created when the block is entered** or the function is called and **destroyed after exiting from the block or when the call returns from the function.**

- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
- **Initialization of Local Variable is Mandatory.**

Instance vs Static Variables

2. Instance Variables:

Instance variables are **non-static variables** and are declared in a class **outside any method, constructor or block**. As instance variables are declared in a class, these variables are **created when an object of the class is created and destroyed when the object is destroyed**.

- Unlike local variables, **we may use access specifiers for instance variables**. If we do not specify any access specifier then the default access specifier will be used.
- **Initialization of Instance Variable is not Mandatory**. Its default value is 0
- **Outside the class, instance Variable can be accessed only by creating objects.**

Instance vs Static Variables

3. Static or Class Variables:

Static variables are **variables** declared with the keyword **static** in a class **outside any method, constructor or block**

When a variable is declared as static, **then a single copy of the variable is created and shared among all objects** at a class level. Static variables are, essentially, global variables. All instances of the class share the same static variable. Important points for static variables :-

- We can create static variables at class-level only.
- Outside the class, public static variables can be accessed without creating an object
- static blocks and static variables are executed in order they are present in a program.

Static variable Syntax

static data_type variable_name;

Instance vs Static Variables

```
class StaticVariableDemo
{
    private int value;
    static int count=0;
    public void
StaticVariableDemo()
    {
        value=0;
        count++;
    }
    public void setValue(int
value){
        this.value=value;
    }
    public int getValue(){
        return value;
    }
}
```

```
public static void main(String args[])
{
    StaticVariableDemo obj1=new
StaticVariableDemo();
    StaticVariableDemo obj2=new
StaticVariableDemo();
    obj1.setValue(3);
    System.out.println("Obj1: value
is="+obj1.getValue());
    System.out.println("Obj2: value
is="+obj2. getValue());
    System.out.println("Obj1: count
is="+obj1.count);
    System.out.println("Obj2: count
is="+obj2.count);
    System.out.println("
StaticVariableDemo : count is="+
StaticVariableDemo.count);
}
```

Instance and Static Methods

There are two types of methods

- **Instance Methods**

These are methods which are **not declared static**.

They are invoked on specific instance of a specific class. Method wants to know upon which class it was invoked. The way it happens there is a invisible parameter called '**this**'. Inside of 'this' we have members of instance class already set with values.

- **Static Methods**

These are methods which are **declared static**. These methods are methods in java **that can be called without actually creating an object of that class**. it becomes part of the class or a class method(**a method that belongs to a class rather than to an object**). They are designed with aim to be shared with all objects created with that class.

Instance and Static Methods

```
class A {  
    int a;  
    int b;  
    public A(int a, int b){  
        this.a=a;  
        this.b=b;  
    }  
    public int add(){  
        return this.a + this.b;  
    }  
  
    public static add(int s1, int s2){  
        return (s1 + s2);  
    }  
}
```

```
class B{  
    int s=8;  
    int t = 5;  
    public int add(int s,int t){  
        return A.add(s,t);  
    }  
}  
  
class Program {  
    public static void main(String [] args){  
        A objA = new A(1,2);  
        int sum1=objA.add();  
        int sum2=A.add(3,5);  
        B objB = new B();  
        int sum3 =objB.add();  
        System.out.println("Sum1: " + sum1);  
        System.out.println("Sum2: " + sum2);  
        System.out.println("Sum3: " + sum3);  
    }  
}
```

Instance and Static Methods

When to use static methods?

- When you have code that can be shared across all instances of the same class, put that portion of code into static method.
- They are basically used to access static field(s) of the class.

Instance method vs Static method

- Instance method can access the instance methods and instance variables directly.
- Instance method can access static variables and static methods directly.
- Static methods can access the static variables and static methods directly.
- Static methods can't access instance methods and instance variables directly. They must use reference to object. And They cannot refer to this or super in any way

Static Block

The static block is a block of statement inside a Java class that will be **executed when a class is first loaded into the JVM**. A static block helps to **initialize the static data members**, just like constructors help to initialize instance members.

This code inside static block is executed only once: the first time the class is loaded into memory.

static blocks are executed before constructors.

Static Block

```
class Test {  
    static int i;  
    int j;  
    static {  
        i = 10;  
        System.out.println("static block  
called ");  
    }  
    Test(){  
        System.out.println("Constructor  
called");  
    }  
    public void demo(){  
        System.out.println("Method  
called");  
    }  
}
```

```
class Main {  
    public static void  
main(String args[])  
    {  
        new Test().demo()  
        new Test().demo()  
    }  
}
```

What is the output of

Inner(or nested) class

In Java, it is also possible to nest classes (a class within a class). To access the inner class, create an object of the outer class, and then create an object of the inner class. The inner class can be default, public, private or protected/ It can be also static

The general syntax for creating a package is

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

Note

- **Inner classes can access attributes and methods of the outer class:**
- **just like methods, a static inner class does not have access to instance members of the outer class.**

Inner(or nested) class

```
class OuterClass {  
    int x = 10;  
    class InnerClass {  
        int y = 5;  
    }  
}
```

```
public class Main {  
    public static void main(String[]  
args) {  
        OuterClass myOuter = new  
OuterClass();  
        OuterClass.InnerClass myInner  
= myOuter.new InnerClass();  
        System.out.println(myInner.y +  
myOuter.x);  
    }  
}
```

```
class OuterClass {  
    int x = 10;  
    private class InnerClass {  
        int y = 5;  
    }  
}
```

```
public class Main {  
    public static void main(String[]  
args) {  
        OuterClass myOuter = new  
OuterClass();  
        OuterClass.InnerClass myInner  
= myOuter.new InnerClass();  
        System.out.println(myInner.y +  
myOuter.x);  
    }  
}
```

What is the output?

PACKAGE in Java is a collection of classes, sub-packages, and interfaces. It helps organize your classes into a folder structure and make it easy to locate and use them.

Advantage of using package

Using packages while coding offers a lot of advantages like:

- **Re-usability:** The classes contained in the packages of another program can be easily reused
- **Name Conflicts:** Packages help us to uniquely identify a class, for example, we can have `company.sales.Employee` and `company.marketing.Employee` classes
- **Controlled Access:** Offers access protection such as protected classes, default classes and private class
- **Data Encapsulation:** They provide a way to hide classes, preventing other programs from accessing classes that are meant for internal use only

Creating Package

The general syntax for creating a package is
package

**packageName1[.packageName2[.packageName3[...]
]];**

How to Create a package?

1. Choose the name of the package.
2. Include the package command as the first line of code in your Java Source File.
3. The Source file contains the classes, interfaces, etc you want to include in the package.
4. Compile to create the Java packages.

Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

If the package statement is omitted, the class belongs to the so-called **default package**, with no name and no sub-directory structure. Use of default package is not recommended other than writing toy program and for quick testing

You can create a hierarchy of packages

```
package mycalculator;
public class Add{
    int a,b;
    public Add(){
        a=b=0;
    }
    public Add(int a, int b){
        this.a=a;
        this.b=b;
    }
    public int add(){
        return a+ b;
    }
}
```

Save the file with name Add.java in the

workspace directory

Open cmd and change into the workspace directory and compile the Add.java file

```
javac -d . Add.java
```

This command forces the compiler to create

a package n1 in the working directory

Now let us create **a sub package** mycalculator2 in mycalculator

```
package
mycalculator.mycalculator2;
public class Multiply{
    int a,b;
    public Multiply(){
        a=b=0;
    }
    public Multiply(int a, int b){
        this.a=a;
        this.b=b;
    }
    public int multiply(){
        return a*b;
    }
}
```

Save the file with name Multiply.java in the workspace directory

Open cmd and change into the workspace directory and compile

Using classes in a Package

There are three ways to access the package from outside the package.

- `import packagename.*;`
If you use `package.*` then all the classes and interfaces of this package will be accessible but not subpackages.
E.g `import mycalculator.*;`
- `import packagename. classname;`
If you import `package.classname` then only declared class of this package will be accessible.
E.g `import mycalculator.mycalculator2.Multiply`
- fully qualified name.
Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.
`mycalculator.mycalculator2.Multiply obj;`

Example

Using fully qualified name

```
package programpackage;
public class ProgramTest{
    public static void main(String [] args){
        mycalculator.Add objAdd=new mycalculator.Add(2,5);
        mycalculator.mycalculator2.Multiply objMul= new
        mycalculator.mycalculator2.Multiply ();
        system.out.println("Sum: " + objAdd.add());
        system.out.println("Product: " + objMul.multiply());
    }
}
```

To compile

javac -d . ProgramTest

To run it

java programpackage ProgramTest

Using import

```
package programpackage;
import mycalculator.*; //importing all
import mycalculator.mycalculator2.Multiply;//importing only Multiply
public class ProgramTest2{
    public static void main(String [] args){
        Add objAdd=new Add(2,5);
        Multiply objMul= new Multiply (2,5);
        System.out.println("Sum: " + objAdd.add());
        System.out.println("Product: " + objMul.multiply());
    }
}
```

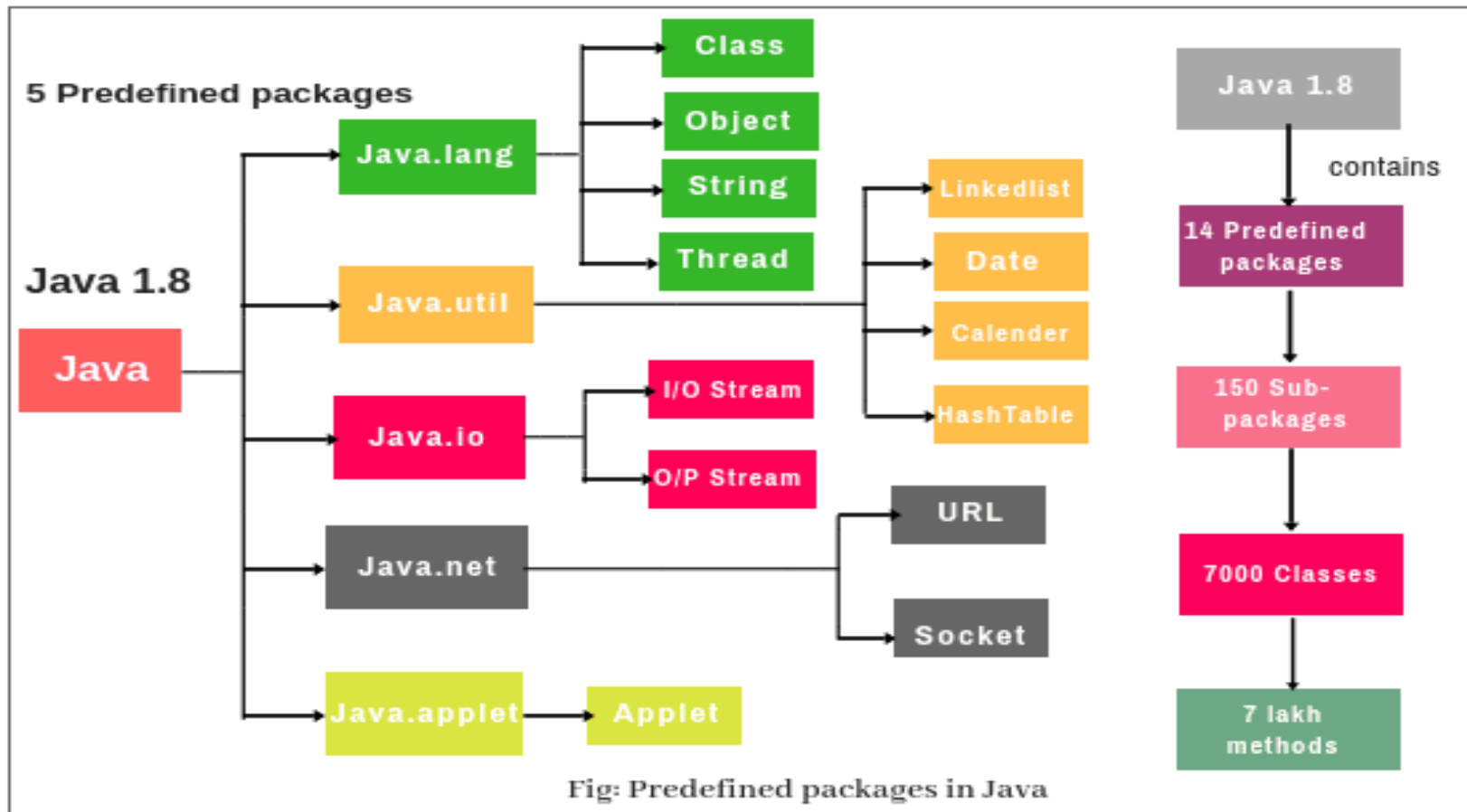
To compile

javac -d . ProgramTest2

To run it

java programpackage ProgramTest2

Java Packages



The java.lang package

The java.lang package provides classes that are fundamental to the design of the Java programming language. The most important classes are

- Object: which is the root of the class hierarchy,
- Class: instances of which represent classes at run time
- The wrapper classes Boolean, Character, Integer, Long, Float, and Double
- Math: functions such as sine, cosine, and square root
- String: The String class represents character strings.
- StringBuffer: A thread-safe, mutable sequence of characters.
- StringBuilder: A mutable sequence of characters
- System: The System class contains several useful class fields and methods: Fields: in,out,err for input/output

Important classes in java.lang package

1. Primitive Wrappers

The java.lang package includes a class for each Java primitive type:

Boolean, Byte, Short, Character, Integer, Float, Long, Double, Void

```
Integer x=Integer.valueOf(2);
```

```
Double y= Double.valueOf(2.5);
```

used for

- Storing primitives in Object based collections
- Parsing/decoding primitives from Strings, for example: `int value = Integer.parseInt(str);`
- Converting/encoding primitives to Strings
- Range definitions (e.g. `Byte.MAX_VALUE`)
- Toolbox of convenience methods — many of them static (e.g., `isLetter()`, `toLowerCase()`, `reverseBytes()`, `isInfinite()`, `isNaN()`)

The automatic conversion of primitive data types into its corresponding wrapper classes is called autoboxing. The reverse process is called unboxing

Overview of ...

2. String

- Automatically created with double quotes
- Pooled (interned) to save memory
- Immutable — once created cannot change
- Concatenated using the expensive + operator
- Many methods for string manipulation/searching

3. StringBuilder and StringBuffer

- Mutable — can quickly grow and shrink as needed
- Few methods modifying the buffer: **append()**, **insert()**, **delete()**, **replace()**
- Use **toString()** if you need to get a String representation of the object's content
- StringBuffer is synchronized for thread-safe access in multithreaded applications
- Use StringBuilder for better performance unless you plan to access the object from multiple threads

StringBuilder strBld=new StringBuilder("I am stringbuilder");

4. Math

- The Math class contains static methods implementing math functions, such as:
- Trigonometry: `sin()`, `cos()`, `tan()`, etc.
- Rounding: `round()`, `floor()`, `ceil()`, `rint()`
- Logarithmic: `log()`, `log10()`
- Exponentiation/root: `sqrt()`, `cbrt()`, `pow()`, `exp()`
- Utility: `abs()`, `min()`, `max()`, `random()`, `hypot()`, `toDegrees()`, `toRadians()`
- Math also includes constants for `PI` and `E`.

5. Object

Every class in Java is directly or indirectly derived from the `Object` class. Important methods in this class:

- `toString()` provides String representation of an `Object` and used to convert an object to String.
- `equals(Object obj)` : Compares the given object to “this” object (the object on which the method is called)
- `getClass()` : Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to