Java Basics

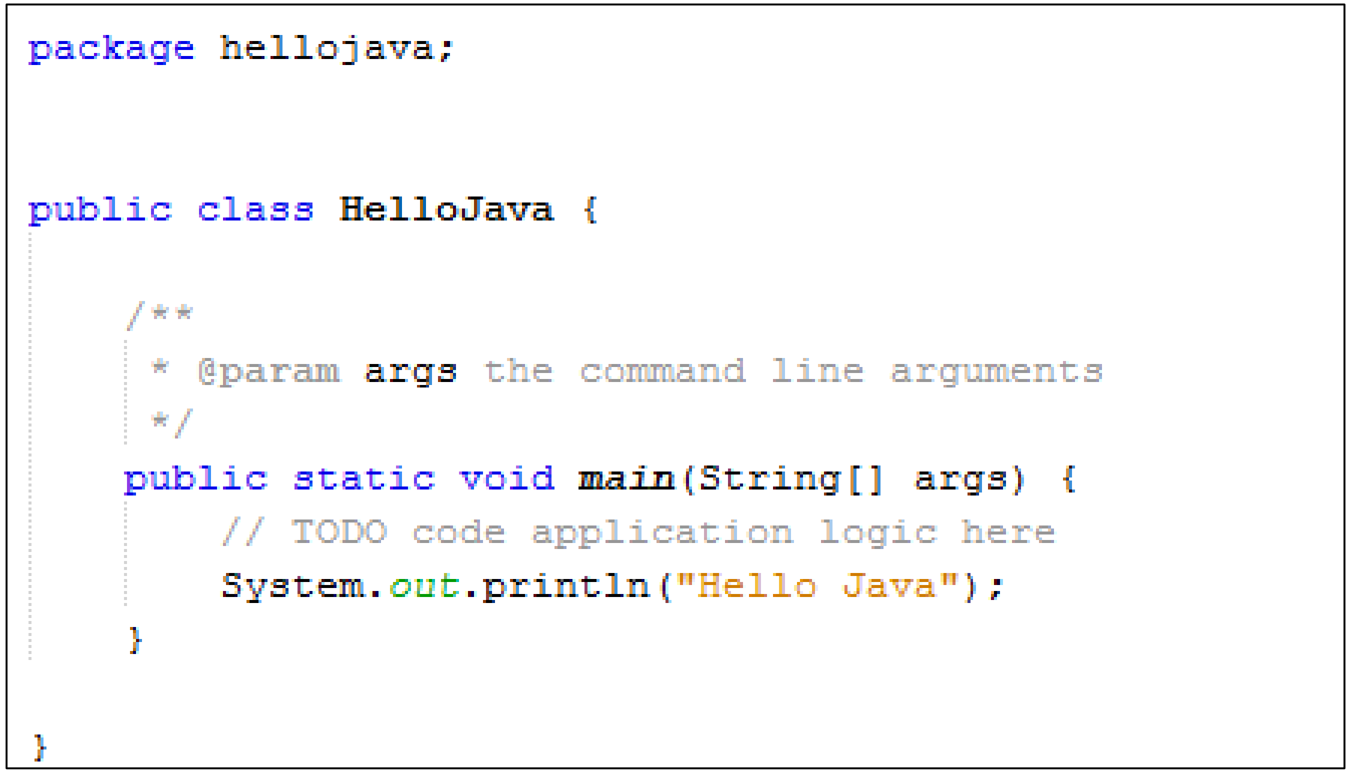
What is Java

- Java is a **programming language** and a **platform**.
- Java is a simple, high level, robust, secured, Platform independent , portable and object-oriented programming language.
- **Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API, it is called platform.

Notes on Java

- Java's syntax is very similar to C++.
- Java is pure object-oriented.
  - Everything must be in a class (including the main method)
- Java is easier in some aspects.
  - No manual dynamic memory management : **Garbage Collector**
  - No Pointers

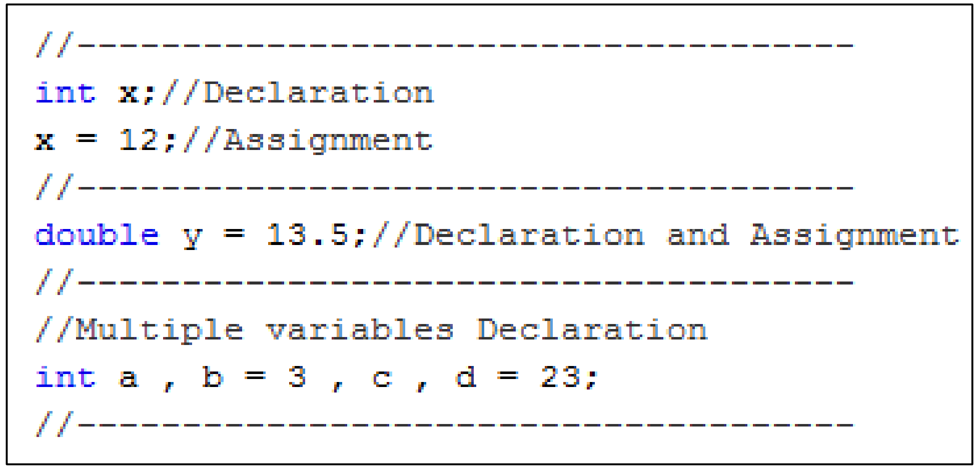
First Java Program



```java
package hellojava;


public class HelloJava {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        System.out.println("Hello Java");
    }

}
```

Variables

- A variable is a placeholder for some **data**.
- Declaring a variable is equivalent to asking the computer to set aside space in the main memory.
- Variables can have their values changed any time using assignments statements.
- Variables cannot be used until they have been initialized (assigned a value for the first time).



```
//-------------------------------------
int x;//Declaration
x = 12;//Assignment
//-------------------------------------
double y = 13.5;//Declaration and Assignment
//-------------------------------------
//Multiple variables Declaration
int a , b = 3 , c , d = 23;
//-------------------------------------
```

Built-in Data Types in Java

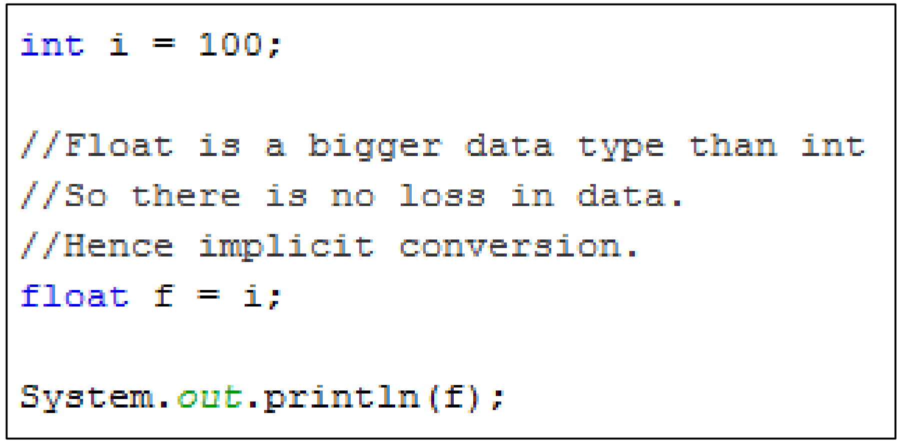
Type	Description	Default value	Range
byte	The byte data type is an 8-bit signed integer	0	-128 to 127
short	The short data type is a 16-bit signed integer.	0	-32,768 to 32,767
Int	The int data type is a 32-bit signed integer.	0	-2^{31} to $2^{31}-1$
long	The long data type is a 64-bit signed integer.	0	-2^{63} to $2^{63}-1$
float	The float data type is a single-precision 32-bit floating point.	0.0	±1.4e-45 to ±3.4e+38
double	The double data type is a double-precision 64-bit floating point.	0.0	±4.94e-324 to ±1.79e+308
char	The char data type is a single 16-bit Unicode character.	'\u0000'	'\u0000' (or 0) to '\uffff‘ (or 65,535)
boolean	The boolean data type has only two possible values: true and false.	false	true and false

Common Operators

- Assignment operator – “=”
- Arithmetic operators - “+,-,*,/,%”
- Increment, Decrement operators - “++,--”
- Comparison operators - “==,!=,>,<,>=,<=”
- Logical operators - “&&,||”

Data Types Conversion

- Implicit Conversion
  - Done by the compiler.
  - When there is no loss in information.
  - Example : converting int to float



```
int i = 100;

//Float is a bigger data type than int
//So there is no loss in data.
//Hence implicit conversion.
float f = i;

System.out.println(f);
```

Data Types Conversion

- Explicit Conversion
  - Example : when converting float to int , loss in fractional part in the original number and also a possibility of overflow .
  - In the case of the previous example an explicit conversion is required .
  - In Java we use cast operators .

```
float f = 112.14f;
//Cannot implicitly convert float to int.
//float is a bigger data type than int.
//Fractional part will be lost.

//int i = f;

//use cast operator () for Explicit conversion
int i = (int)f;

System.out.println(i);
```

String

- A character or a group of characters written between " "is called a String
- Example escape sequences included in Java
  - \n for new line
  - \t for a tab
  - \"for a double quote
  - \'for a single quote
  - \\ for a back slash

Comments in Java

- block or multiline comments
  - delimited with /* and */
- Single line comments
  - delimited with //
- JavaDoc comments
  - delimited with /**and */

Constants

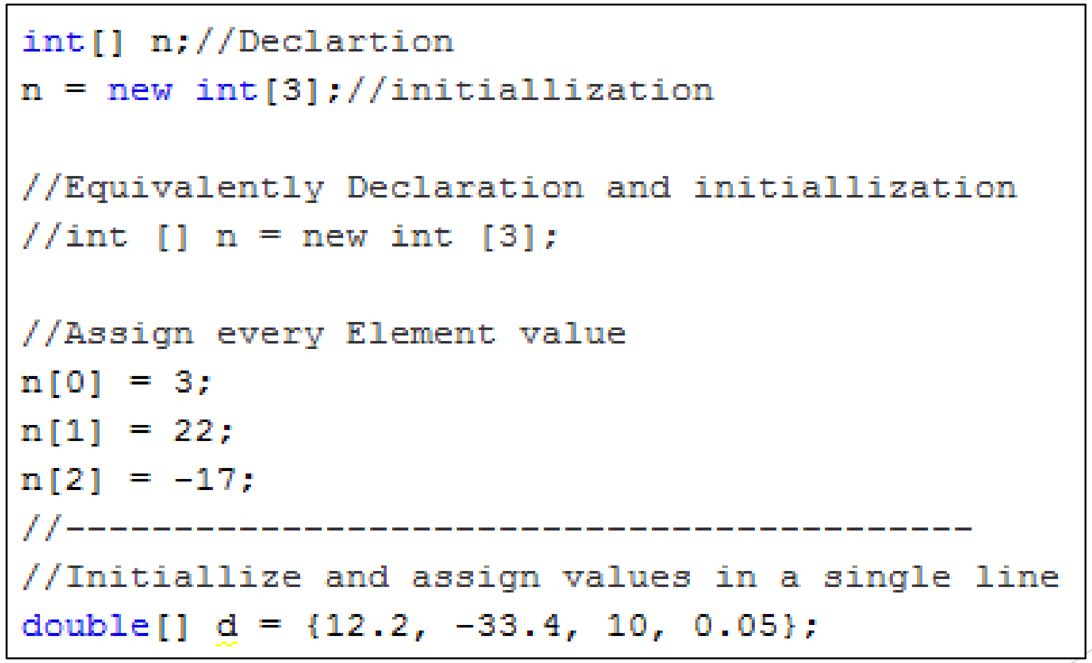
- Using the key word `final`
  - Example : `final int num = 5;`
- `No #define in Java`

Math Functions

- Using Math class
- Some Functions
  - sin , cos , tan , sinh , cosh, tanh
  - sqrt , pow , log , log10 , exp
  - abs , floor , ceil

Arrays

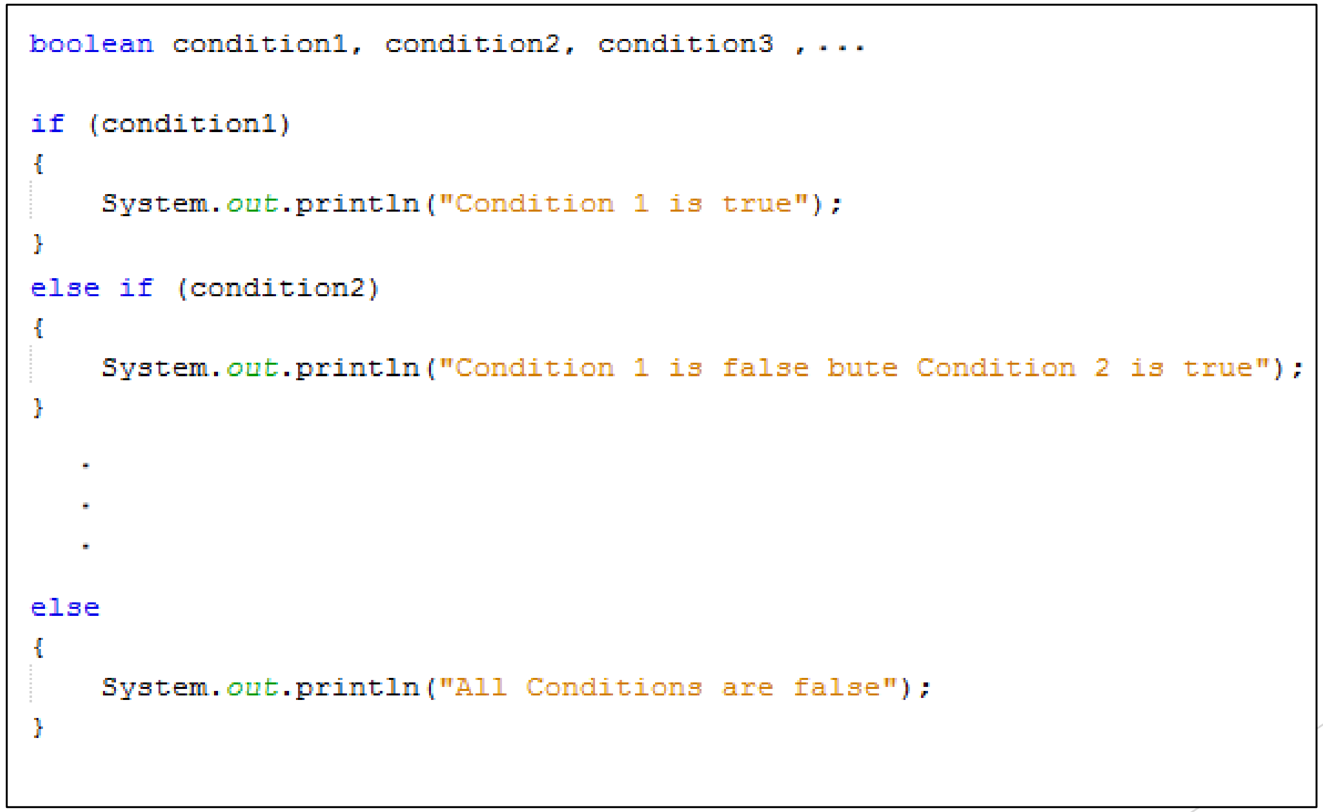
- An *array* is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created.
- After creation, its length is fixed.
  - Arrays can’t grow in size once being initialized



```
int[] n;//Declartion
n = new int[3];//initiallization

//Equivalently Declaration and initiallization
//int [] n = new int [3];

//Assign every Element value
n[0] = 3;
n[1] = 22;
n[2] = -17;
//------------------------------------------
//Initiallize and assign values in a single line
double[] d = {12.2, -33.4, 10, 0.05};
```

IF-statement

- An **if** statement identifies which statement to run based on the value of a **Boolean** expression.
- If-else statement syntax :



```
boolean condition1, condition2, condition3 ,...

if (condition1)
{
    System.out.println("Condition 1 is true");
}
else if (condition2)
{
    System.out.println("Condition 1 is false bute Condition 2 is true");
}

   .
   .
   .

else
{
    System.out.println("All Conditions are false");
}
```

Conditional Operator

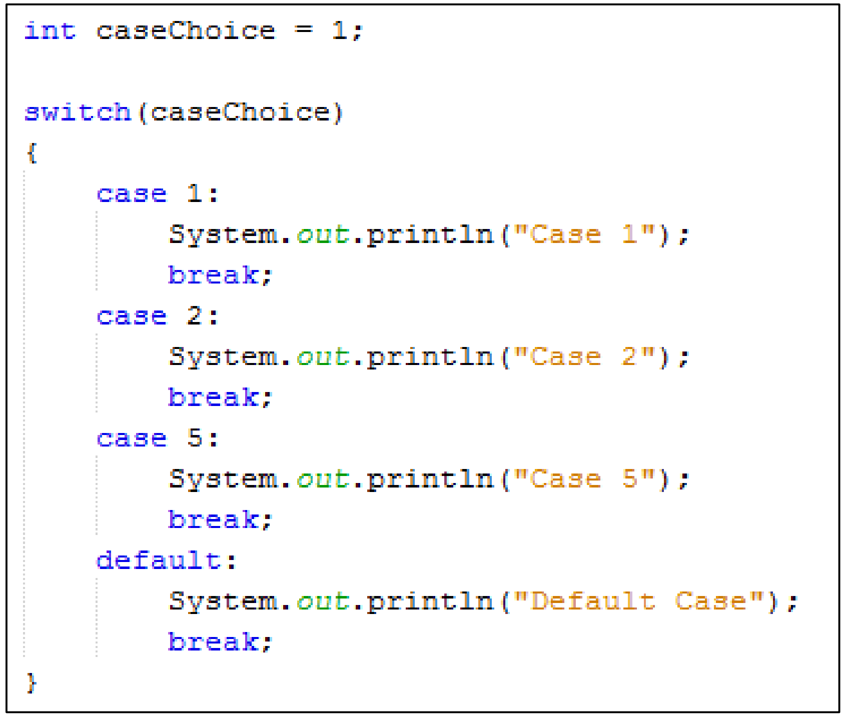
- Ternary operator - "?"

```
//without using ternary operator
int a = 5; int b = 6; boolean larger;
if (a > b)
{
 larger = true;
}
else
{
 larger = false;
}
```

```
//using ternary operator
int a = 5; int b = 6;
boolean larger;
larger = (a > b) ?  true:false;
```

Switch statement

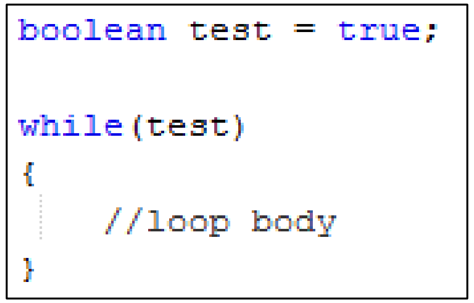
- The **switch** statement is a control statement that selects a *switch section* to execute from a list of candidates.
- The switch statement syntax :



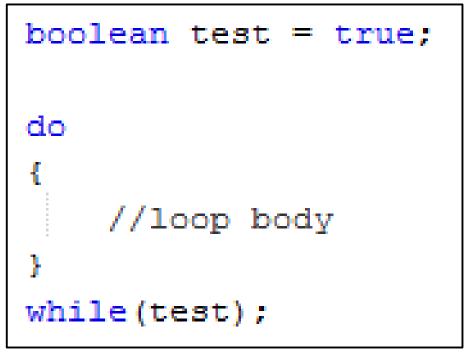
```
int caseChoice = 1;

switch(caseChoice)
{
    case 1:
        System.out.println("Case 1");
        break;
    case 2:
        System.out.println("Case 2");
        break;
    case 5:
        System.out.println("Case 5");
        break;
    default:
        System.out.println("Default Case");
        break;
}
```

While vs. do While

- While Loop
  - Checks the condition first
  - If the condition is true ,statements within loop body are executed
  - Repeat as long as the condition is true
- Do while Loop
  - Checks the condition at the end of the loop
  - The statements within the loop body are executed at least once



```
boolean test = true;

while(test)
{
    //loop body
}
```



```
boolean test = true;

do
{
    //loop body
}
while(test);
```

For and For each

- For loop is very similar to a while loop, in while loop we put the initialization in one place the check in another place and variable modifying in another place. The for loop has all these in one place.

```
for (int i = 0; i < 10; i++)
  /*(initialization ; check ; variable modification)*/
{
    //loop body
}
```

- A for each loop is used to iterate through the items of a collection "ex. Items of an array"

```
int[] numbers = {1,2,3,4,5};

for(int num : numbers)
{
    System.out.println(num);
}
```

Methods

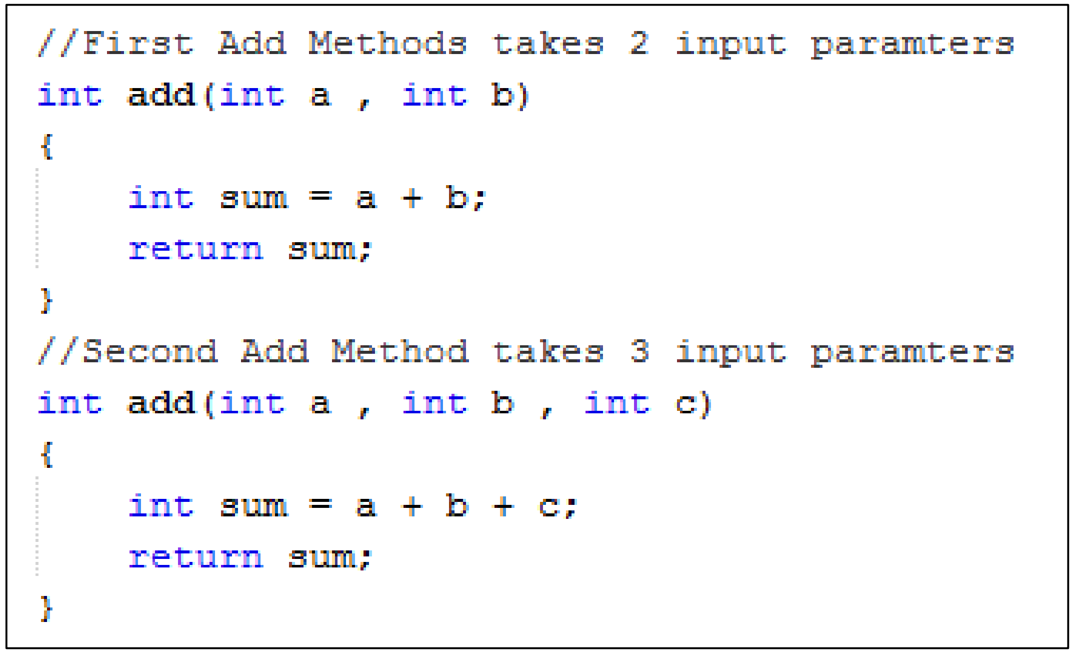
- Functions are called Methods “both terms are used interchangeably”.
- Methods are extremely useful as they allow you to define the logic once and use it in many places.
- Method Syntax :-

```
return-type method_Name(parameters)
 {
     //Method Body
 }
```

  - Return-type can be any valid type or void
  - Method_Name can be any meaningful name
  - Parameters are optional
- Data passed from the caller to the callee through arguments is passed by **value.**

Methods overloading

- Methods overloading refers to defining multiple versions of the same method each having different parameters.



```
//First Add Methods takes 2 input paramters
int add(int a , int b)
{
    int sum = a + b;
    return sum;
}
//Second Add Method takes 3 input paramters
int add(int a , int b , int c)
{
    int sum = a + b + c;
    return sum;
}
```

Recursive Methods

- A recursive method is a function that calls itself.
- When writing recursive method one has to be careful to make sure the program will eventually terminate.
  - Using a conditional statement checking for a well known **stopping condition**.
  - Making sure that the recursive call uses **different parameter values**.

Practice questions

1. Write a program that prints the area and perimeter of a shape of the following geometric shapes.
   a) A rectangle its width = 5 and height = 6
   b) A square it’s side length = 4
   c) A circle it’s radius = 8.5.
2. Using nested loops, write a program that prints the character ‘*’repeatedly forming the triangular shape shown below.

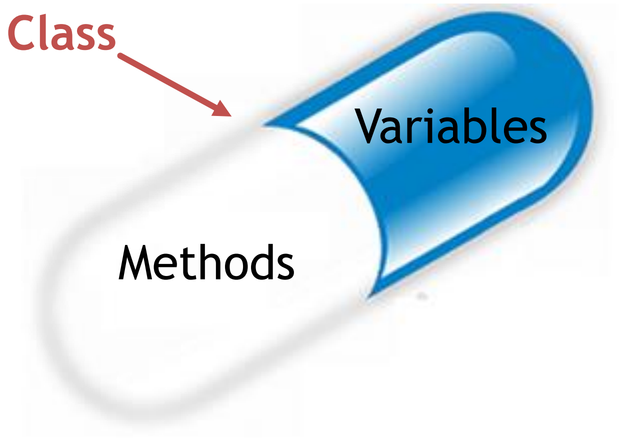
```
*
*  *
*  *  *
*  *  *  *
*  *  *  *  *
```

3. Write a method that computes the sum of an array of integers “use for each”
4. Write another version of sum method in ex.3 using recursion

OOP Basics

Classes and Encapsulation

- So far we have seen simple data types like int , float, double ,…
- If we want to create a complex custom type we will use classes
- A class Encapsulates data “fields” and behavior “Methods”

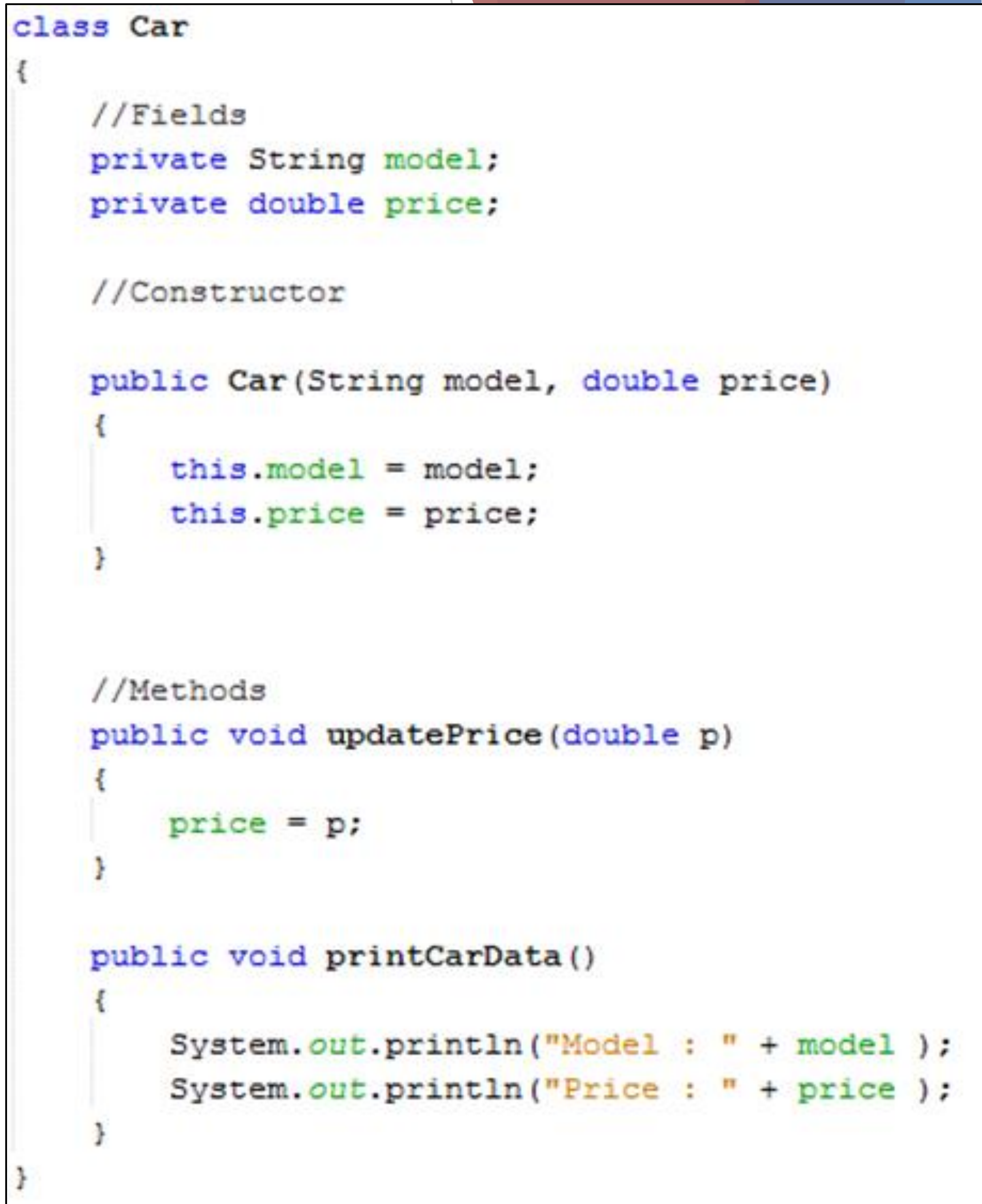


Classes and Objects

- A class is like a **template** that has several members (fields or methods)
- An object is a **variable/instance** of certain class definition
  - Example
    - `int x; // x is a variable of type int “primitive data type”`
    - `Car c;//c is a variable of type Car “Car is a custom class”`

Example : Car Class

- Our First Java Class.
- This Class Describes a Car.
  - Car Model.
  - Car Price.



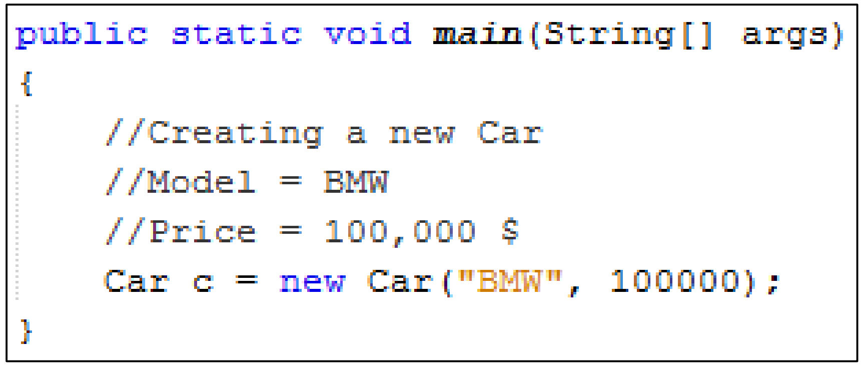
```
class Car
{
    //Fields
    private String model;
    private double price;

    //Constructor

    public Car(String model, double price)
    {
        this.model = model;
        this.price = price;
    }


    //Methods
    public void updatePrice(double p)
    {
        price = p;
    }

    public void printCarData()
    {
        System.out.println("Model : " + model );
        System.out.println("Price : " + price );
    }
}
```

Constructor

- The purpose of the constructor is to initialize object fields.
- The class constructor automatically called when an instance of a class is created.
- Constructors have no return values.
- Constructors have the same name as the Class.
- Constructors are not mandatory , if no constructor is provided a default constructor is automatically provided.
- Constructor can be overloaded.

Creating an Object of a Class

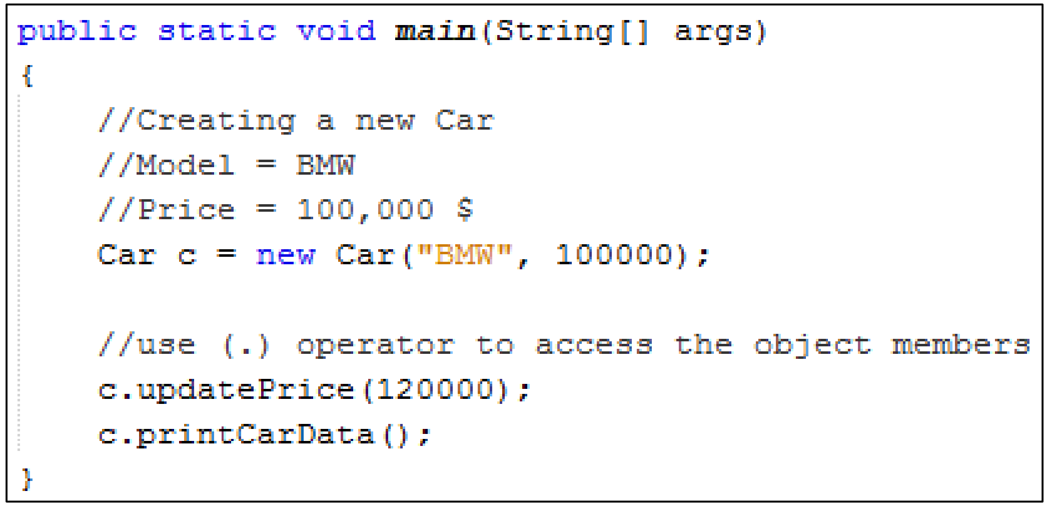
- Objects are created using the **new** keyword followed by a call to the class constructor.
- The Object creation returns a **reference** to the created object.
- All Java objects are allocated in the heap and automatically deleted by Java’s garbage collector when no longer accessible.



```
public static void main(String[] args)
{
    //Creating a new Car
    //Model = BMW
    //Price = 100,000 $
    Car c = new Car("BMW", 100000);
}
```

Using Objects

- After an object is created, accessible members are used via the dot operator.



```java
public static void main(String[] args)
{
    //Creating a new Car
    //Model = BMW
    //Price = 100,000 $
    Car c = new Car("BMW", 100000);

    //use (.) operator to access the object members
    c.updatePrice(120000);
    c.printCarData();
}
```

Access Modifiers

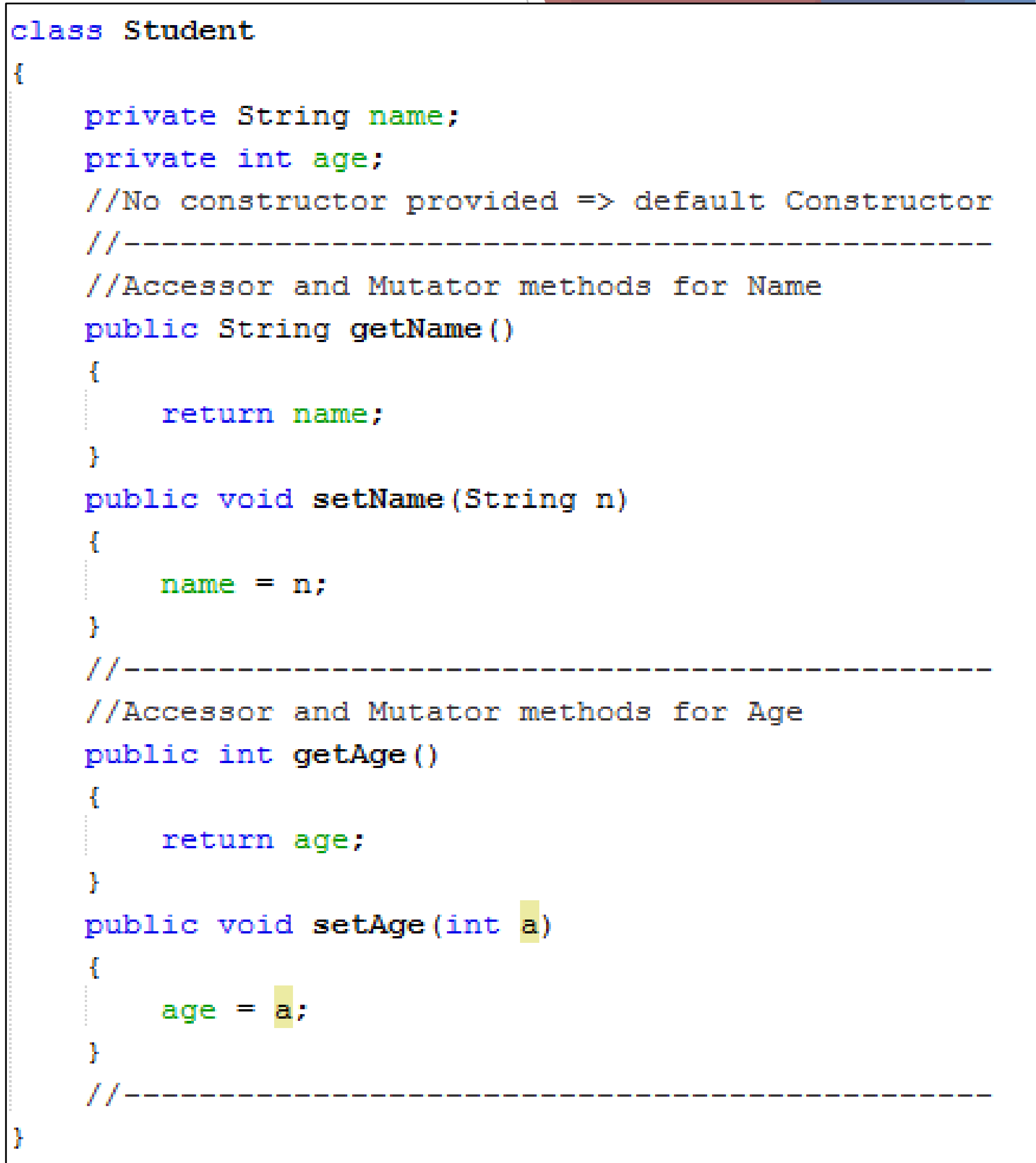
- Each member of the class “Field or method” must have it’s scope determined using an access modifier :
  - Visible to the package. the default (none).
  - Visible to the class only (private).
  - Visible to the world (public).
  - Visible to the package and all subclasses (protected).

- Typically fields are hidden while methods aren’t.

Accessor and Mutator

- Making a class field public is a bad practice as we will have no control over the assigned values , so fields should be private.
- Private fields are usually accessed via **Accessor** and **Mutator** methods.
- Accessor methods are typically public methods that do not take any parameters and return the value of a field.
- Mutator methods are typically public methods that take one or more parameters used to set the value of one or more fields(usually after applying some checks).
- Also Called as Getters and Setters.

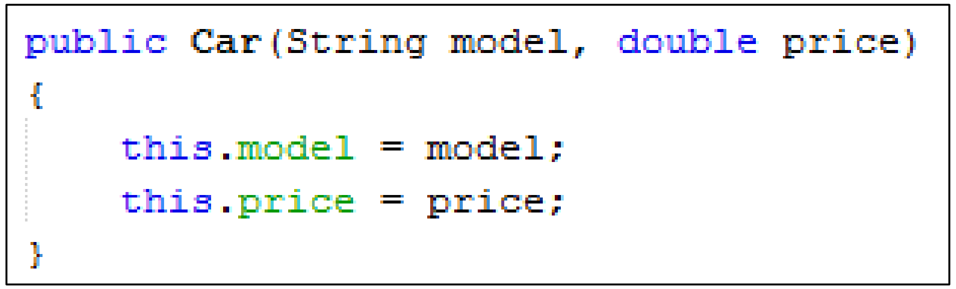
Accessor and Mutator



```
class Student
{
    private String name;
    private int age;
    //No constructor provided => default Constructor
    //------------------------------------------------
    //Accessor and Mutator methods for Name
    public String getName()
    {
        return name;
    }
    public void setName(String n)
    {
        name = n;
    }
    //------------------------------------------------
    //Accessor and Mutator methods for Age
    public int getAge()
    {
        return age;
    }
    public void setAge(int a)
    {
        age = a;
    }
    //------------------------------------------------
}
```

This Key Word

- The **this** keyword refers to the current object of the class .



```
public Car(String model, double price)
{
    this.model = model;
    this.price = price;
}
```

Practice Questions

- **Vector class**
- Define a class that represents a 2D Vector.
- The class should encapsulate both a X Component and an Y Component .
- The class should have an empty constructor initializing them to zeroes and a non-empty constructor that initializes them according to arguments.
- The Class should have the following Methods:
  - **double getX()** and **double getY()** : X and Y components accessors.
  - **void setX(double x)** and **void setY(double y)** : X and Y components mutators.
  - **double magnitude()** : returns the magnitude of the current vector.
  - **double angle()** : returns the angle of the current vector.
  - **void print()** : prints the vector in 2 formats **(x i + y j)** and **(Magnitude [angle]).**

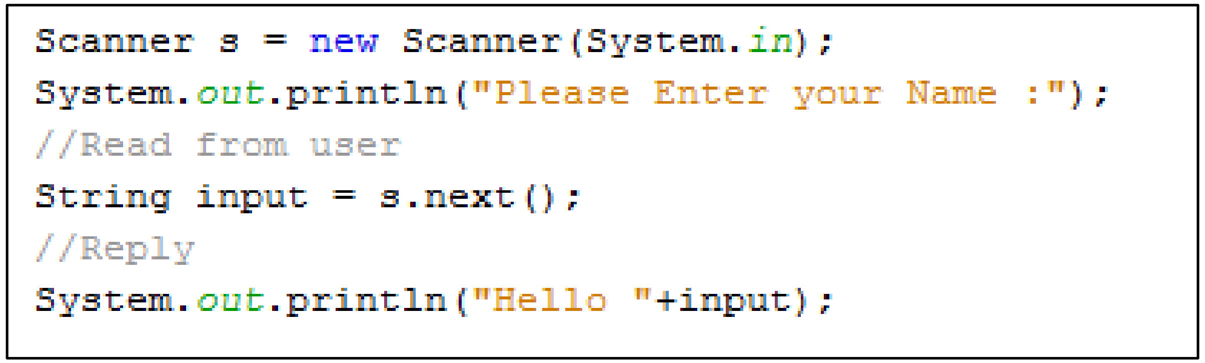
- Write a driver program that tests all the class functionalities

Java APIs

- Java comes with a set of built-in classes that you can use immediately
- Known as JAVA APIs (application program interface)
- Documentation can be found for online browsing at: http://docs.oracle.com/javase/8/docs/api/
- Some useful classes & methods
  - Math
  - String
  - Scanner
  - Arrays
  - System.arraycopy()

java.util.Scanner

- A simple text scanner which can parse primitive types and strings.



```
Scanner s = new Scanner(System.in);
System.out.println("Please Enter your Name :");
//Read from user
String input = s.next();
//Reply
System.out.println("Hello "+input);
```

Static Members

- When a class member includes a static modifier , the member is called a **static member** , while when no static modifier is present the member is called **non static.**
- Static members (either fields or methods) are members that belong to the class itself, not to objects.
- Static members can be accessed without creating any objects.
- Non-static members to be accessed require an object to be created.
- Static methods cannot access non-static members (fields or methods).
- <u>This</u> key word cannot be used inside a static method.

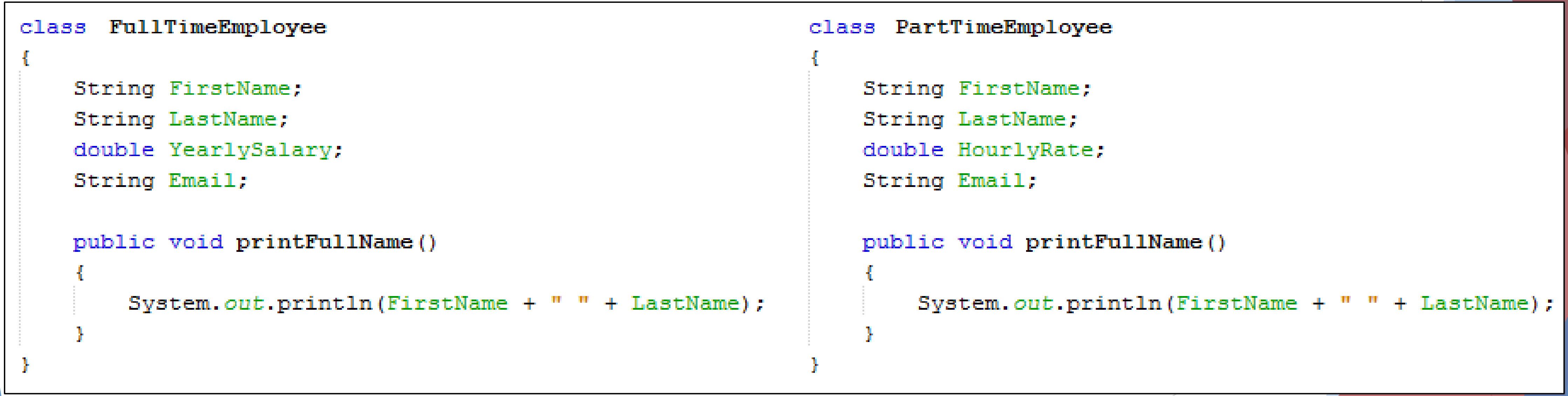
Practice Questions

- **Vector class (version 2)**
- Modify the Vector class you wrote before adding the following Methods:
  - **void read() :** sets the X and Y components of the current Vector based on user provided values.
  - **static Vector add(Vector v1 , Vector v2) :** returns a vector that represents the result of the addition of vectors v1 and v2.
  - **static Vector sub(Vector v1 , Vector v2) :** returns a vector that represents the result of the subtraction of vectors v1 and v2.
- Write a driver program that tests all the class functionalities

OOP Inheritance

Inheritance

- You can define a class that inherits everything (fields and methods) from another class.
- Why inheritance : Useful to avoid code redundancy
  - Example :
    A lot of code is duplicated
    between these 2 classes



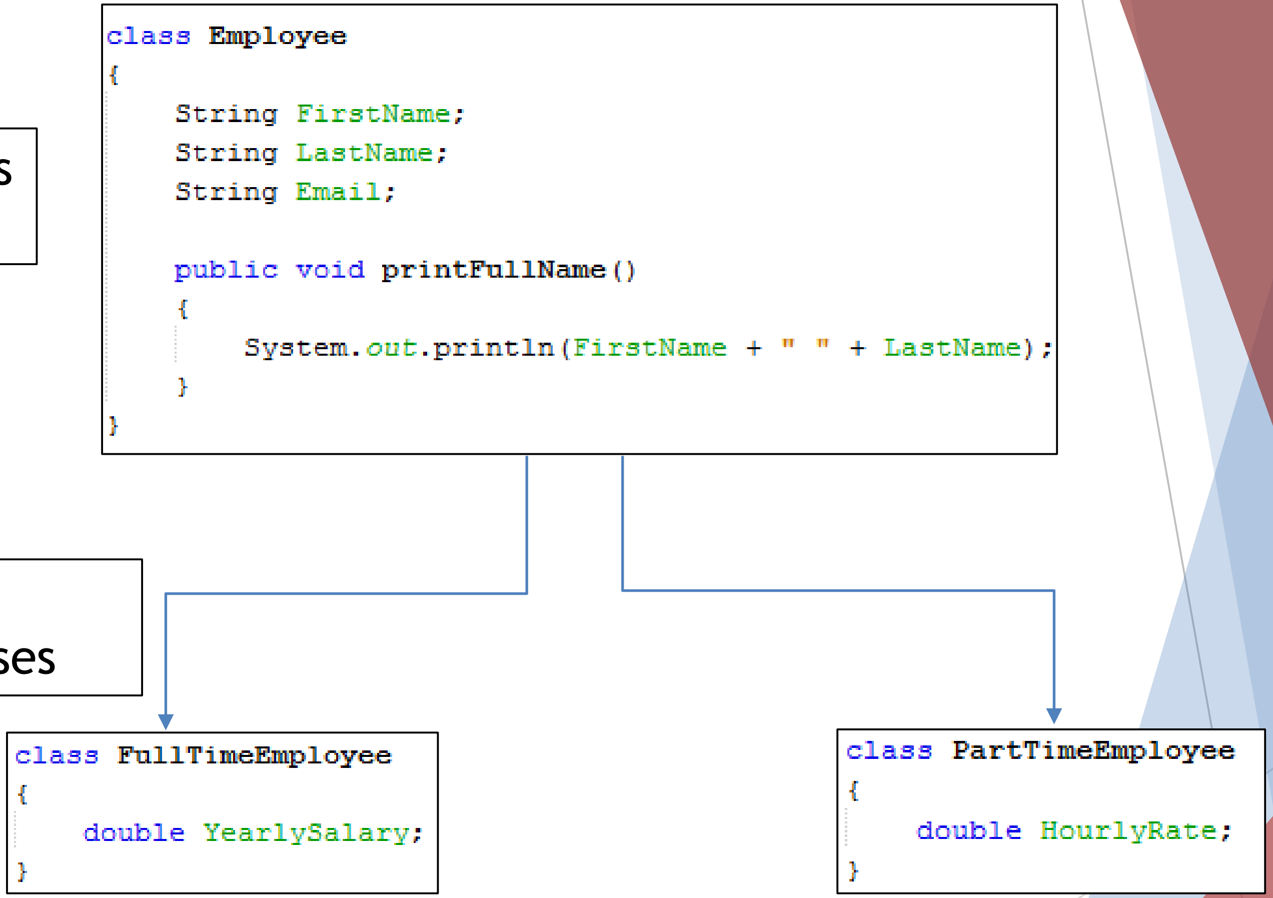
```java
class  FullTimeEmployee
{
    String FirstName;
    String LastName;
    double YearlySalary;
    String Email;

    public void printFullName()
    {
        System.out.println(FirstName + " " + LastName);
    }
}
```

```java
class  PartTimeEmployee
{
    String FirstName;
    String LastName;
    double HourlyRate;
    String Email;

    public void printFullName()
    {
        System.out.println(FirstName + " " + LastName);
    }
}
```

Inheritance

Move all common code to Employee Class
"Parent Class"

Move FullTime and PartTime Employee specific code in the respective child classes



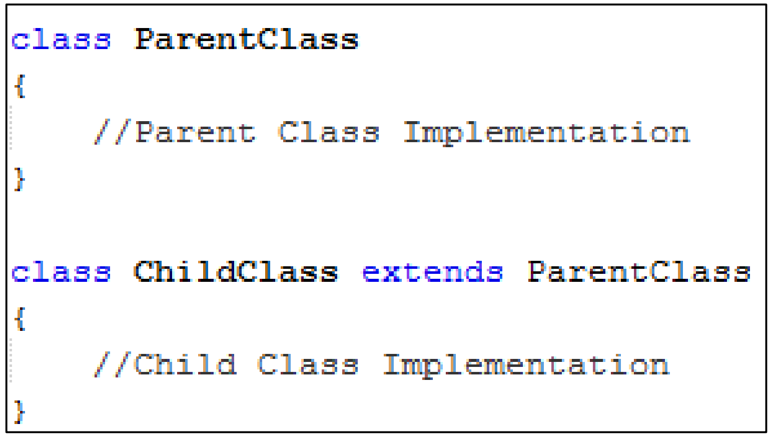
```
class Employee
{
    String FirstName;
    String LastName;
    String Email;

    public void printFullName()
    {
        System.out.println(FirstName + " " + LastName);
    }
}
```

```
class FullTimeEmployee
{
    double YearlySalary;
}
```

```
class PartTimeEmployee
{
    double HourlyRate;
}
```

Inheritance

- To create a child class (aka derived class or subclass) inheriting from a parent class (aka base class or superclass), Java uses the **extends** keyword
- In this example ChildClass inherits from ParentClass.



```
class ParentClass
{
    //Parent Class Implementation
}

class ChildClass extends ParentClass
{
    //Child Class Implementation
}
```

- Child class is a specialization of parent class.
- Parent class is automatically instantiated before child class.
- Parent class constructor executes before child class constructor.

Inheritance

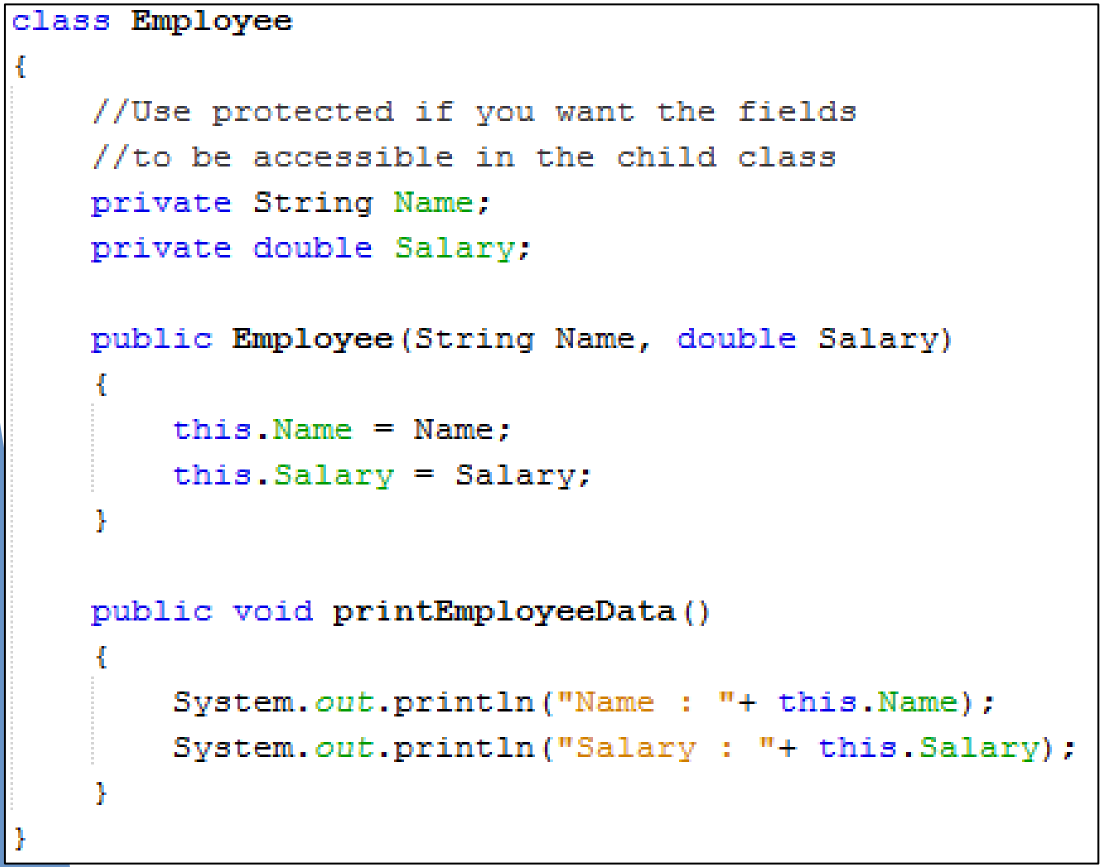
- Notes
  - You can cast the child object to parent type <u>“and vise versa”</u> using cast operator “()”.
  - There is no limit on inheritance depth.
  - Java supports only single class inheritance
    (a class can only inherit from a single class)

Class Object

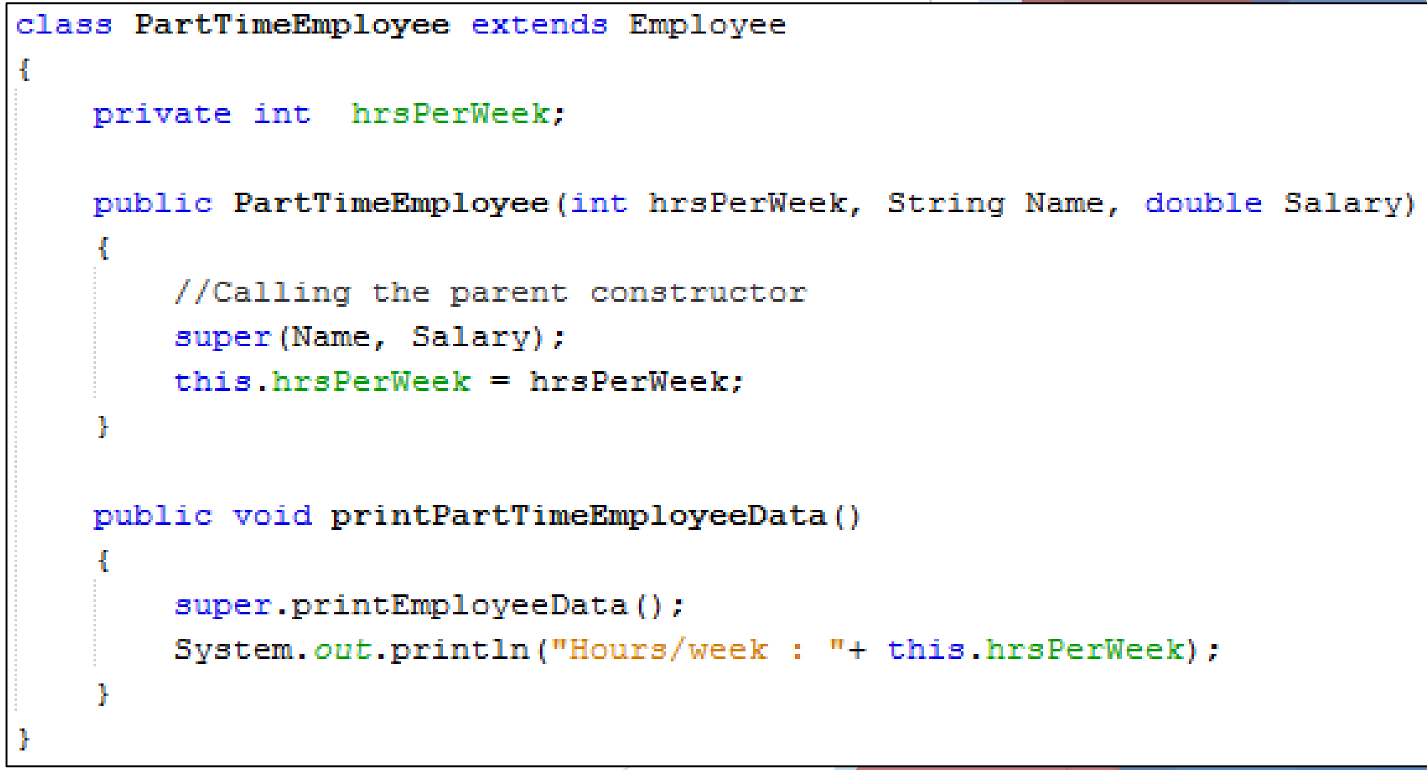
- class **Object** is the cosmic superclass
- All methods in class Object are automatically inherited by all classes.
  - Ex: toString(),equals()

Calling Parent Class members

- Use **super** key word.



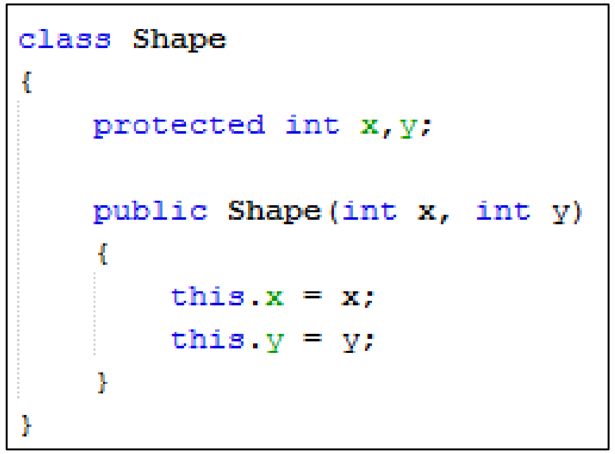
```java
class Employee
{
    //Use protected if you want the fields
    //to be accessible in the child class
    private String Name;
    private double Salary;

    public Employee(String Name, double Salary)
    {
        this.Name = Name;
        this.Salary = Salary;
    }

    public void printEmployeeData()
    {
        System.out.println("Name : "+ this.Name);
        System.out.println("Salary : "+ this.Salary);
    }
}
```



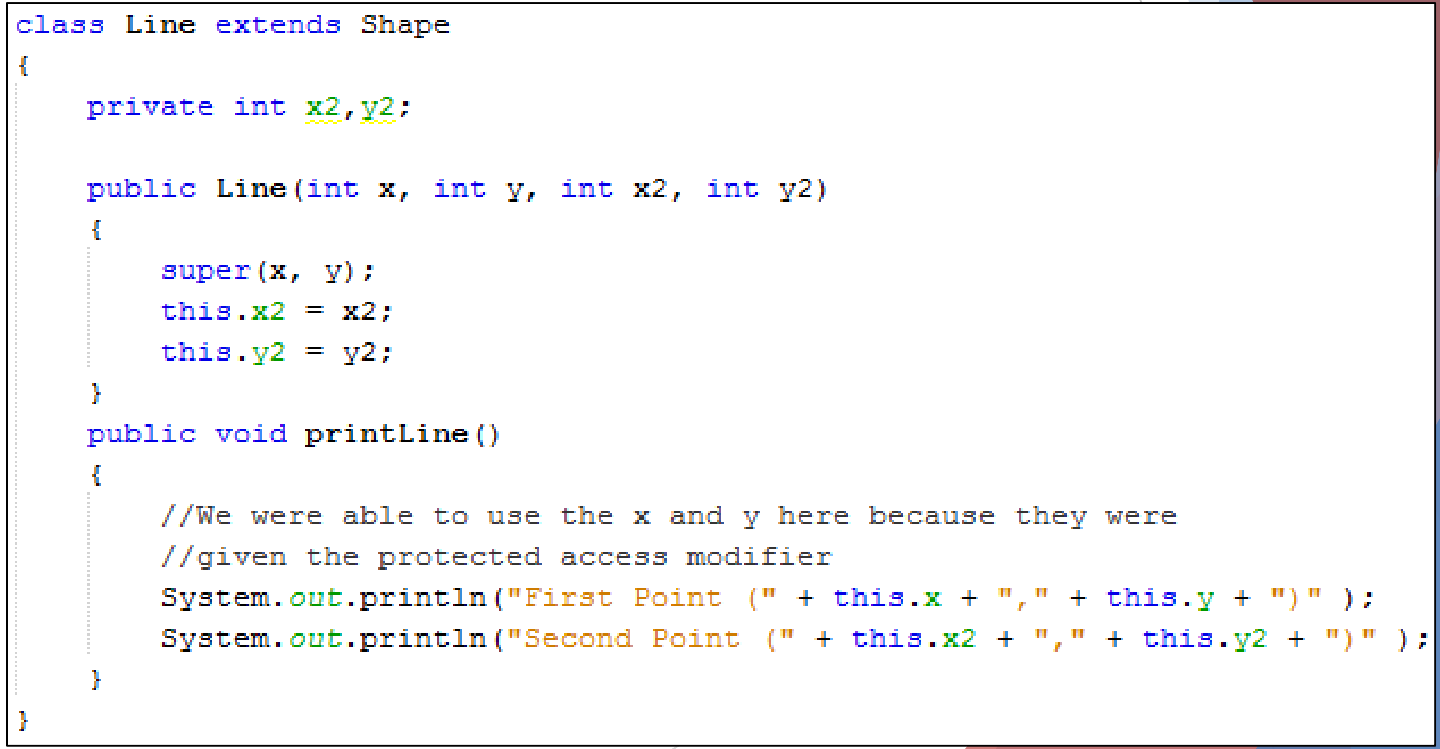
```java
class PartTimeEmployee extends Employee
{
    private int  hrsPerWeek;

    public PartTimeEmployee(int hrsPerWeek, String Name, double Salary)
    {
        //Calling the parent constructor
        super(Name, Salary);
        this.hrsPerWeek = hrsPerWeek;
    }

    public void printPartTimeEmployeeData()
    {
        super.printEmployeeData();
        System.out.println("Hours/week : "+ this.hrsPerWeek);
    }
}
```

Protected Access Modifier

- Private Fields/Methods in the Parent Class aren’t Accessible from the child class.
- If You want a Field/method to be Accessible to all subclasses then you must use the (protected) access modifier.



```
class Shape
{
    protected int x,y;

    public Shape(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```



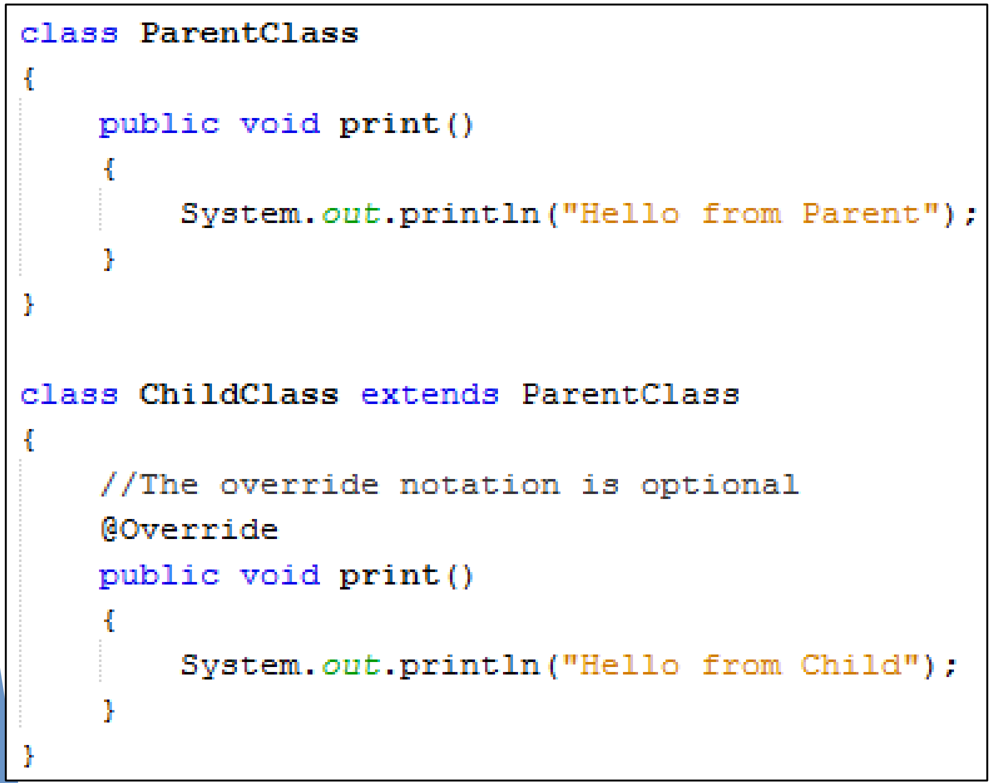
```
class Line extends Shape
{
    private int x2,y2;

    public Line(int x, int y, int x2, int y2)
    {
        super(x, y);
        this.x2 = x2;
        this.y2 = y2;
    }
    public void printLine()
    {
        //We were able to use the x and y here because they were
        //given the protected access modifier
        System.out.println("First Point (" + this.x + "," + this.y + ")" );
        System.out.println("Second Point (" + this.x2 + "," + this.y2 + ")" );
    }
}
```

Practice Questions

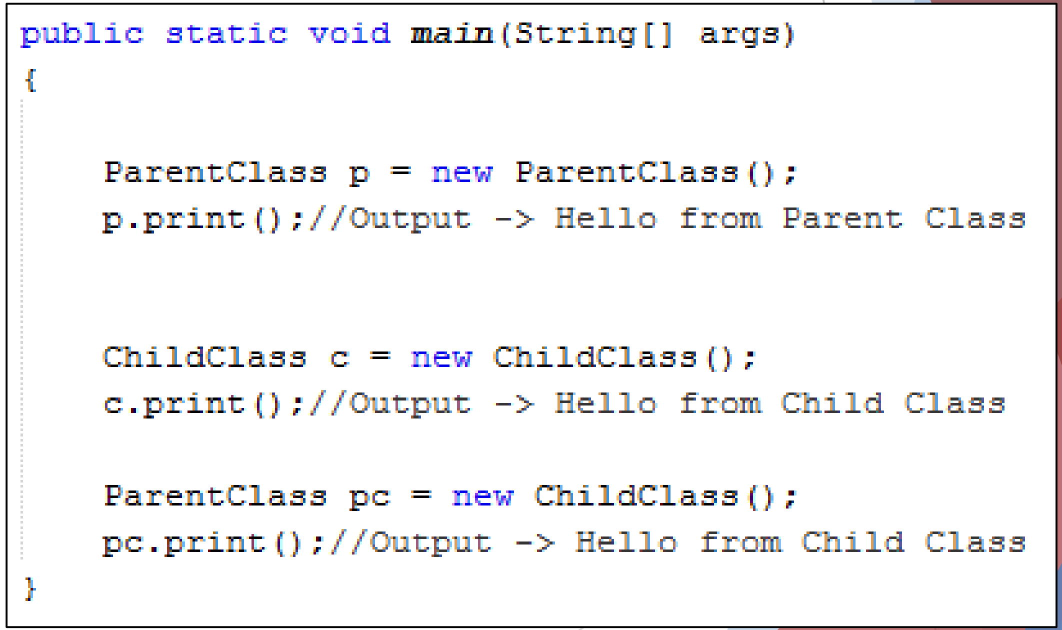
- **Library**
  - Assume a library keeps two types of publications: books and audio CDs.
  - For a book, we need to keep track of the ID (int) ,title(String), price(float), and number of pages(int).
  - For a CD, we need to keep track of the ID (int) ,title (String), price (float), and the length in minutes (float).
  - The ID is just a serial number initialized at construction time. This means that the first publication object constructed should have id= 1and the second one id= 2, and so on.
  - **Define the appropriate classes.**
  - All the classes should provide a print method as well as accessor methods for all their fields.
- Write a driver program that asks the user for the number of publications he wants to provide, creates an array of publications of the appropriate size, and then reads the data of every publication from the user. When the user finishes the program should list all the entered publications.

Method overriding

- Base class reference pointing at child class object will invoke the overridden method in child class .



```
class ParentClass
{
    public void print()
    {
        System.out.println("Hello from Parent");
    }
}

class ChildClass extends ParentClass
{
    //The override notation is optional
    @Override
    public void print()
    {
        System.out.println("Hello from Child");
    }
}
```



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

    ParentClass p = new ParentClass();
    p.print();//Output -> Hello from Parent Class


    ChildClass c = new ChildClass();
    c.print();//Output -> Hello from Child Class

    ParentClass pc = new ChildClass();
    pc.print();//Output -> Hello from Child Class
}
```

Polymorphism

- Polymorphism literally means “Many Forms”
- Polymorphism happens when we have a reference of a parent type pointing to an object of a child type.
- Polymorphism allows such statements to be considered correct
  - Shape s = new Rectangle(...);
- Polymorphism allow invoking child class methods through a parent class reference during run time.
- Use method overriding to allow polymorphism.

Practice Questions

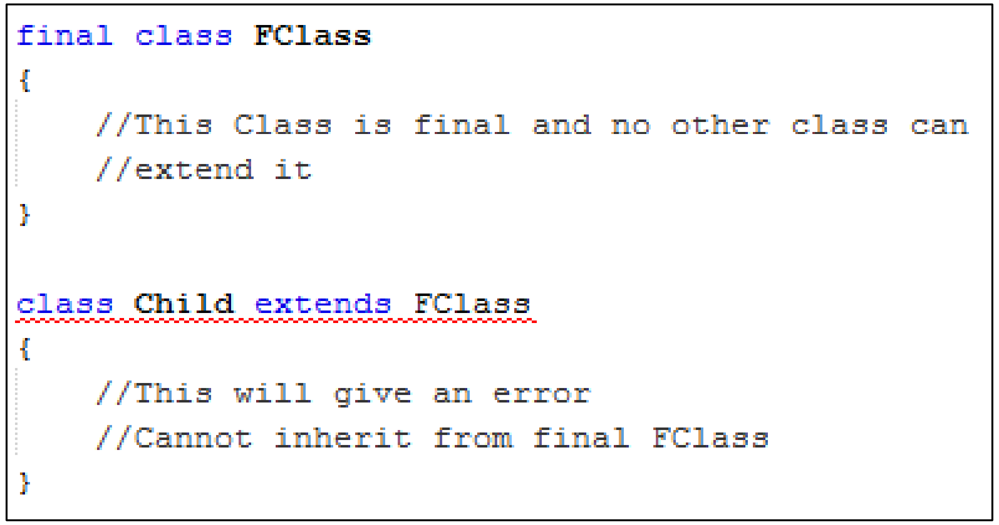
Library (Version 2)

- Modify the Library Program class you wrote before using method overriding and polymorphism:
  - Override all print methods in all classes [use method overriding here].
- Write a driver program that asks the user for the number of publications he wants to provide, creates an array of publications of the appropriate size, and then reads the data of every publication from the user. When the user finishes the program should list all the entered publications[use polymorphism here].

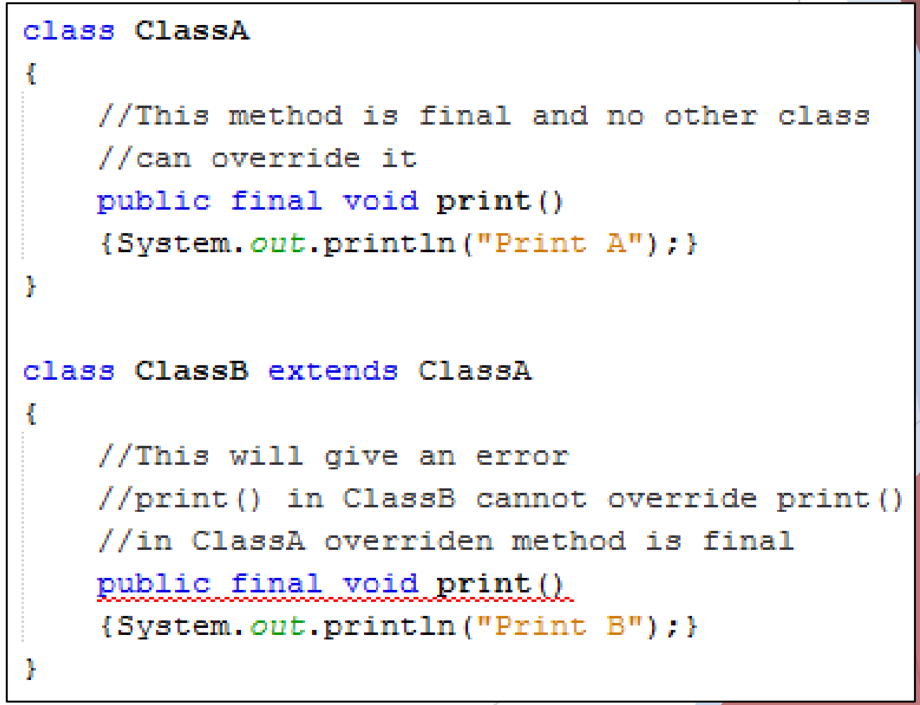
OOP Advanced

Final Key word

- Final classes cannot be inherited.
- Final methods cannot be overridden.
- Final variables are variables whose values cannot be changed once assigned.



```java
final class FClass
{
    //This Class is final and no other class can
    //extend it
}

class Child extends FClass
{
    //This will give an error
    //Cannot inherit from final FClass
}
```

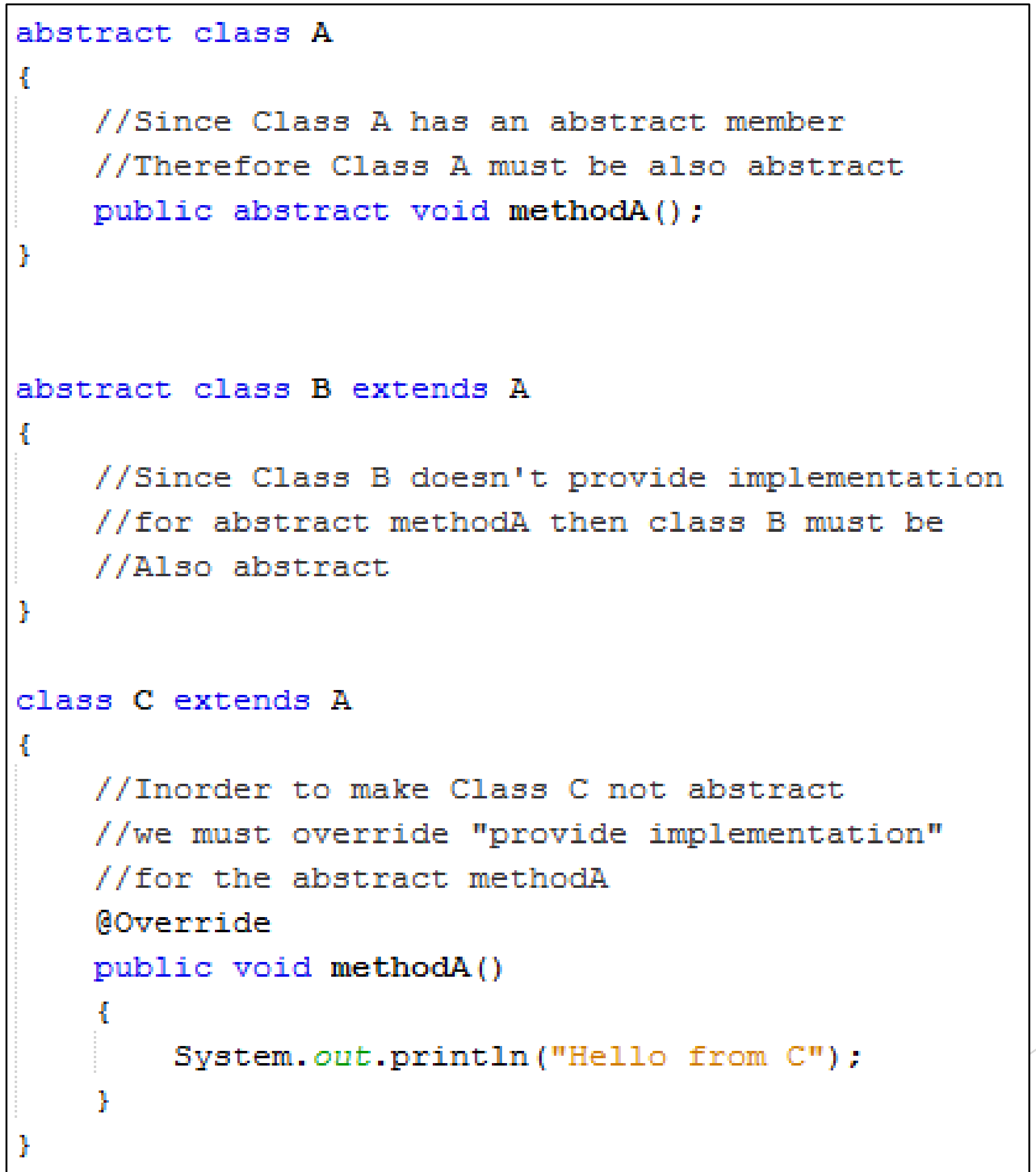


```java
class ClassA
{
    //This method is final and no other class
    //can override it
    public final void print()
    {System.out.println("Print A");}
}

class ClassB extends ClassA
{
    //This will give an error
    //print() in ClassB cannot override print()
    //in ClassA overriden method is final
    public final void print()
    {System.out.println("Print B");}
}
```

Abstract Classes

- Abstract Class is a class that it is not complete so no object can be created from it.
- Abstract class are used only as a base class .
- Abstract classes can’t be Final.
- An Abstract method is a method with no implementation.
- An Abstract class may or may not contain abstract methods.
- A class with an abstract method must be abstract.
- If a class inherits from an abstract class
    - It must overrides all its abstract methods.
    - Or, it can be also abstract

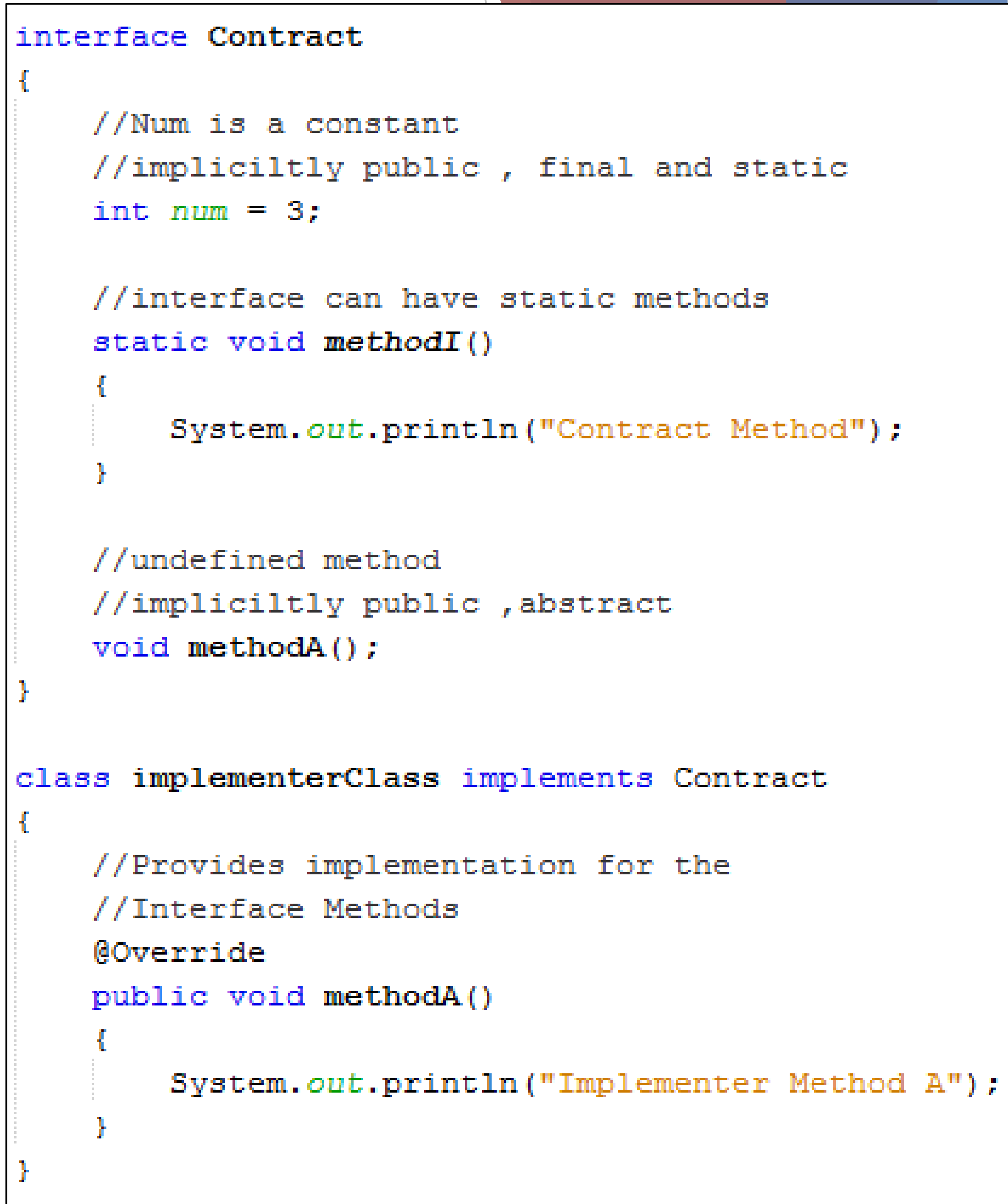
Abstract Classes



```
abstract class A
{
    //Since Class A has an abstract member
    //Therefore Class A must be also abstract
    public abstract void methodA();
}


abstract class B extends A
{
    //Since Class B doesn't provide implementation
    //for abstract methodA then class B must be
    //Also abstract
}

class C extends A
{
    //Inorder to make Class C not abstract
    //we must override "provide implementation"
    //for the abstract methodA
    @Override
    public void methodA()
    {
        System.out.println("Hello from C");
    }
}
```

Interfaces

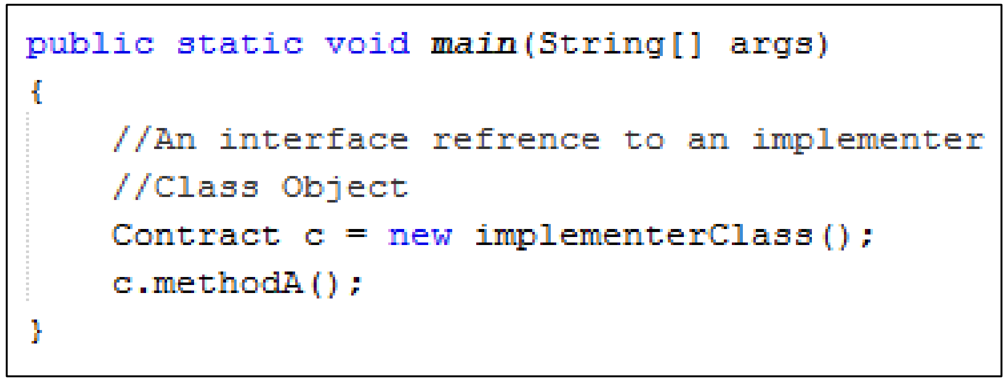
- Interface is a type similar to a class but with only unimplemented (Abstract)methods.
- Beside unimplemented methods an interface may also contain constants and static methods.
  - **Constants** are implicitly consider public, static, and final.
  - **methods** are implicitly considered public and abstract.
- A class that inherits from **(implements)** an interface must provide implementation for all interface methods.
- **Interfaces are like contracts about functionality that implementing classes must support.**



```
interface Contract
{
    //Num is a constant
    //impliciltly public , final and static
    int num = 3;

    //interface can have static methods
    static void methodI()
    {
        System.out.println("Contract Method");
    }

    //undefined method
    //impliciltly public ,abstract
    void methodA();
}

class implementerClass implements Contract
{
    //Provides implementation for the
    //Interface Methods
    @Override
    public void methodA()
    {
        System.out.println("Implementer Method A");
    }
}
```

Interfaces

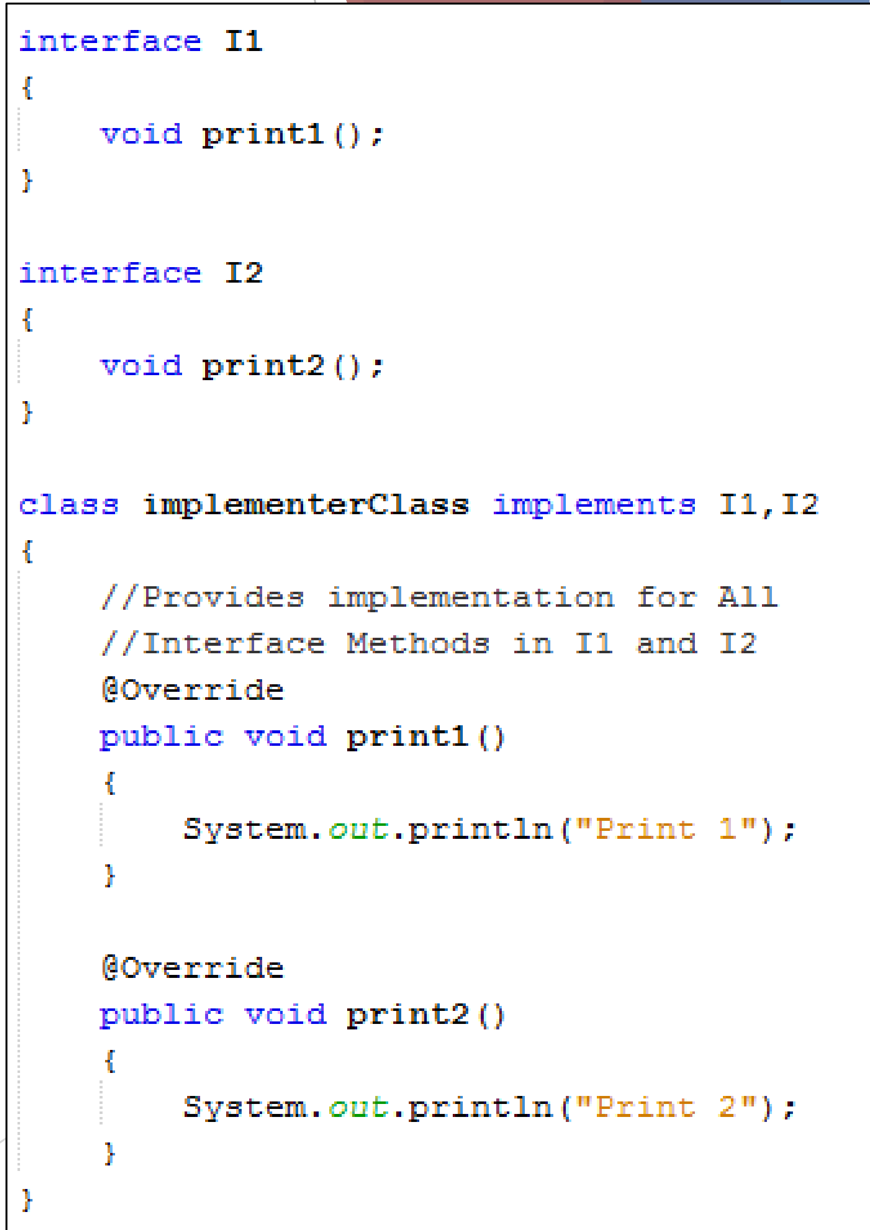
- We can’t create an object from an interface , but we can use an interface reference that points to an object of a class that implements this interface.



```
public static void main(String[] args)
{
    //An interface refrence to an implementer
    //Class Object
    Contract c = new implementerClass();
    c.methodA();
}
```

Interfaces and multiple inheritance

- A class can inherit from one base class only .
- A class can inherit from one or more interfaces at the same time.



```java
interface I1
{
    void print1();
}

interface I2
{
    void print2();
}

class implementerClass implements I1,I2
{
    //Provides implementation for All
    //Interface Methods in I1 and I2
    @Override
    public void print1()
    {
        System.out.println("Print 1");
    }

    @Override
    public void print2()
    {
        System.out.println("Print 2");
    }
}
```

Practice Questions

- Question 1:
  - Assume a Bank has two types of Bank accounts:
    - SavingsAccount : Account with interest rate
    - CheckingAccount : Account with transaction Fees
  - For a SavingsAccount , we need to keep track of Account ID(int), balance(double), and interest rate(double).
  - For a CheckingAccount , we need to keep track of Account ID(int), balance(double), and transaction Fees (double).
  - Define the appropriate classes (one abstract class and two concrete classes).
  - All classes should have a public constructor and should override the toString and Equals (neglect balance in your comparison) methods appropriately.
  - All the classes should provide a print method that uses the toString method.

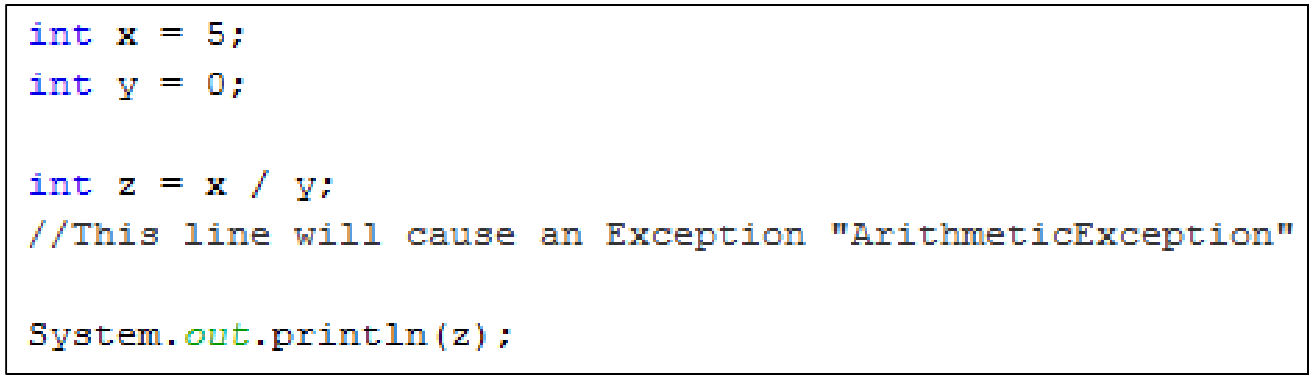
Practice Questions

- Question 2:
  - Create a Class that Represents a Bank
  - A Bank has multiple Bank Accounts (use ArrayList)
  - The Class should have The following methods :
    - Add Account : Add new Account to the bank.
    - Remove Account : remove an account from the bank.
    - Get Account: Checks for an account in the bank **“returns null if not found”**.
    - List All : List all Bank accounts
- Question 3:
  - Write a simple driver program that test the functionality of the Bank Class.

Exception Handling Basics

Exception

- Things can go wrong while executing a Java program.
- An Exception is a problem that arises during the **execution** of a program.
- Examples :
  - Division by Zero
  - Trying to read from a file that doesn’t exist

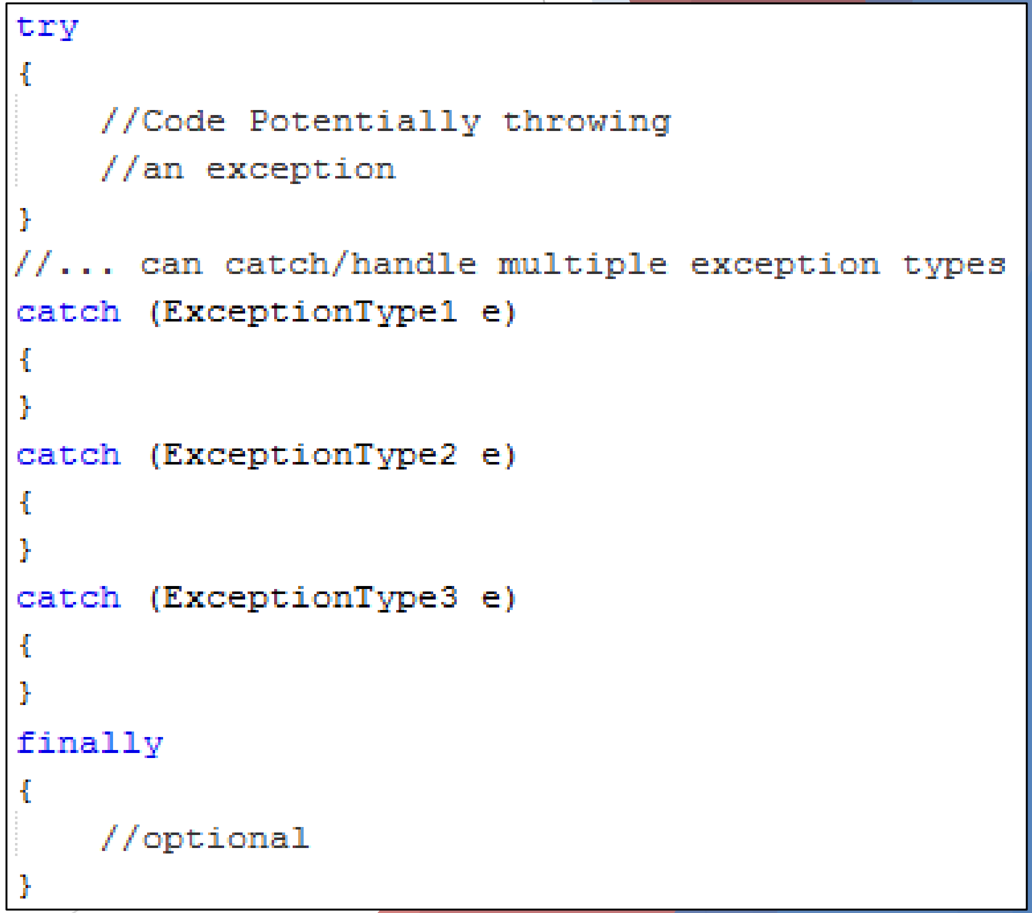


```
int x = 5;
int y = 0;

int z = x / y;
//This line will cause an Exception "ArithmeticException"

System.out.println(z);
```

- When an Exception occurs **(thrown)** the normal flow of the program is disrupted and the program/Application terminates abnormally **(crashes)** which is undesirable.

Exception Handling

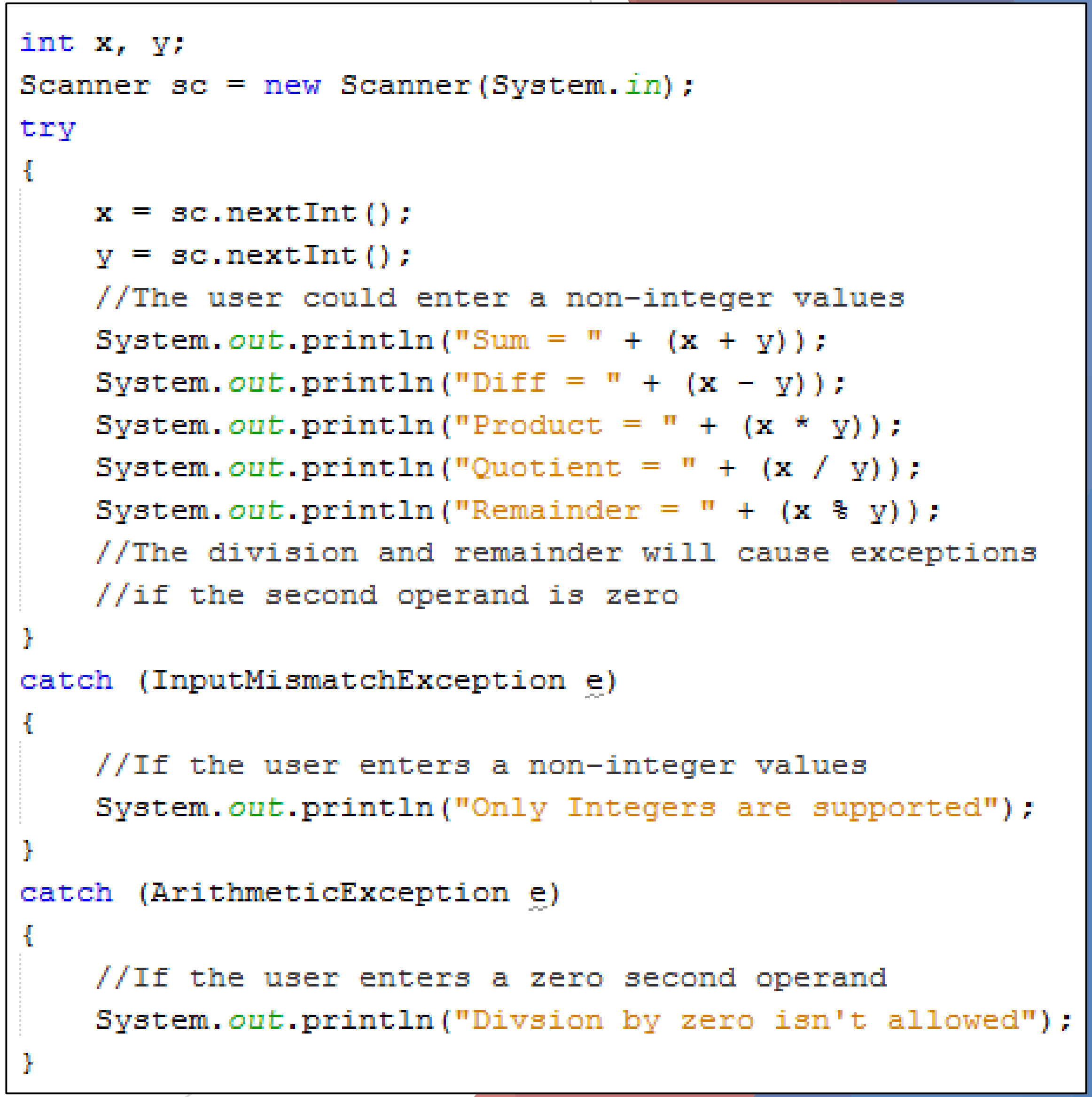
- In order to avoid the program crashing we need to handle**(catch)** any potential exceptions.
- We use the try, catch and finally blocks
  - Try block – The code that can cause an exception.
  - Catch block – code that handles the exceptions
  - Finally block – used to clean and free resources "Optional" Finally block of code always executes, irrespective of occurrence of an Exception



```
try
{
    //Code Potentially throwing
    //an exception
}
//... can catch/handle multiple exception types
catch (ExceptionType1 e)
{
}
catch (ExceptionType2 e)
{
}
catch (ExceptionType3 e)
{
}
finally
{
    //optional
}
```

Exception Handling (Example)

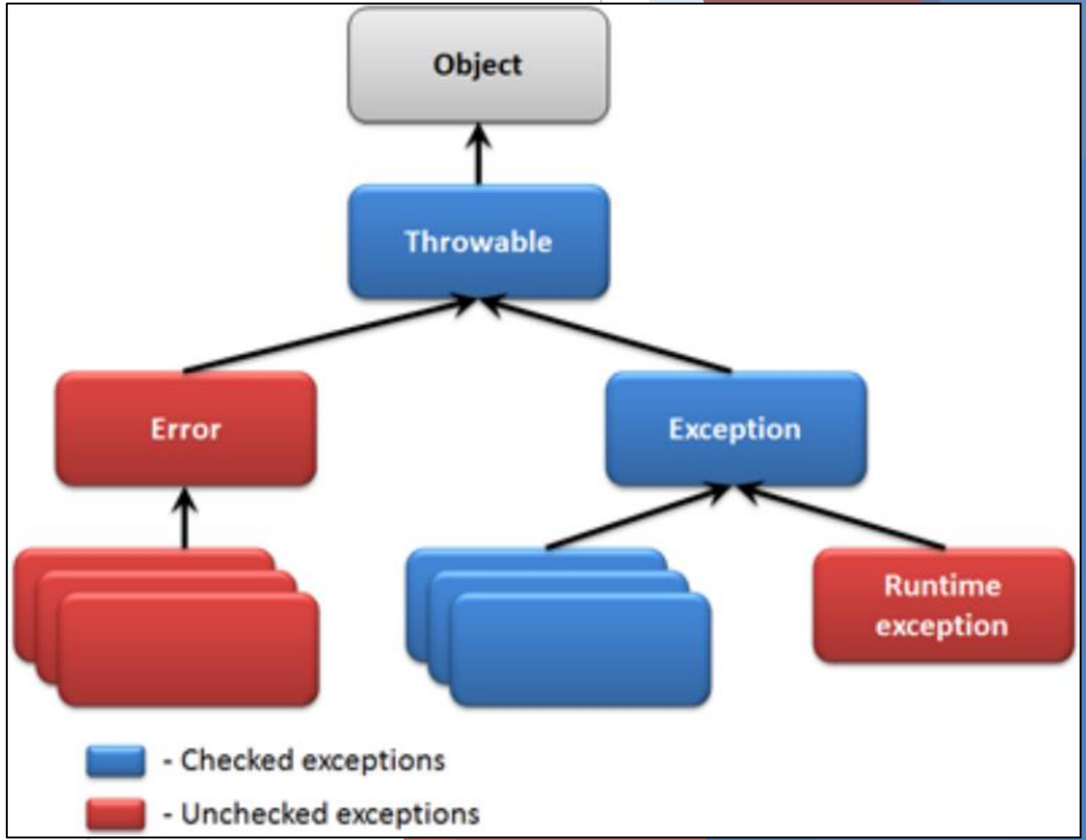
- We want to read two integers from the user, then print their sum, difference, product, quotient, and division remainder.
- The application should use exception handling to verify that the numbers are actually integers .
- Also it should use exception handling to support the case were the second operand is zero.



```java
int x, y;
Scanner sc = new Scanner(System.in);
try
{
    x = sc.nextInt();
    y = sc.nextInt();
    //The user could enter a non-integer values
    System.out.println("Sum = " + (x + y));
    System.out.println("Diff = " + (x - y));
    System.out.println("Product = " + (x * y));
    System.out.println("Quotient = " + (x / y));
    System.out.println("Remainder = " + (x % y));
    //The division and remainder will cause exceptions
    //if the second operand is zero
}
catch (InputMismatchException e)
{
    //If the user enters a non-integer values
    System.out.println("Only Integers are supported");
}
catch (ArithmeticException e)
{
    //If the user enters a zero second operand
    System.out.println("Divsion by zero isn't allowed");
}
```

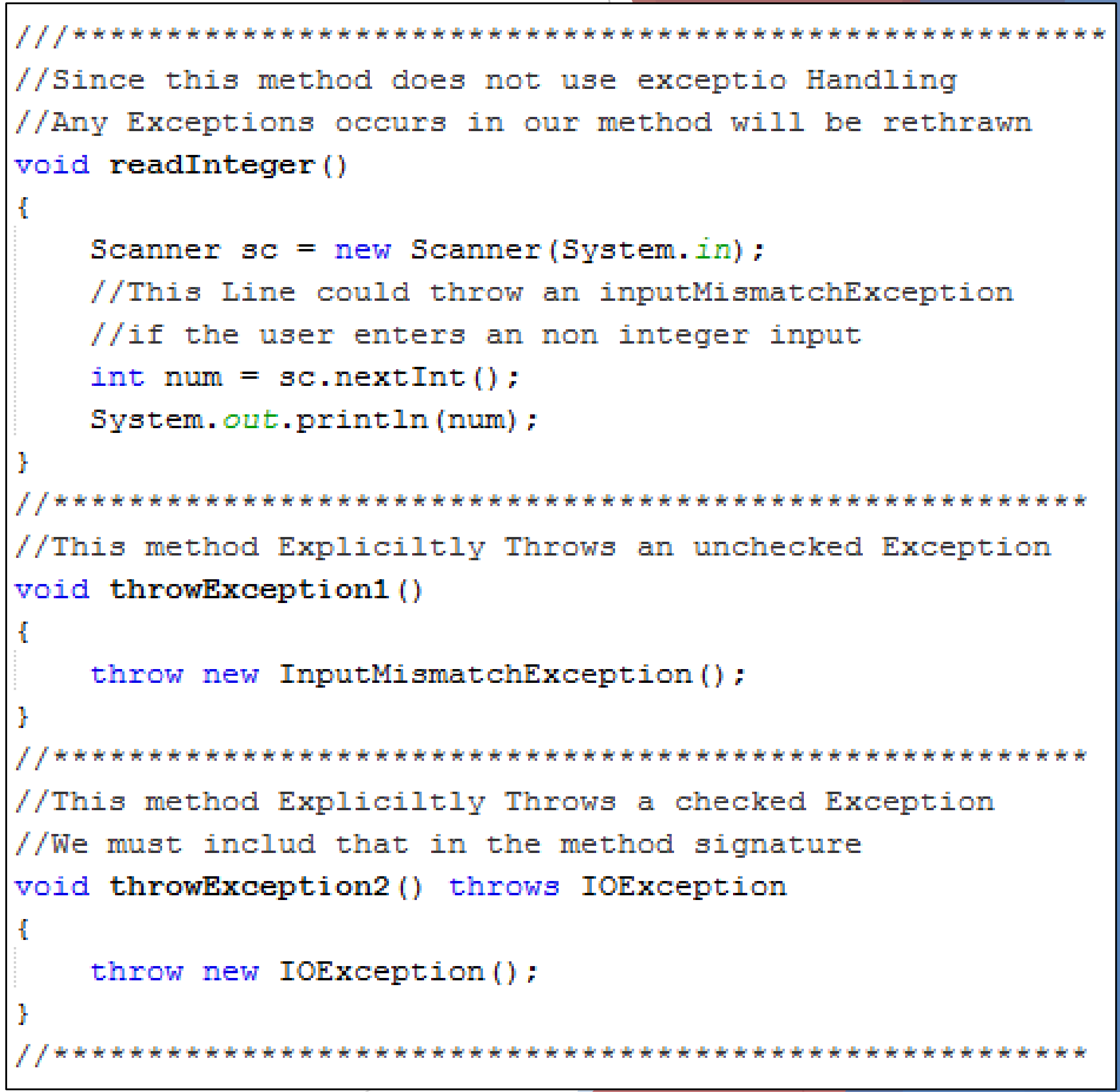
Exception Hierarchy

- All exception classes are subtypes of the java.lang.Exception class.
- **Checked/compile time exception:**
  is an exception that is checked at compile time. If some code within a method throws a checked exception the programmer must handle **(catch)** this exception.
  - **SQLException, IOException, etc..**
- **Unchecked/Runtime exception:**
  is an exceptions that is not checked at compile time. This type of exception are mainly logic errors or improper use of an API. The programmer may ignore the handling of runtime exceptions.
  - NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException, ClassCastException, NumberFormatException, etc…



Throwing Exceptions

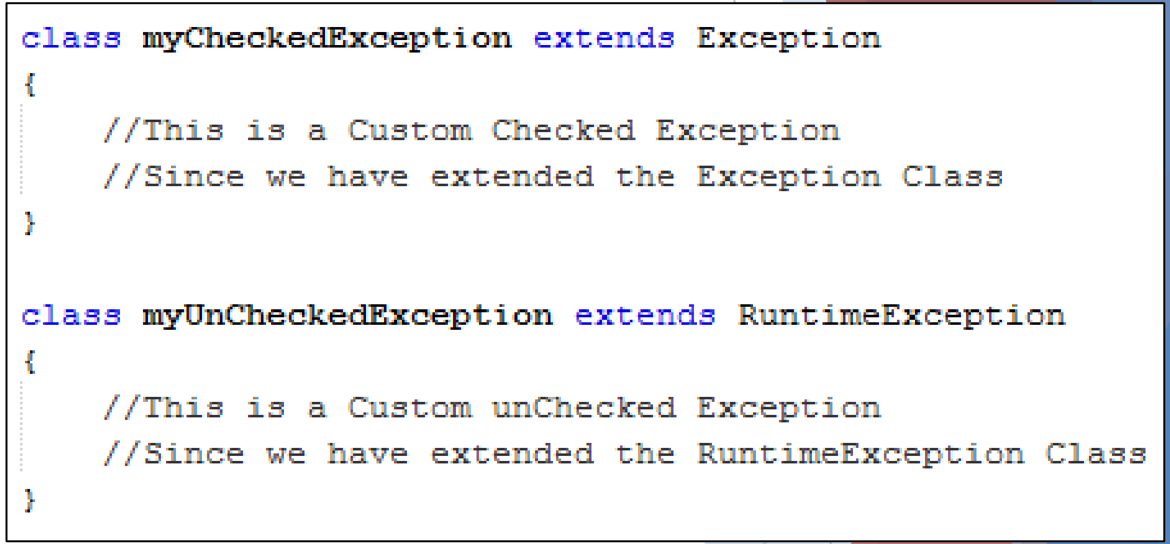
- Instead of (or in addition to) catching an exception a method might throw/rethrow an exception.
  - Either by not catching an exception
  - by explicitly throwing an exception
    - throw new ExceptionType();
- In case the method might throw a checked exception, this has to be declared explicitly in the method signature



```
///*********************************************************
//Since this method does not use exceptio Handling
//Any Exceptions occurs in our method will be rethrawn
void readInteger()
{
    Scanner sc = new Scanner(System.in);
    //This Line could throw an inputMismatchException
    //if the user enters an non integer input
    int num = sc.nextInt();
    System.out.println(num);
}
//********************************************************
//This method Expliciltly Throws an unchecked Exception
void throwException1()
{
    throw new InputMismatchException();
}
//********************************************************
//This method Expliciltly Throws a checked Exception
//We must includ that in the method signature
void throwException2() throws IOException
{
    throw new IOException();
}
//********************************************************
```

Custom Exceptions

- You can create a custom exception by inheriting from any of the exception classes.
- We do that when the already existing exception are not adequate to describe the problem.
- To create a custom checked exception extend the Exception class*“or any other checked exception subclass”*.
- To create a custom unchecked exception extend the RuntimeException class*“or any other unchecked exception subclass”*..



```
class myCheckedException extends Exception
{
    //This is a Custom Checked Exception
    //Since we have extended the Exception Class
}

class myUnCheckedException extends RuntimeException
{
    //This is a Custom unChecked Exception
    //Since we have extended the RuntimeException Class
}
```

Practicing Questions

- Write a console application that accepts two double numbers from the user, then print their sum, difference, product and quotient.
- The application should use exception handling to overcome any potential exceptions .
- The application should also use exception handling to verify that the entered numbers are not **very big** or **very small.**
- In case of the occurrence any Exception the program should state the error to the user and allows him to retry the procedure again.