QS STARS RATED FOR EXCELLENCE 2012

FPT Education

FPT UNIVERSITY

ĐẠI HỌC FPT CẦN THƠ

Session 04
More on classes
Nested classes

cantho.fpt.edu.vn

Objectives

- Describe methods
- Explain the process of creation and invocation of methods
- Explain passing and returning values from methods
- Explain variable argument methods
- Describe access specifiers and the types of access specifiers

Objectives

- Explain the use of access specifiers with methods
- Explain the concept of method overloading
- Explain the use of this keyword
- Nested classes

FPT Education

FPT UNIVERSITY

Methods

A Java method can be defined as a set of statements grouped together for performing a specific task.

For example, a call to the `main()` method which is the point of entry of any Java program, will execute all the statements written within the scope of the `main()` method.

- The syntax for declaring a method is as follows:

Syntax

```
modifier return_type method_name([list_of_parameters]) {
// Body of the method
}
```

where,

modifier: Specifies the visibility of the method. Visibility indicates which object can access the method. The values can be public, private, or protected.

return_type: Specifies the data type of the value returned by the method.

method_name: Specifies the name of the method.

list_of_parameters: Specifies the comma-delimited list of values passed to the method.

Methods

1

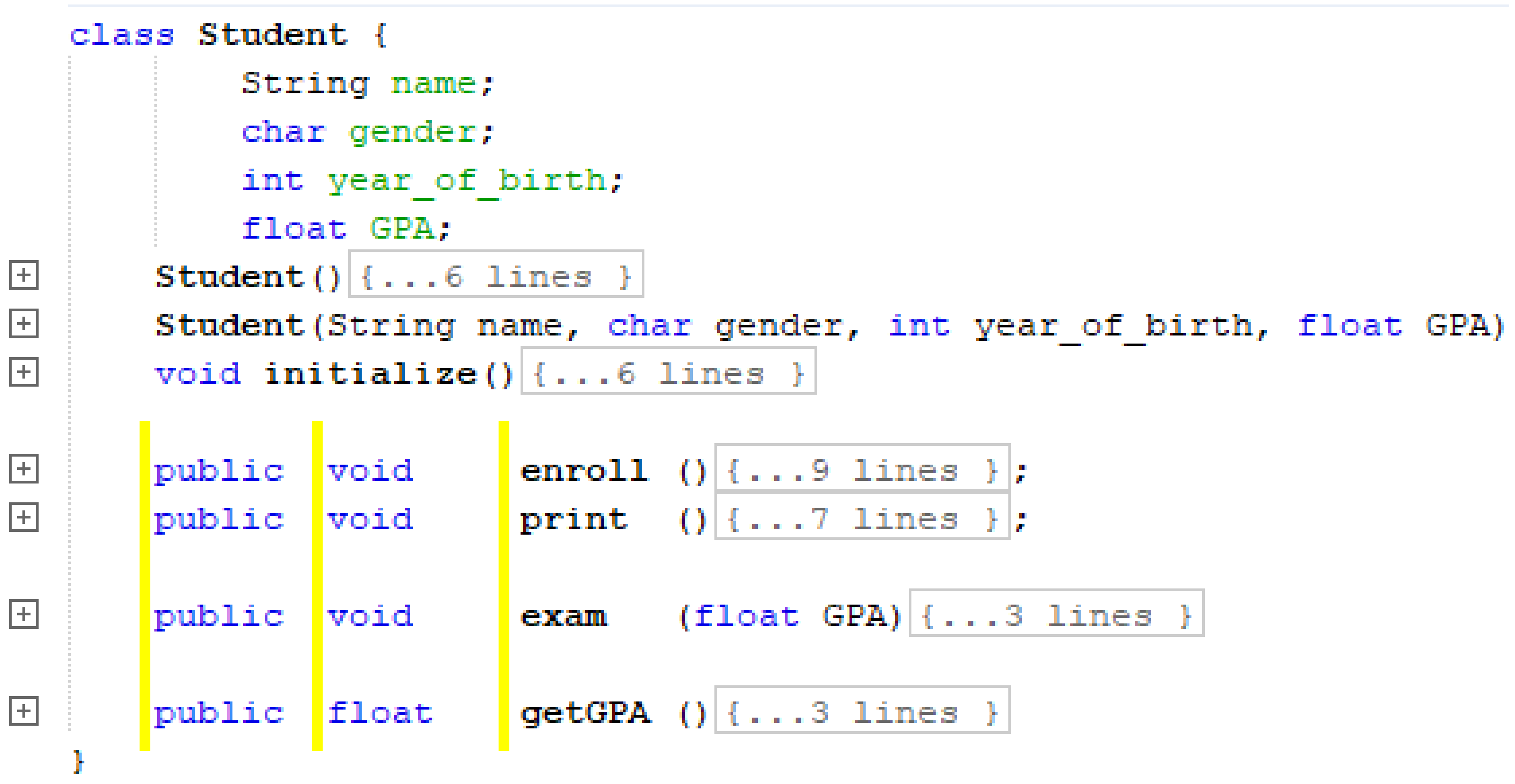
- Modifiers such as `public`, `private`, and `protected`.

2

- A return type that indicates the data type of the value returned by the method.
- The return type is set to `void` if the method does not return a value.

3

- The method name that is specified based on certain rules. A method name:
  - cannot be a Java keyword
  - cannot have spaces
  - cannot begin with a digit
  - cannot begin with any symbol other than a $ or _
  - can be a **verb** in lowercase
  - can be a multi-word name that begins with a verb in lowercase, followed by adjectives or nouns
  - can be a multi-word name with the first letter of the second word and each of the following words capitalized
  - should be descriptive and meaningful



```
class Student {
            String name;
            char gender;
            int year_of_birth;
            float GPA;
      Student() {...6 lines }
      Student(String name, char gender, int year_of_birth, float GPA)
      void initialize() {...6 lines }

      public  void      enroll  () {...9 lines };
      public  void      print   () {...7 lines };

      public  void      exam    (float GPA) {...3 lines }

      public  float     getGPA  () {...3 lines }
}
```

Methods

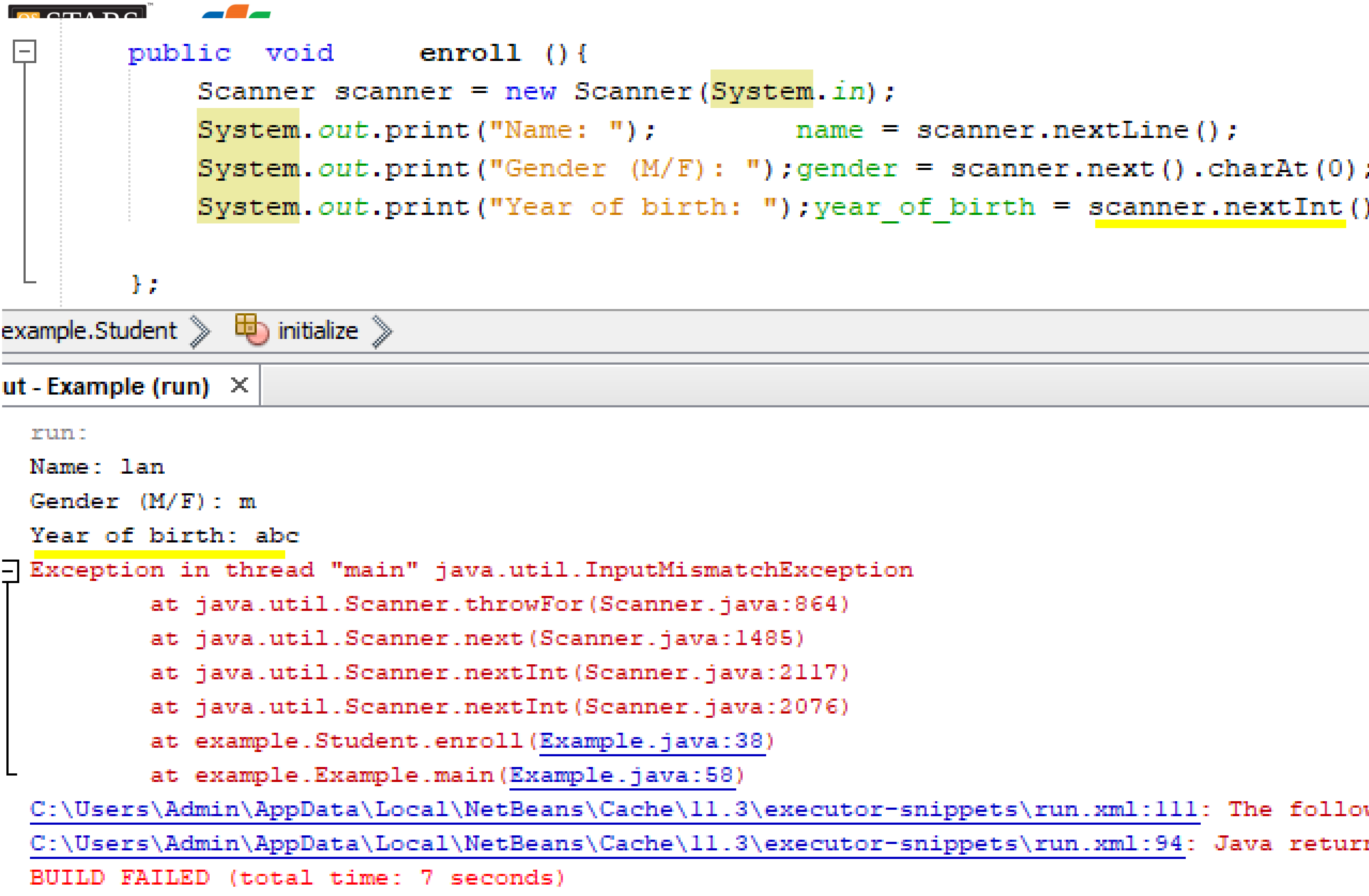
- Some valid method names are `add`, `_view`, `$calc`, `add_num`, `setFirstName`, `compareTo`, `isValid`, and so on.

4
- Parameter list in parenthesis is separated with a comma delimiter.
- Each parameter is preceded by its data type.
- If there are no parameters, an empty parenthesis is used.

5
- An exception list that specifies the names of exceptions that can be thrown by the method.
- An exception is an event encountered during the execution of the program, disrupting the flow of program execution.

6
- Method body consists of a set of statements enclosed between curly braces '{}'.
- Method body can have variables, method calls, and even classes.

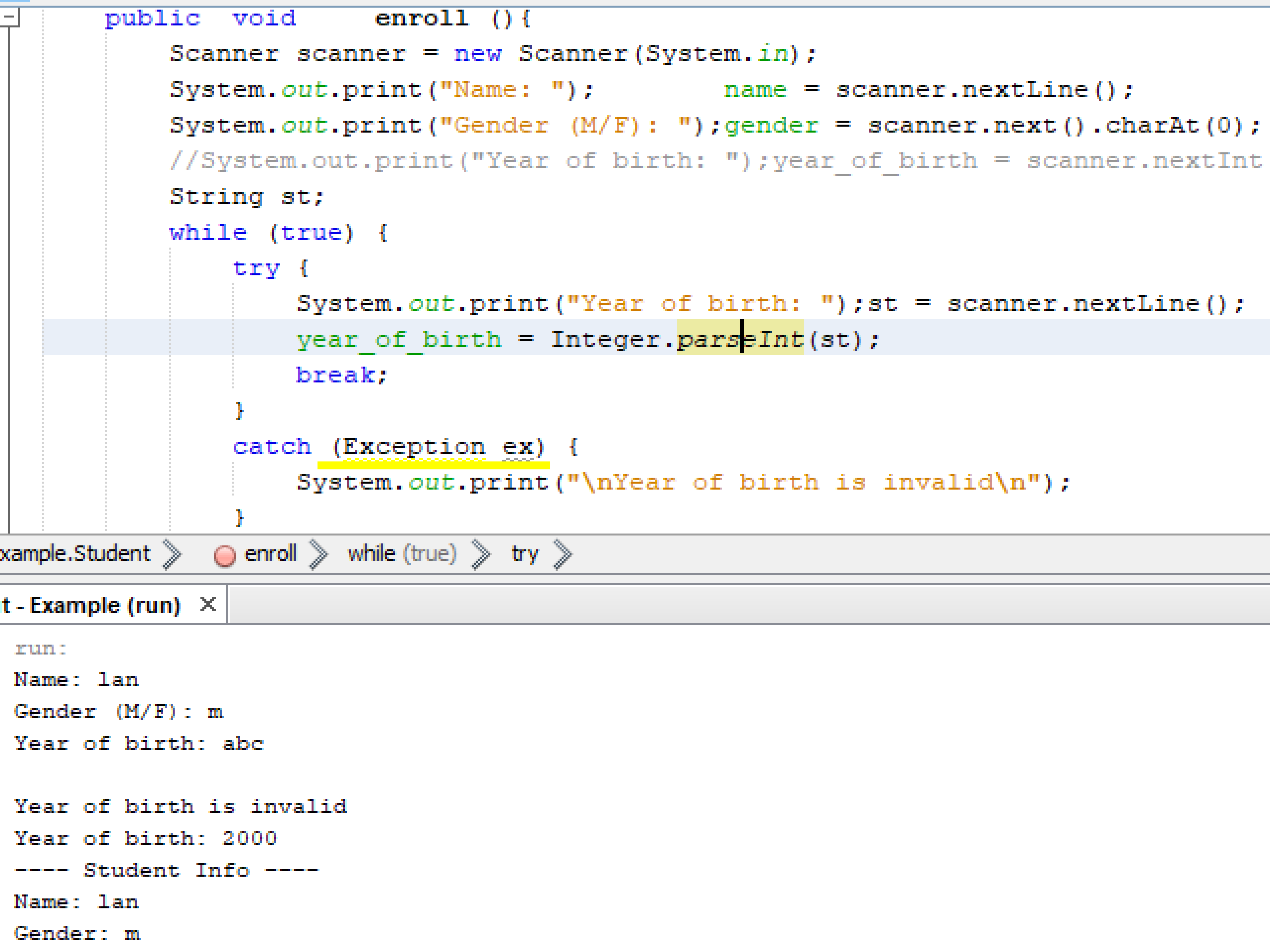
- The two components of a method declaration namely, the method name and the parameter types comprise the method signature.



```
    public  void     enroll  (){
        Scanner scanner = new Scanner(System.in);
        System.out.print("Name: ");          name = scanner.nextLine();
        System.out.print("Gender (M/F): ");gender = scanner.next().charAt(0)
        System.out.print("Year of birth: ");year_of_birth = scanner.nextInt()

    };
```

```
run:
Name: lan
Gender (M/F): m
Year of birth: abc
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at example.Student.enroll(Example.java:38)
        at example.Example.main(Example.java:58)
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:111: The follo
C:\Users\Admin\AppData\Local\NetBeans\Cache\11.3\executor-snippets\run.xml:94: Java retur
BUILD FAILED (total time: 7 seconds)
```



```
    public  void     enroll (){
        Scanner scanner = new Scanner(System.in);
        System.out.print("Name: ");         name = scanner.nextLine();
        System.out.print("Gender (M/F): ");gender = scanner.next().charAt(0);
        //System.out.print("Year of birth: ");year_of_birth = scanner.nextInt
        String st;
        while (true) {
            try {
                System.out.print("Year of birth: ");st = scanner.nextLine();
                year_of_birth = Integer.parseInt(st);
                break;
            }
            catch (Exception ex) {
                System.out.print("\nYear of birth is invalid\n");
            }
```

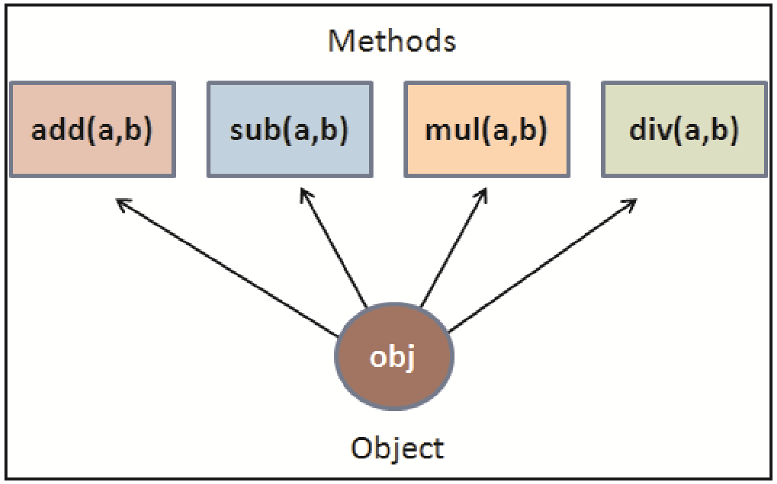
```
run:
Name: lan
Gender (M/F): m
Year of birth: abc

Year of birth is invalid
Year of birth: 2000
---- Student Info ----
Name: lan
Gender: m
```

FPT UNIVERSITY

Creating and Invoking Methods

- Methods help to segregate tasks to provide modularity to the program.
- A program is modular when different tasks in a program are grouped together into modules or sections.
- For example, to perform different types of mathematical operations such as addition, subtraction, multiplication, and so on, a user can create individual methods as shown in the following figure:



- The figure shows an object named **obj** accessing four different methods namely, **add(a,b)**, **sub(a,b)**, **mul(a,b)**, and **div(a,b)** for performing the respective operations on two numbers.

FPT Education

FPT UNIVERSITY

Creating and Invoking Methods

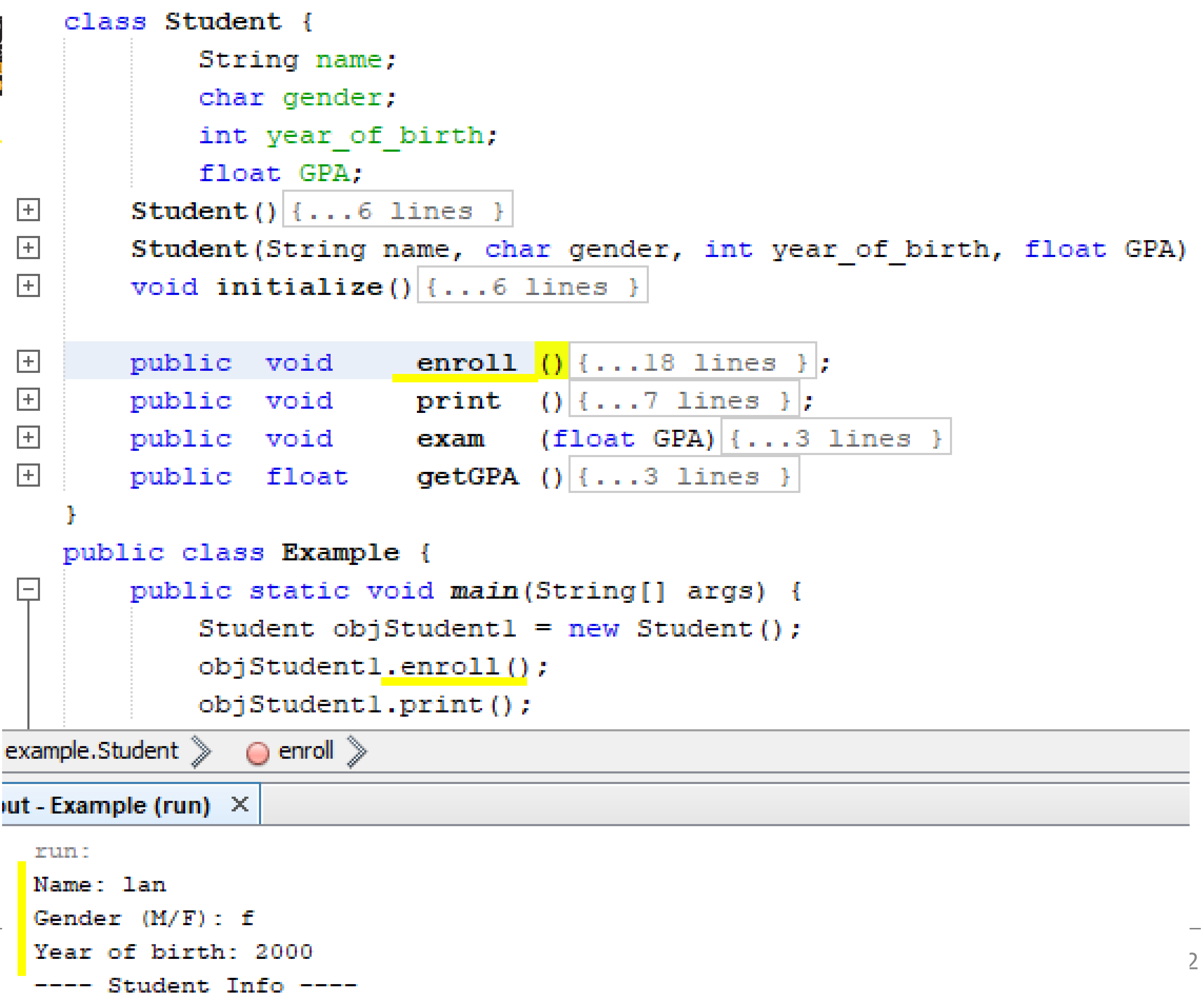
- To use a method, it must be called or invoked. When a program calls a method, the control is transferred to the called method.
- The called method executes and returns control to the caller.
- The call is returned back after the return statement of a method is executed or when the closing brace is reached.
- A method can be invoked in one of the following ways:

If the method returns a value, then, a call to the method results in return of some value from the method to the caller. For example,

```
int result = obj.add(20, 30);
```

If the method's return type is set to `void`, then, a call to the method results in execution of the statements within the method without returning any value to the caller.

For example, a call to the method would be `obj.add(23,30)` without anything returned to the caller.



```
class Student {
        String name;
        char gender;
        int year_of_birth;
        float GPA;
    Student(){...6 lines }
    Student(String name, char gender, int year_of_birth, float GPA)
    void initialize(){...6 lines }

    public  void     enroll ()({...18 lines };
    public  void     print  (){...7 lines };
    public  void     exam   (float GPA){...3 lines }
    public  float    getGPA (){...3 lines }
}
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        objStudent1.enroll();
        objStudent1.print();
```

example.Student 》 enroll 》

ut - Example (run) ✕

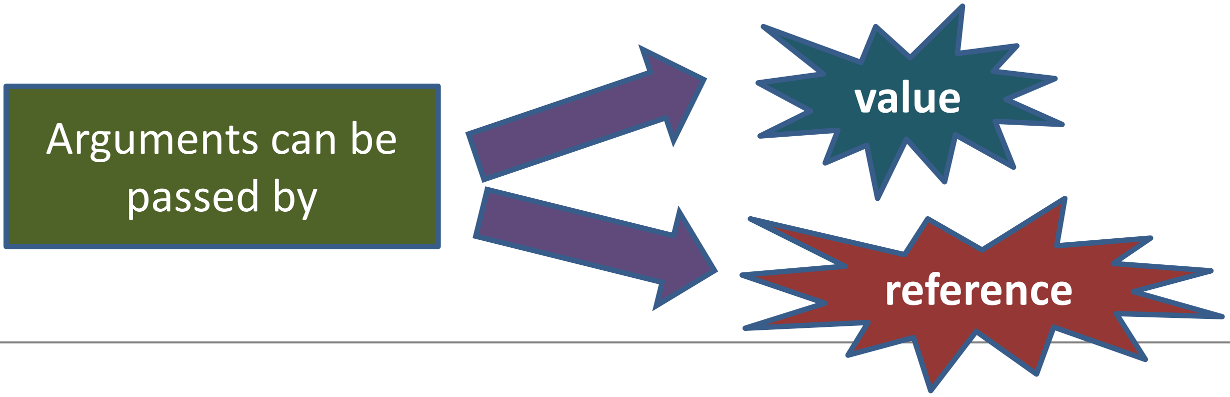
```
run:
Name: lan
Gender (M/F): f
Year of birth: 2000
---- Student Info ----
```

Passing and Returning Values from Methods

Parameters	Arguments
Parameters are the list of variables specified in a method declaration.	Arguments are the actual values that are passed to the method when it is invoked.

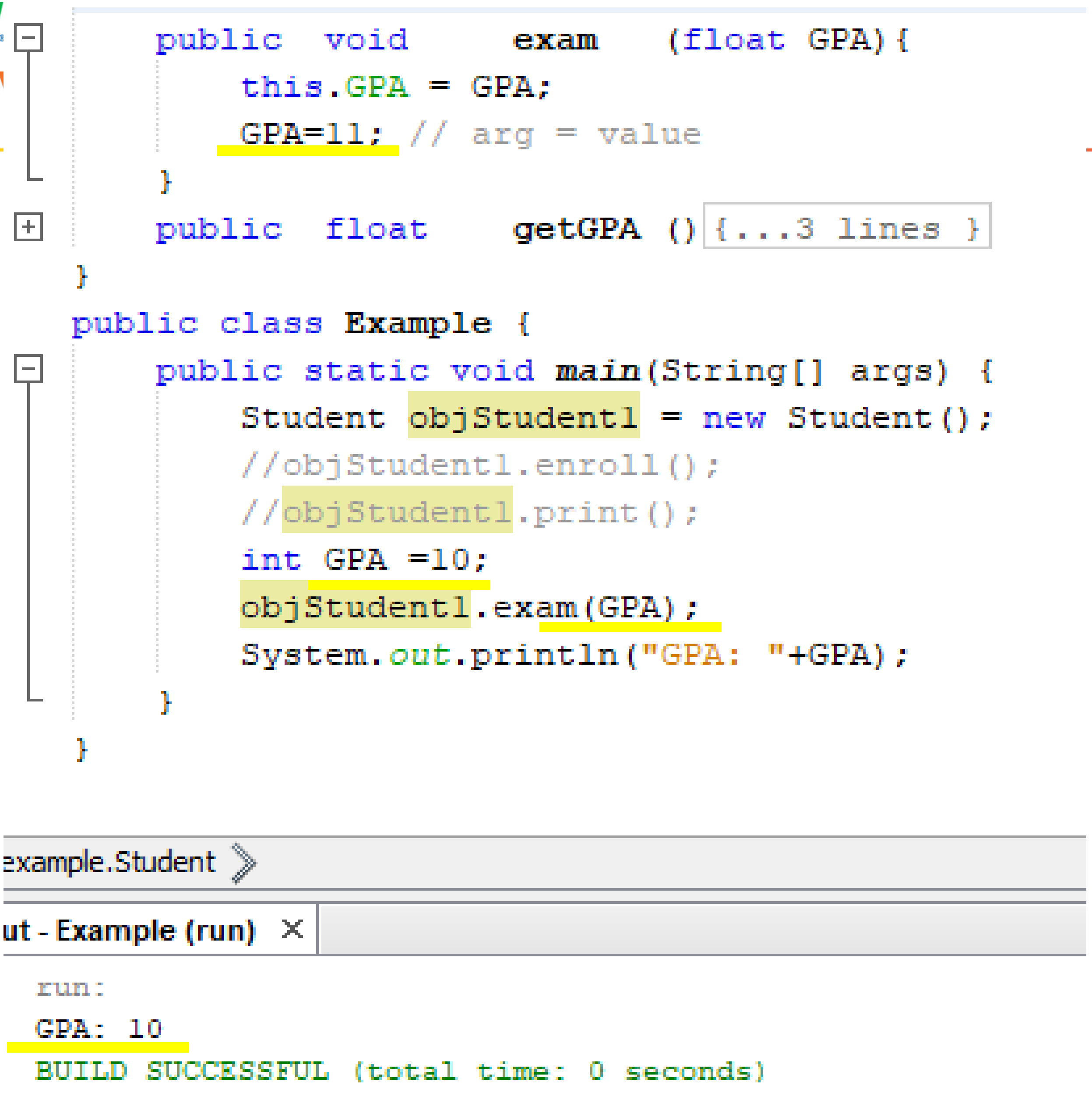
When a method is invoked, the type and order of arguments that are passed must match the type and order of parameters declared in the method.

A method can accept primitive data types such as `int`, `float`, `double`, and so on as well as reference data types such as arrays and objects as a parameter.



Passing Arguments by Value

- A copy of the argument is passed from the calling method to the called method.
- Changes made to the argument passed in the called method will not modify the value in the calling method.
- Variables of primitive data types such as `int` and `float` are passed by value.



```
    public  void      exam    (float GPA){
        this.GPA = GPA;
        GPA=11; // arg = value
    }
    public  float     getGPA () {...3 lines }
}
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        //objStudent1.enroll();
        //objStudent1.print();
        int GPA =10;
        objStudent1.exam(GPA);
        System.out.println("GPA: "+GPA);
    }
}
```

example.Student

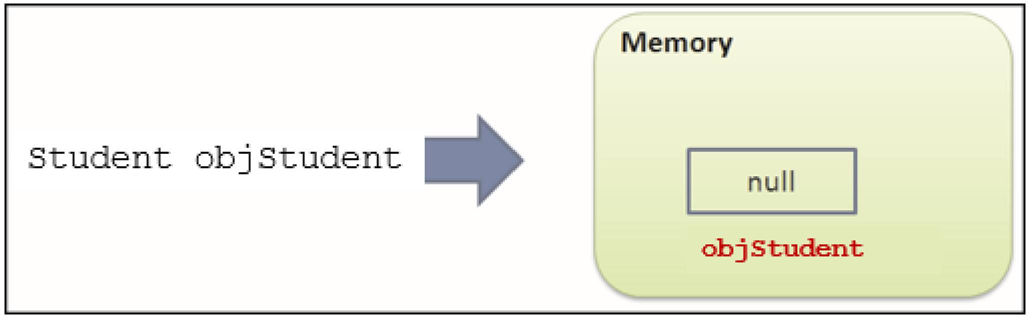
ut - Example (run)

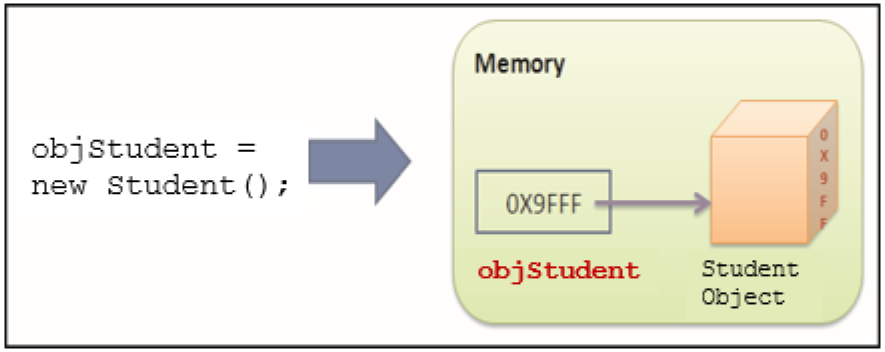
```
run:
GPA: 10
BUILD SUCCESSFUL (total time: 0 seconds)
```

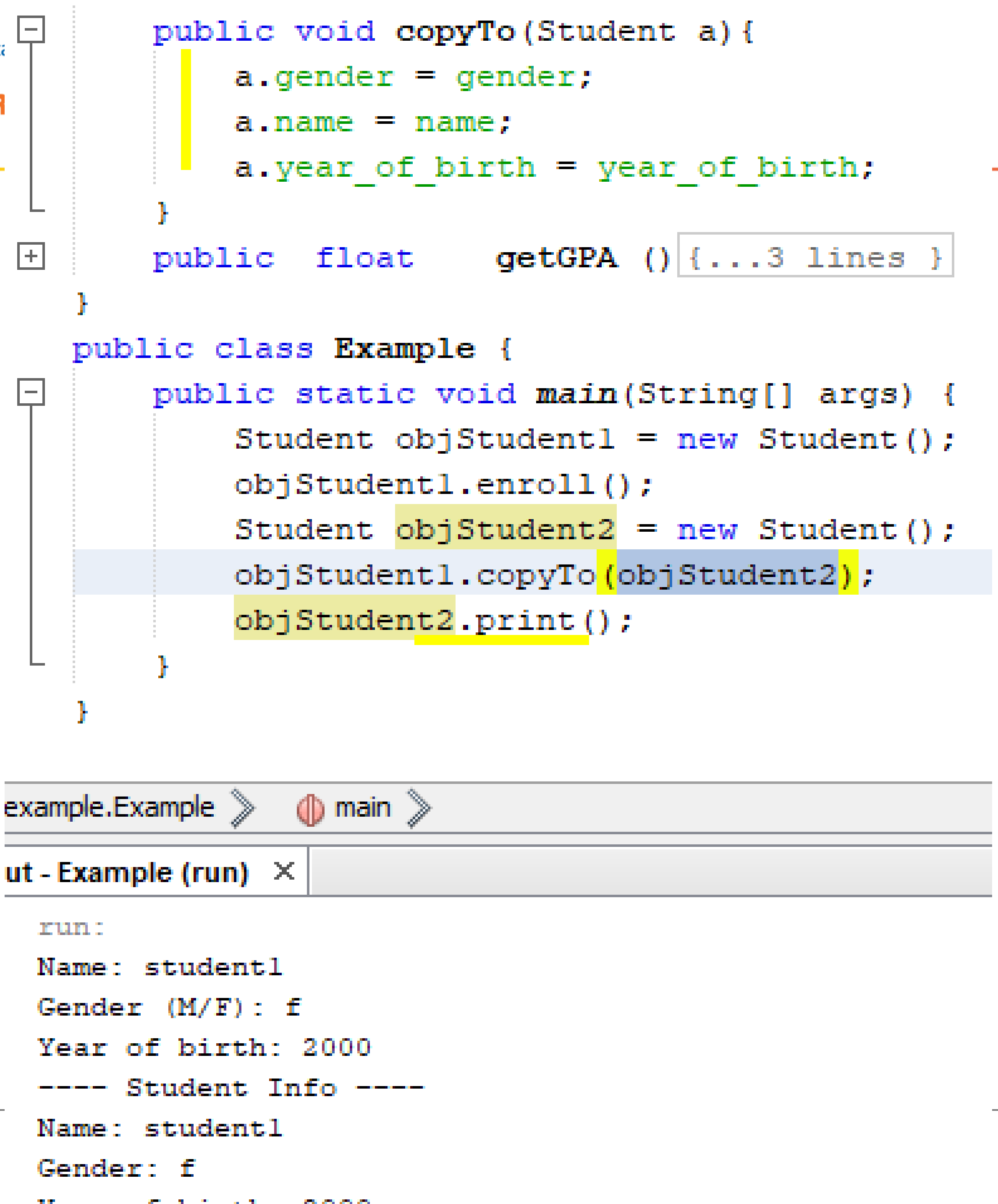
Passing Arguments by Reference

- The actual memory location of the argument is passed to the called method and the object or a copy of the object is not passed.
- The called method can change the value of the argument passed to it.
- Variables of reference types such as objects are passed to the methods by reference.
- There are two references of the same object namely, argument reference variable and parameter reference variable.

FPT UNIVERSITY







```
    public void copyTo(Student a){
        a.gender = gender;
        a.name = name;
        a.year_of_birth = year_of_birth;
    }
    public  float    getGPA (){...3 lines }
}
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        objStudent1.enroll();
        Student objStudent2 = new Student();
        objStudent1.copyTo(objStudent2);
        objStudent2.print();
    }
}
```

```
run:
Name: student1
Gender (M/F): f
Year of birth: 2000
---- Student Info ----
Name: student1
Gender: f
```

Declaring Variable Argument Methods

- Java provides a feature called `varargs` to pass variable number of arguments to a method.
- `varargs` is used when the number of a particular type of argument that will be passed to a method is not known until runtime.
- It serves as a shortcut to creating an array manually.
- To use `varargs`, the type of the last parameter is followed by ellipsis (...), then, a space, followed by the name of the parameter.
- This method can be called with any number of values for that parameter, including none.

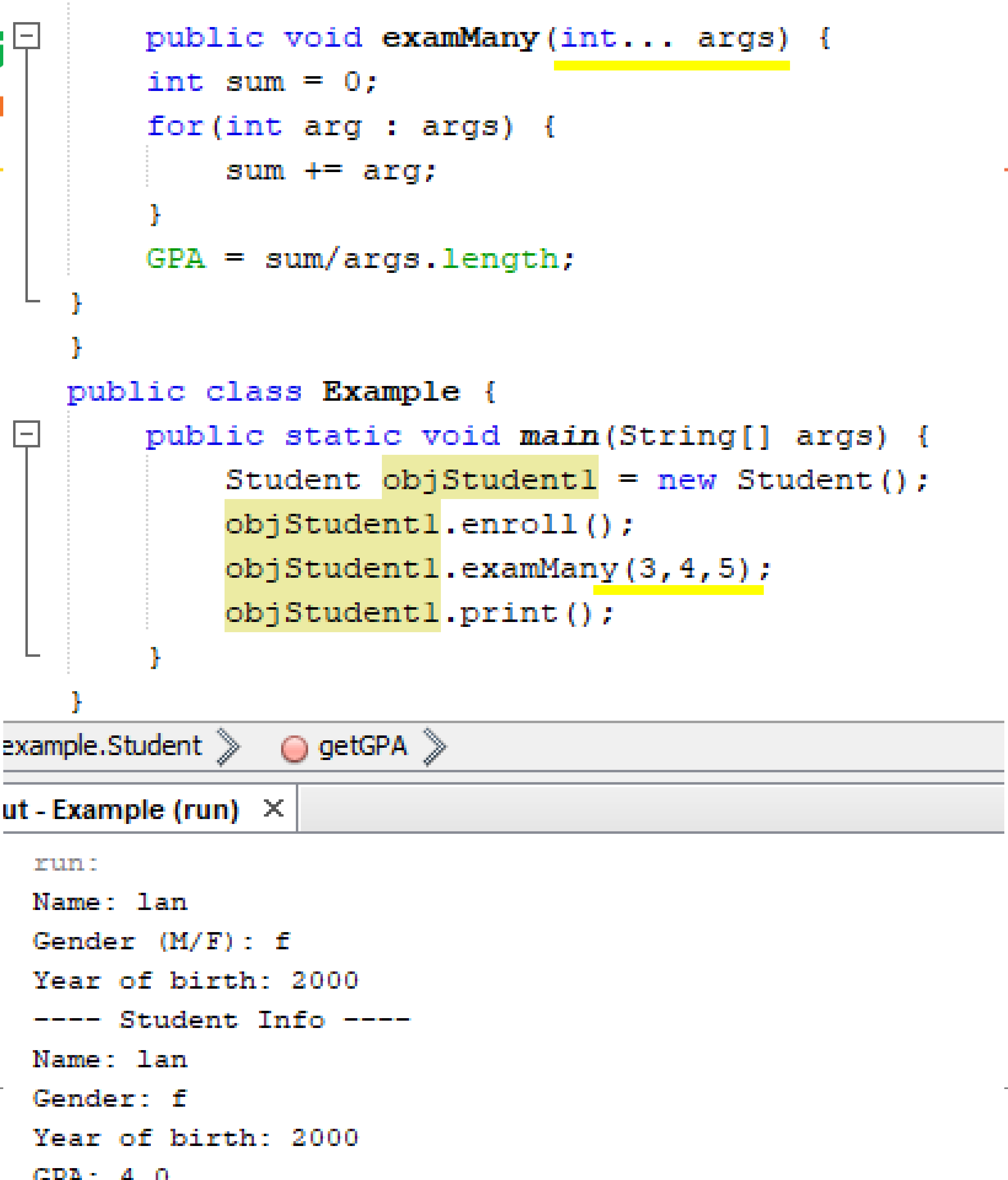
Declaring Variable Argument Methods

Syntax

```
<method_name>(type … variableName){
// method body
}
```

where,

‘…’: Indicates the variable number of arguments.


```
    public void examMany(int... args) {
        int sum = 0;
        for(int arg : args) {
            sum += arg;
        }
        GPA = sum/args.length;
    }
}
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        objStudent1.enroll();
        objStudent1.examMany(3,4,5);
        objStudent1.print();
    }
}
```

example.Student » getGPA »

ut - Example (run) ×

```
run:
Name: lan
Gender (M/F): f
Year of birth: 2000
---- Student Info ----
Name: lan
Gender: f
Year of birth: 2000
GPA: 4.0
```

Access Specifiers

- Access specifiers help to control the access of classes and class members.
- Access specifiers help to prevent misuse of class details as well as hide the implementation details that are not required by other classes.
- The access specifiers also determine whether classes and the members of the classes can be invoked by other classes or interfaces.
- Accessibility affects inheritance and how members are inherited by the subclass.
- A package is always accessible by default.

Type of Access Specifiers

public

- The `public` access specifier is the least restrictive of all access specifiers.
- A field, method, or class declared `public` is visible to any class in a Java application in the same package or in another.

private

- The `private` access specifier cannot be used for classes and interfaces as well as fields and methods of an interface.
- Fields and methods declared `private` cannot be accessed from outside the enclosing class.

Type of Access Specifiers

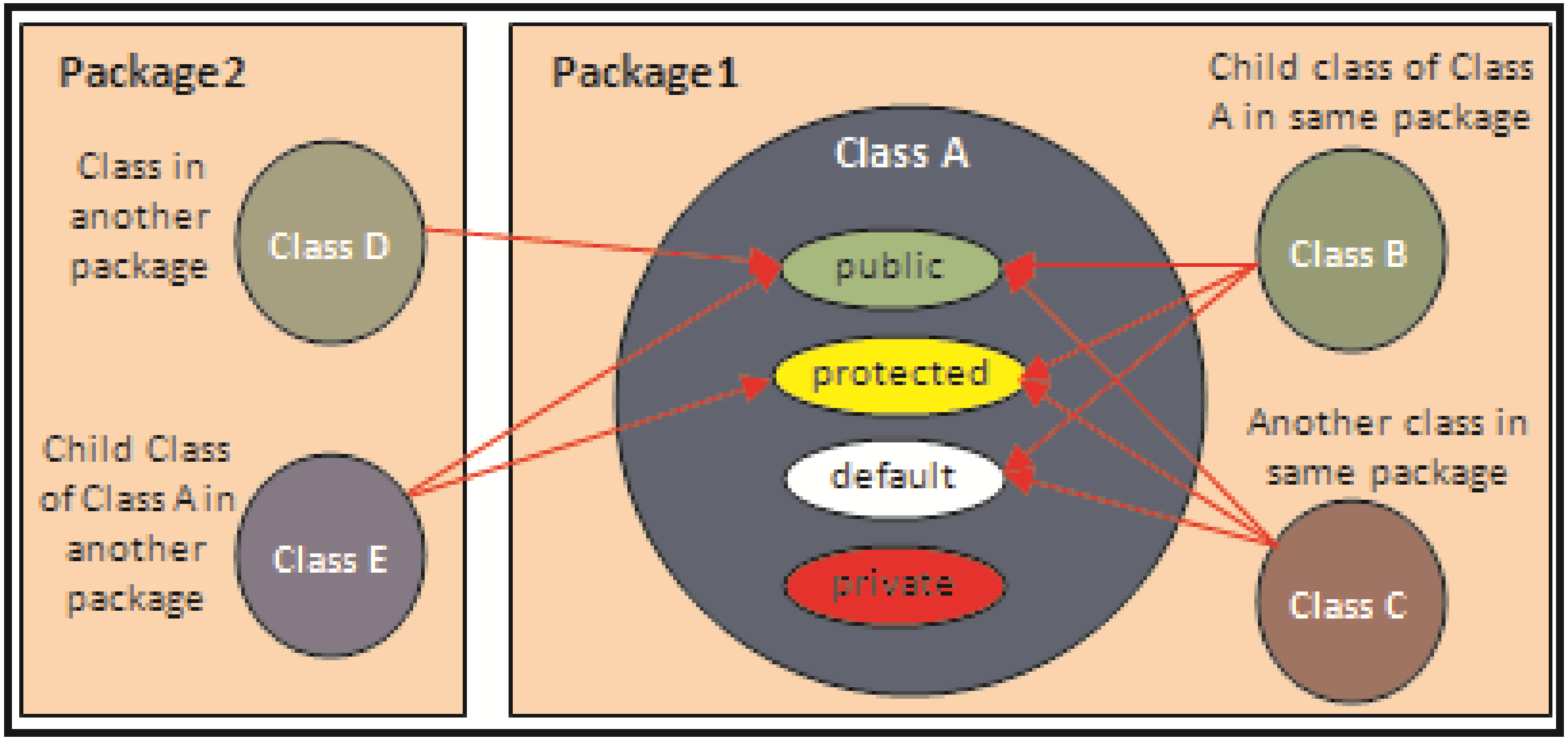
protected

- The `protected` access specifier is used with classes that share a parent-child relationship which is referred to as inheritance.
- The `protected` keyword cannot be used for classes and interfaces as well as fields and methods of an interface.
- Fields and methods declared `protected` in a parent or super class can be accessed only by its child or subclass in another packages.
- Classes in the same package can also access protected fields and methods, even if they are not a subclass of the `protected` member's class.

Default

- The default access specifier is used when no access specifier is present.
- The default specifier gets applied to any class, field, or method for which no access specifier has been mentioned.
- With default specifier, the class, field, or method is accessible only to the classes of the same package.
- The default specifier is not used for fields and methods within an interface.

Type of Access Specifiers


Package2
Class in another package
Class D
Child Class of Class A in another package
Class E
Package1
Class A
public
protected
default
private
Child class of Class A in same package
Class B
Another class in same package
Class C

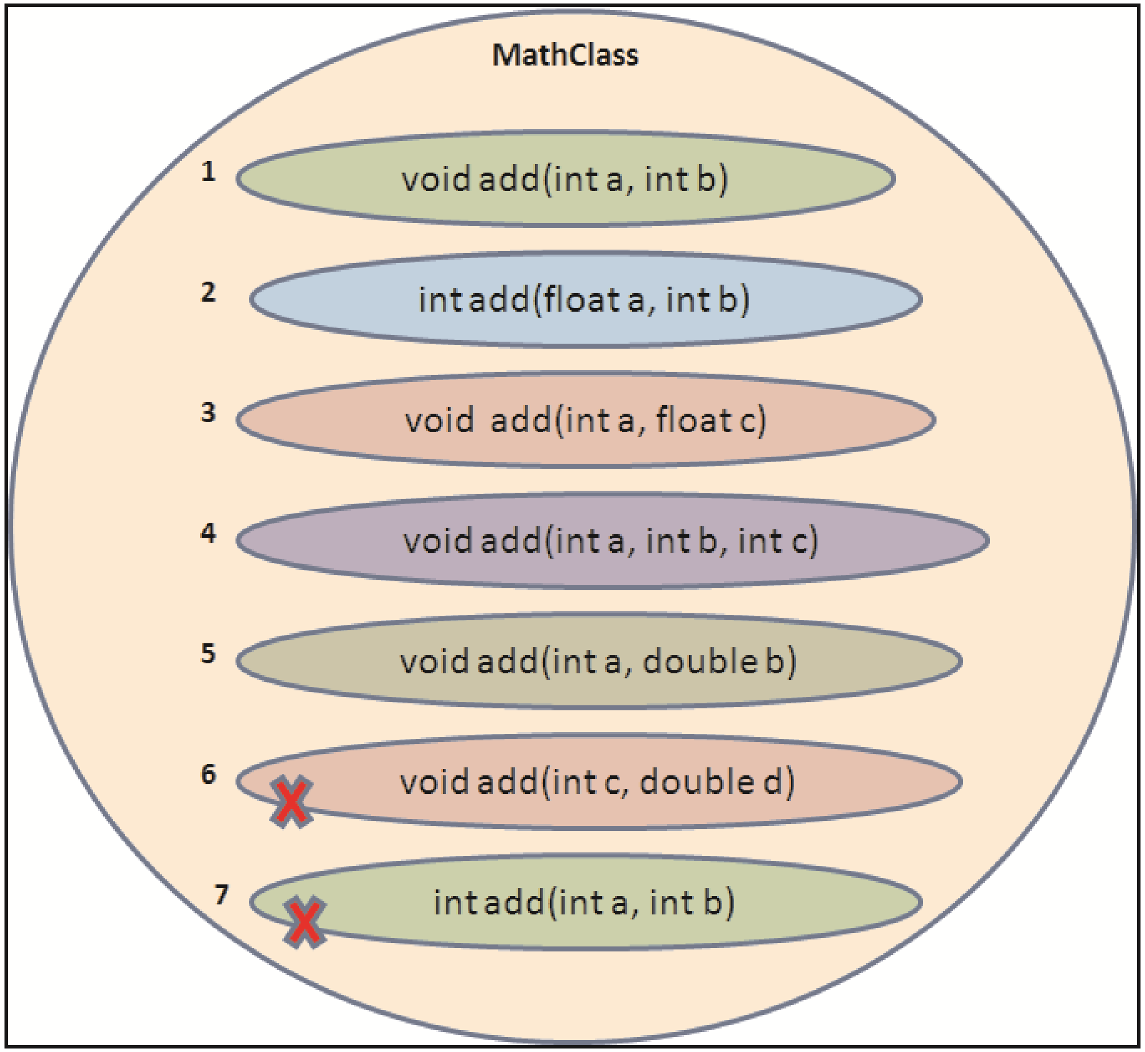
Rules for Access Control

- While declaring members, a `private` access specifier cannot be used with `abstract`, but it can be used with `final` or `static`.
- No access specifier can be repeated twice in a single declaration.
- A constructor when declared `private` will be accessible in the class where it was created.
- A constructor when declared `protected` will be accessible within the class where it was created and in the inheriting classes.

Rules for Access Control

- `private` cannot be used with fields and methods of an interface.
- The most restrictive access level must be used that is appropriate for a particular member.
- Mostly, a `private` access specifier is used at all times unless there is a valid reason for not using it.
- Avoid using `public` for fields except for constants.

Method Overloading


MathClass
1
void add(int a, int b)
2
int add(float a, int b)
3
void add(int a, float c)
4
void add(int a, int b, int c)
5
void add(int a, double b)
6
void add(int c, double d)
7
int add(int a, int b)

FPT Education

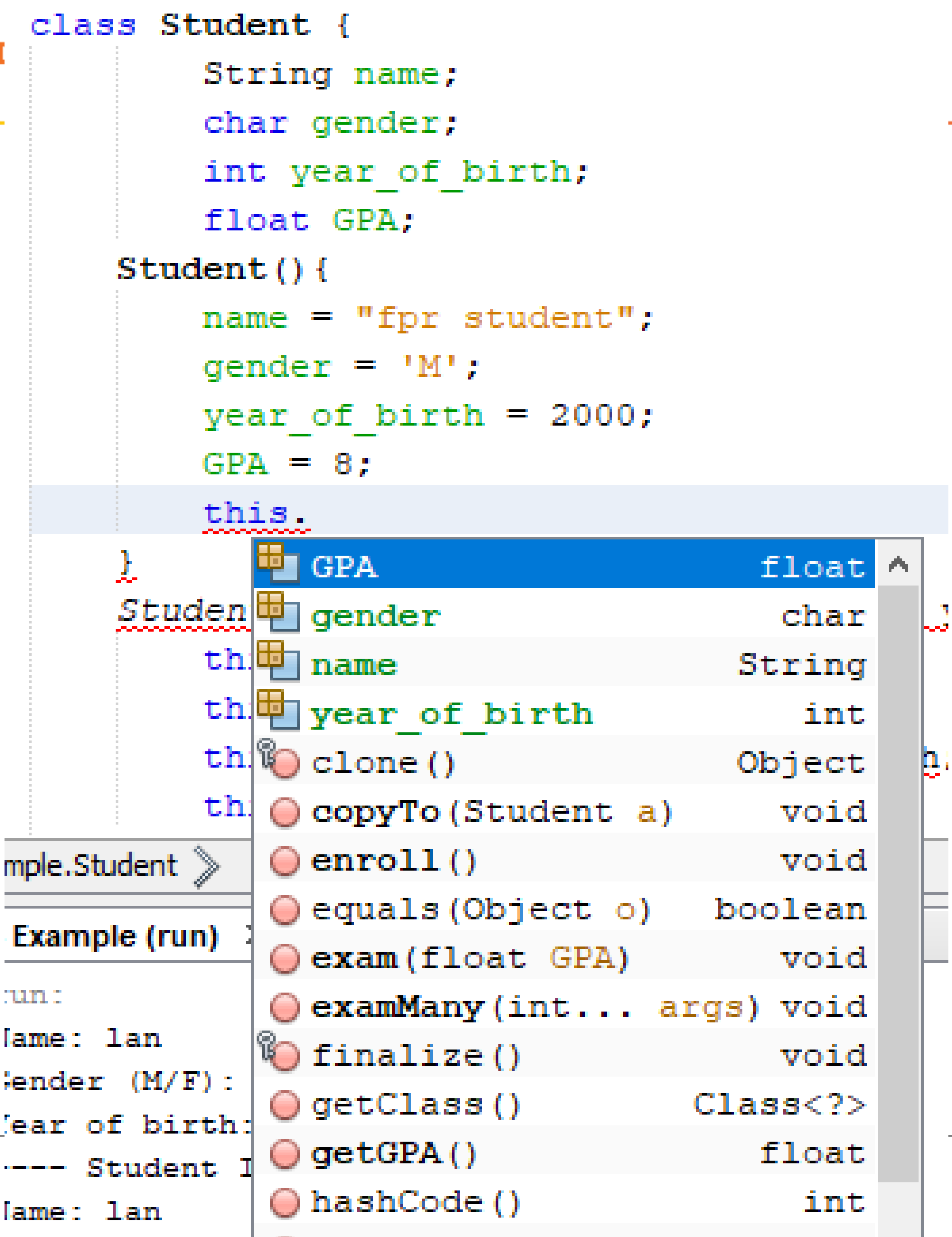
FPT UNIVERSITY

Constructor Overloading

- Constructor is a special method of a class that has the same name as the class name.
- A constructor is used to initialize the variables of a class.
- Similar to a method, a constructor can also be overloaded to initialize different types and number of parameters.
- When the class is instantiated, the compiler invokes the constructor based on the number, type, and sequence of arguments passed to it.

Using ‘this’ Keyword

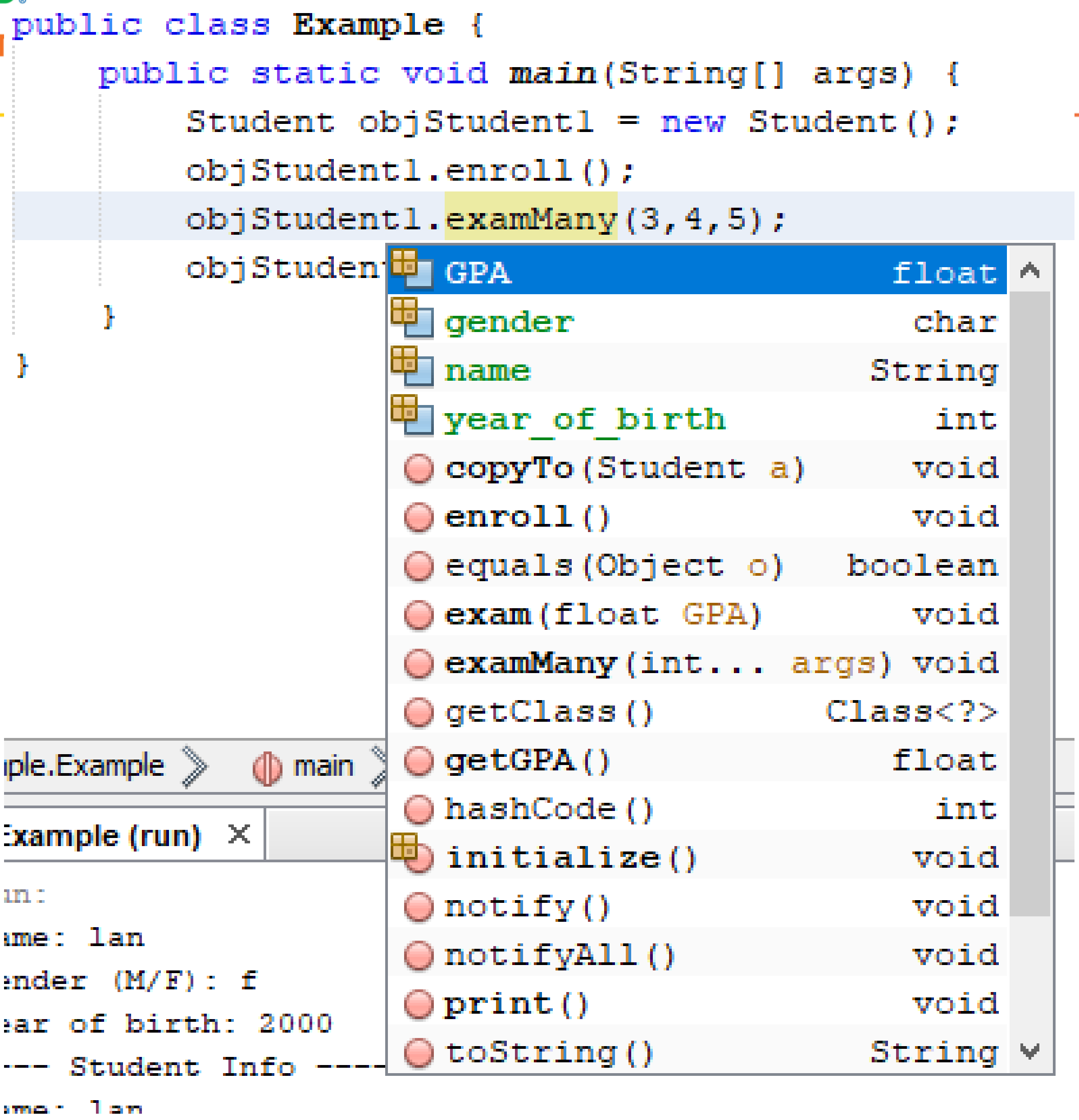
- Java provides the keyword this which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being called.
- Any member of the current object can be referred from within an instance method or a constructor by using the `this` keyword.
- The keyword `this` is not explicitly used in instance methods while referring to variables and methods of a class.
- The keyword this can also be used to invoke a constructor from within another constructor.
- The keyword this can be used to resolve naming conflicts when the names of actual and formal parameters of a method or a constructor are the same.



```
class Student {
        String name;
        char gender;
        int year_of_birth;
        float GPA;
    Student(){
        name = "fpr student";
        gender = 'M';
        year_of_birth = 2000;
        GPA = 8;
        this.
    }
    Studen
        th
        th
        th
        th
```

```
GPA                      float
gender                   char
name                     String
year_of_birth            int
clone()                  Object
copyTo(Student a)        void
enroll()                 void
equals(Object o)         boolean
exam(float GPA)          void
examMany(int... args)    void
finalize()               void
getClass()               Class<?>
getGPA()                 float
hashCode()               int
```

```
:un:
lame: lan
;ender (M/F):
'ear of birth:
---- Student I
lame: lan
```



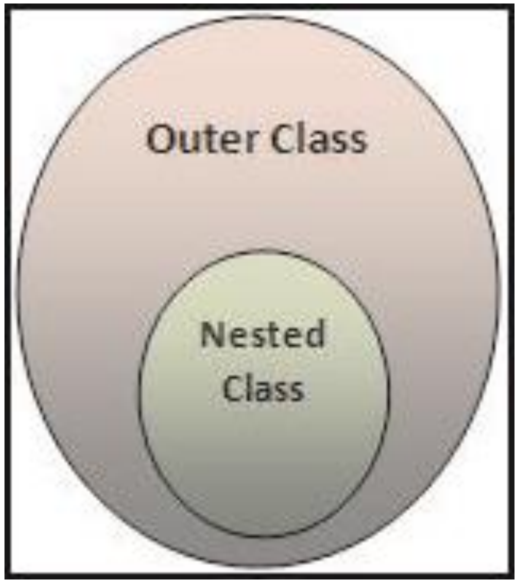
```
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        objStudent1.enroll();
        objStudent1.examMany(3,4,5);
        objStuden
    }
}
```

```
GPA                       float
gender                    char
name                      String
year_of_birth             int
copyTo(Student a)         void
enroll()                  void
equals(Object o)          boolean
exam(float GPA)           void
examMany(int... args)     void
getClass()                Class<?>
getGPA()                  float
hashCode()                int
initialize()              void
notify()                  void
notifyAll()               void
print()                   void
toString()                String
```

```
in:
ame: lan
ender (M/F): f
ar of birth: 2000
--- Student Info ----
ame: lan
```

Nested Class

- Java allows defining a class within another class.
- Such a class is called a nested class as shown in the following figure:

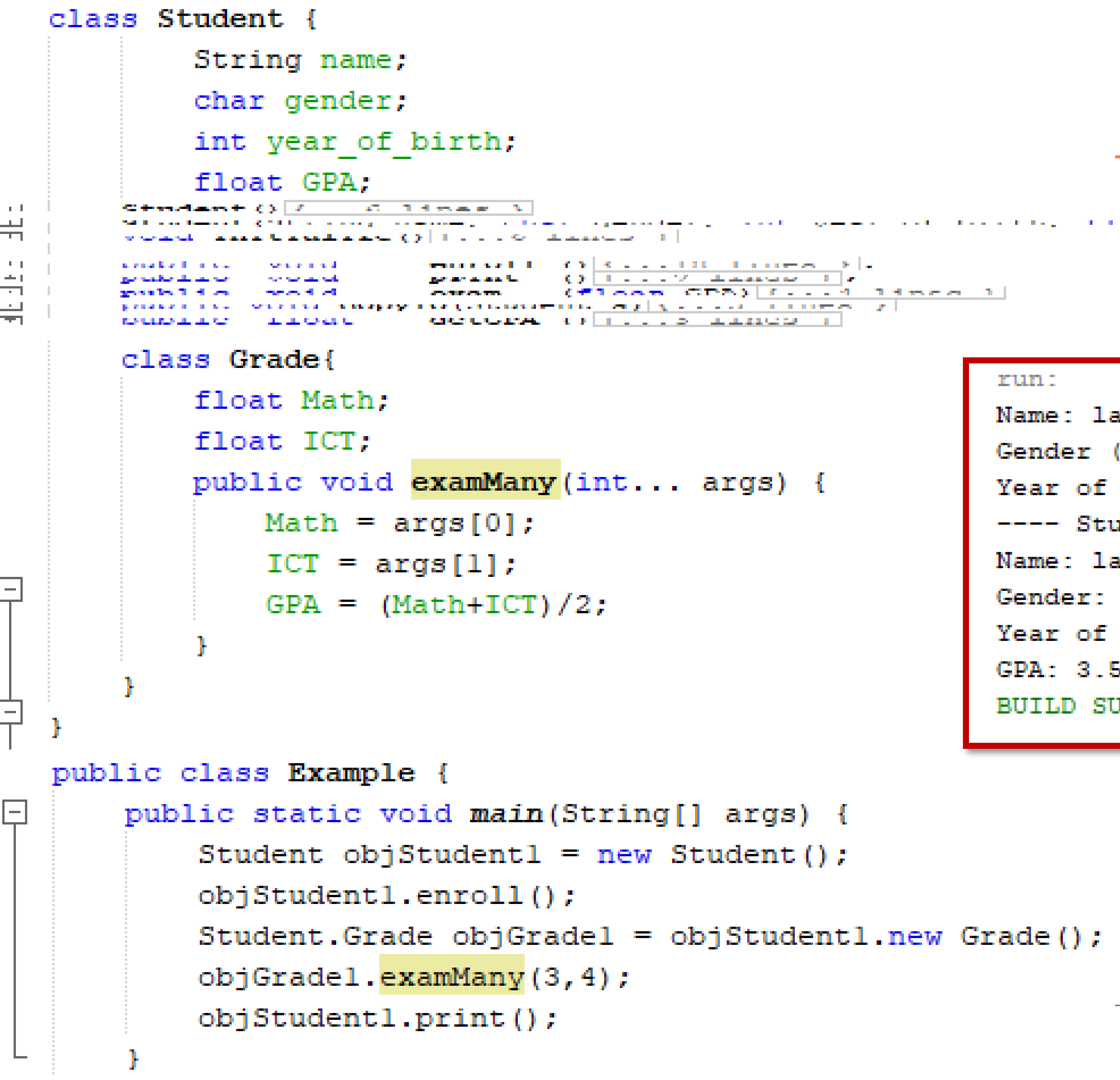


```
class Outer{
    ...
    class Nested{
        ...
    }
}
```

Nested Class

- Nested classes are classified as static and non-static.
- Nested classes that are declared `static` are simply termed as `static` nested classes whereas non-static nested classes are termed as inner classes.

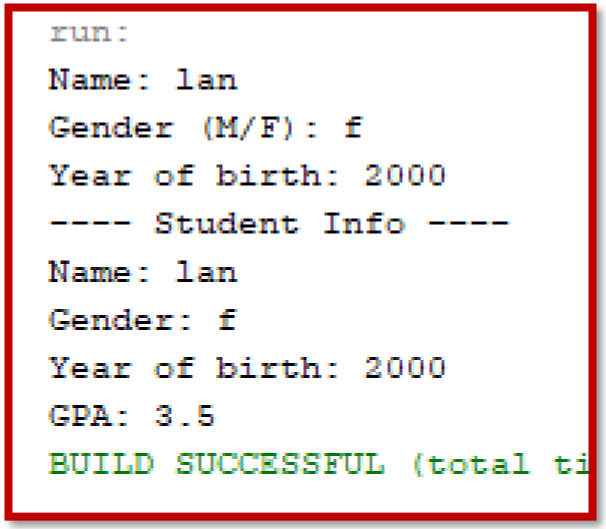
```
class Outer{
   ...
   static class StaticNested{
      ...
   }
   class Inner{
   ...
   }
}
```



```
class Student {
        String name;
        char gender;
        int year_of_birth;
        float GPA;
    Student() {...6 lines }
    Student(String name, char gender, int year_of_birth, [illegible]
    void initialize() {...6 lines }

    public void    enroll() {...11 lines };
    public void    print()  {...7 lines };
    public void    exam    (float GPA) {...4 lines }
    public void examMany(int... args) {...[illegible] lines }
    public float   getGPA  () {...3 lines }

    class Grade{
        float Math;
        float ICT;
        public void examMany(int... args) {
            Math = args[0];
            ICT = args[1];
            GPA = (Math+ICT)/2;
        }
    }
}
public class Example {
    public static void main(String[] args) {
        Student objStudent1 = new Student();
        objStudent1.enroll();
        Student.Grade objGrade1 = objStudent1.new Grade();
        objGrade1.examMany(3,4);
        objStudent1.print();
    }
```



```
run:
Name: lan
Gender (M/F): f
Year of birth: 2000
---- Student Info ----
Name: lan
Gender: f
Year of birth: 2000
GPA: 3.5
BUILD SUCCESSFUL (total ti
```

Benefits of Using Nested Class

Creates logical grouping of classes

- If a class is of use to only one class, then it can be embedded within that class and the two classes can be kept together.
- In other words, it helps in grouping the related functionality together.
- Nesting of such 'helper classes' helps to make the package more efficient and streamlined.

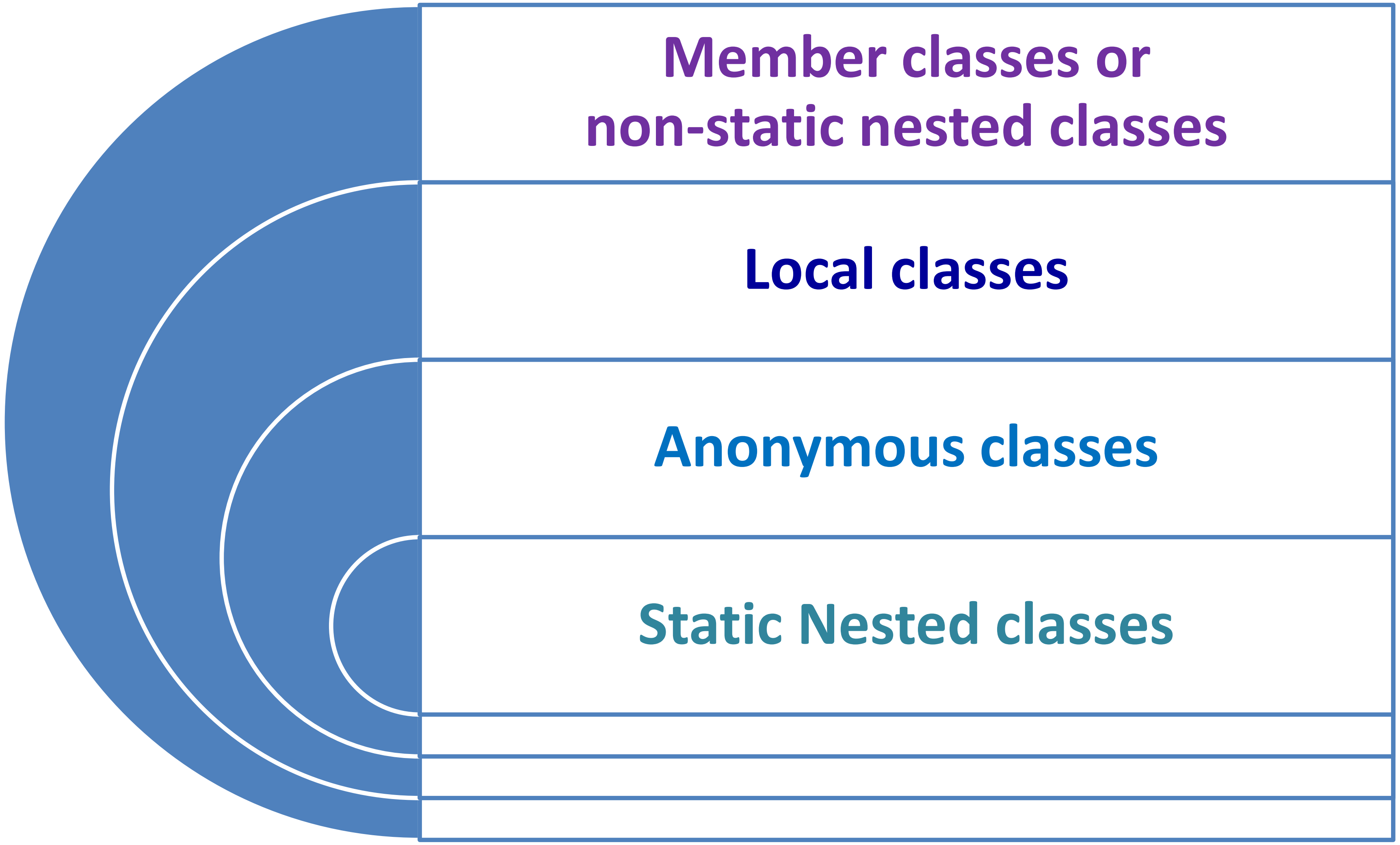
Increases encapsulation

- In case of two top level classes such as class A and B, when B wants access to members of A that are `private`, class B can be nested within class A so that B can access the members declared as `private`.
- Also, this will hide class B from the outside world.
- Thus, it helps to access all the members of the top-level enclosing class even if they are declared as `private`.

Increased readability and maintainability of code

- Nesting of small classes within larger top-level classes helps to place the code closer to where it will be used.

Types of Nested Classes



Member Classes

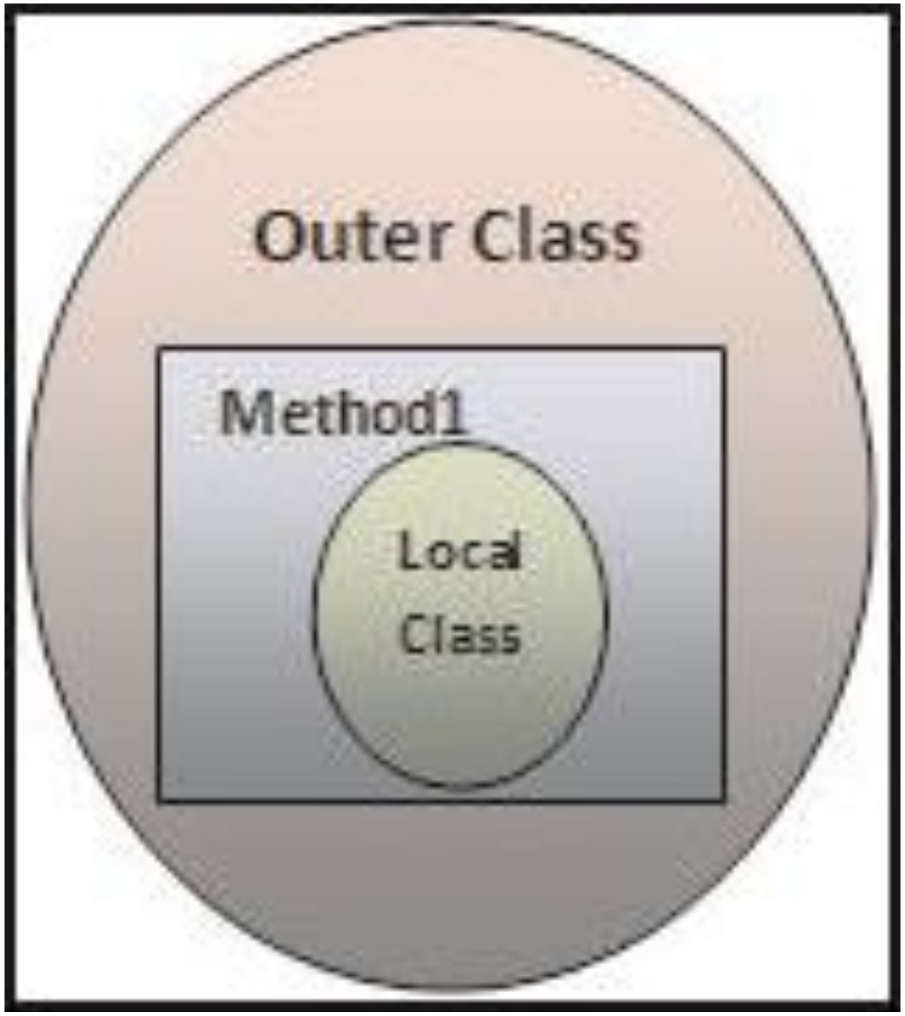
- A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

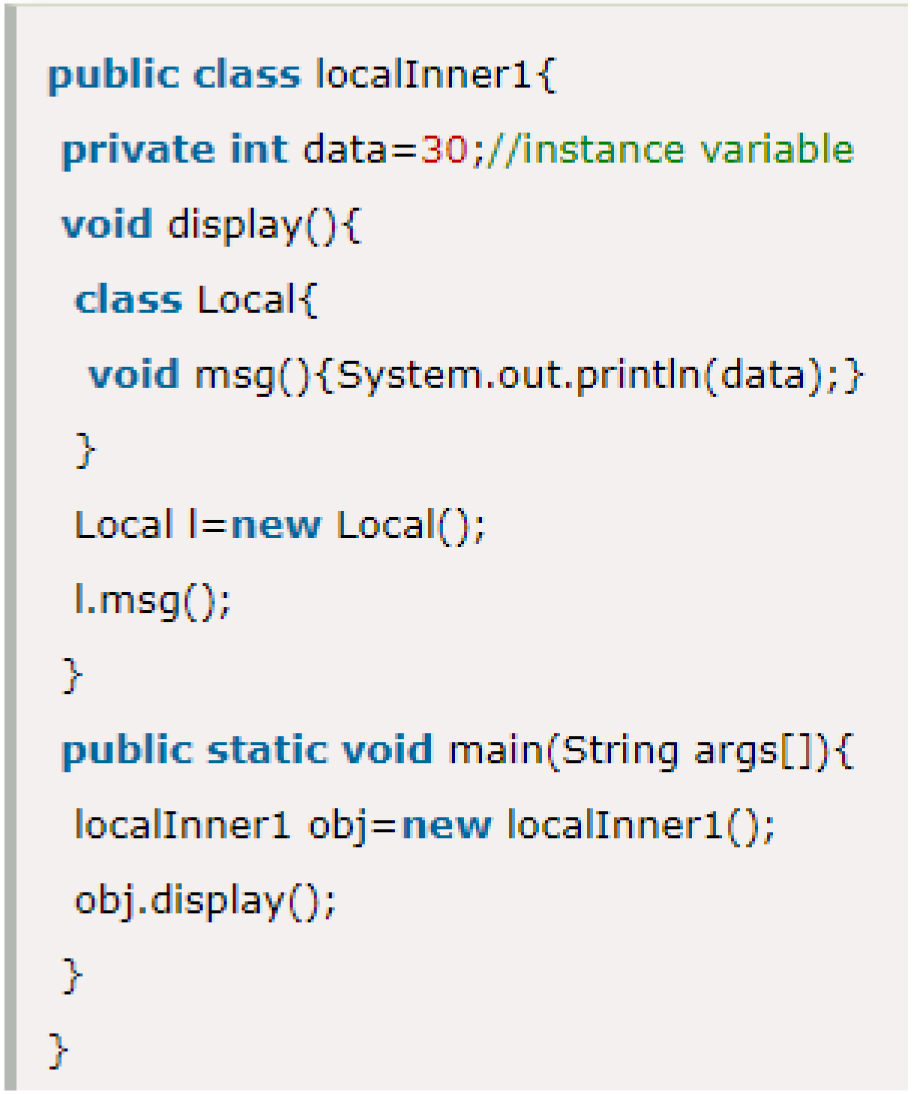
```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Local class



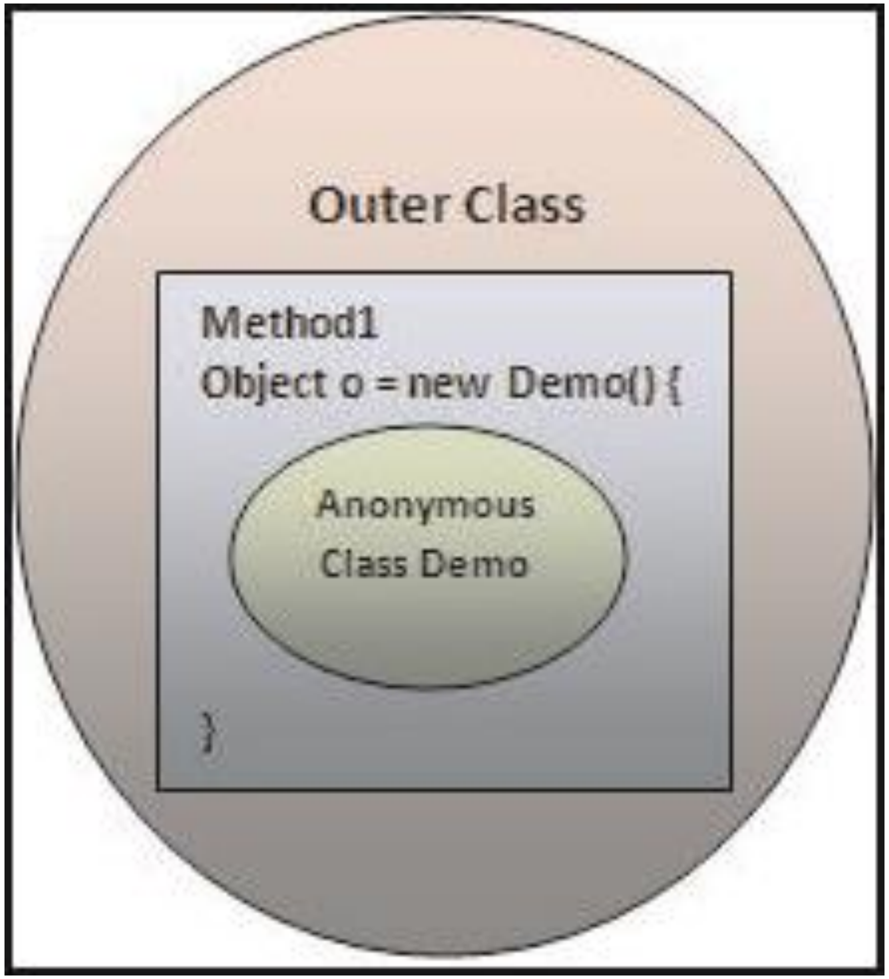
Local class

- A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.



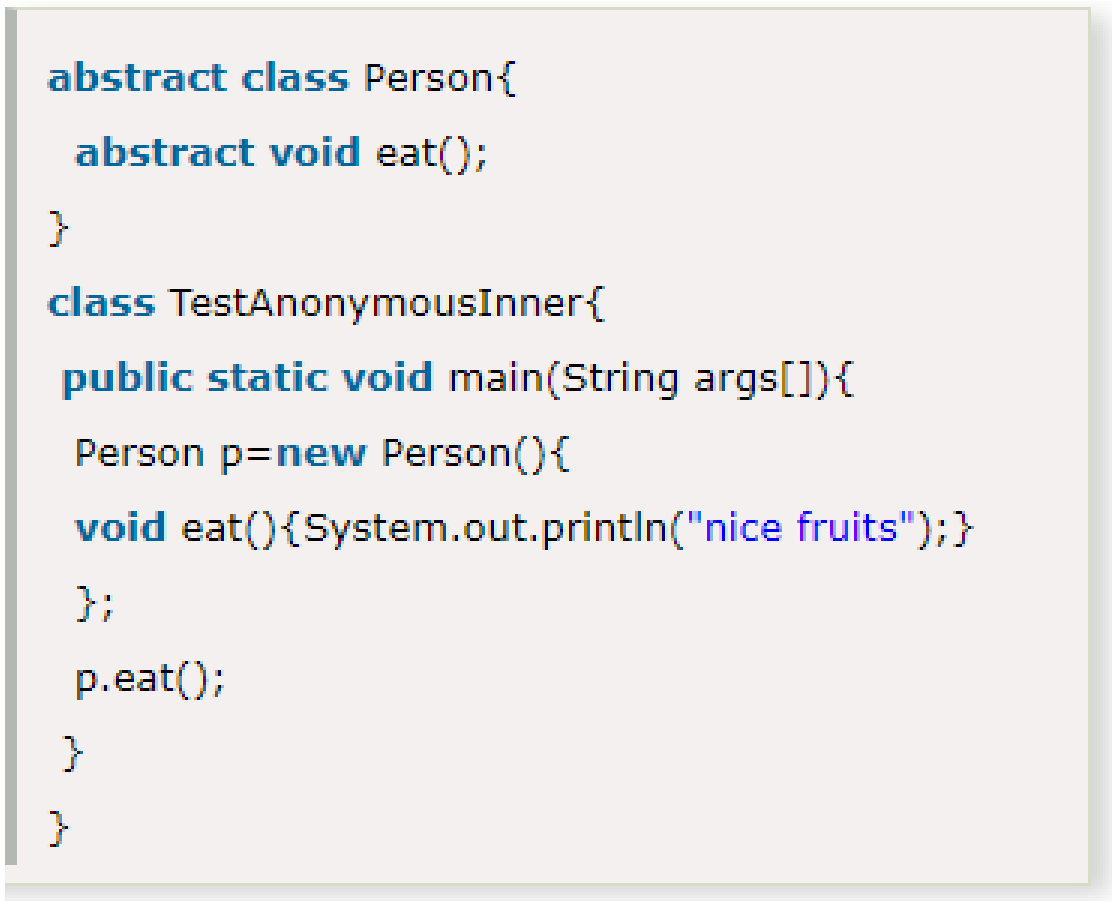
```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Anonymous Class



Anonymous Class

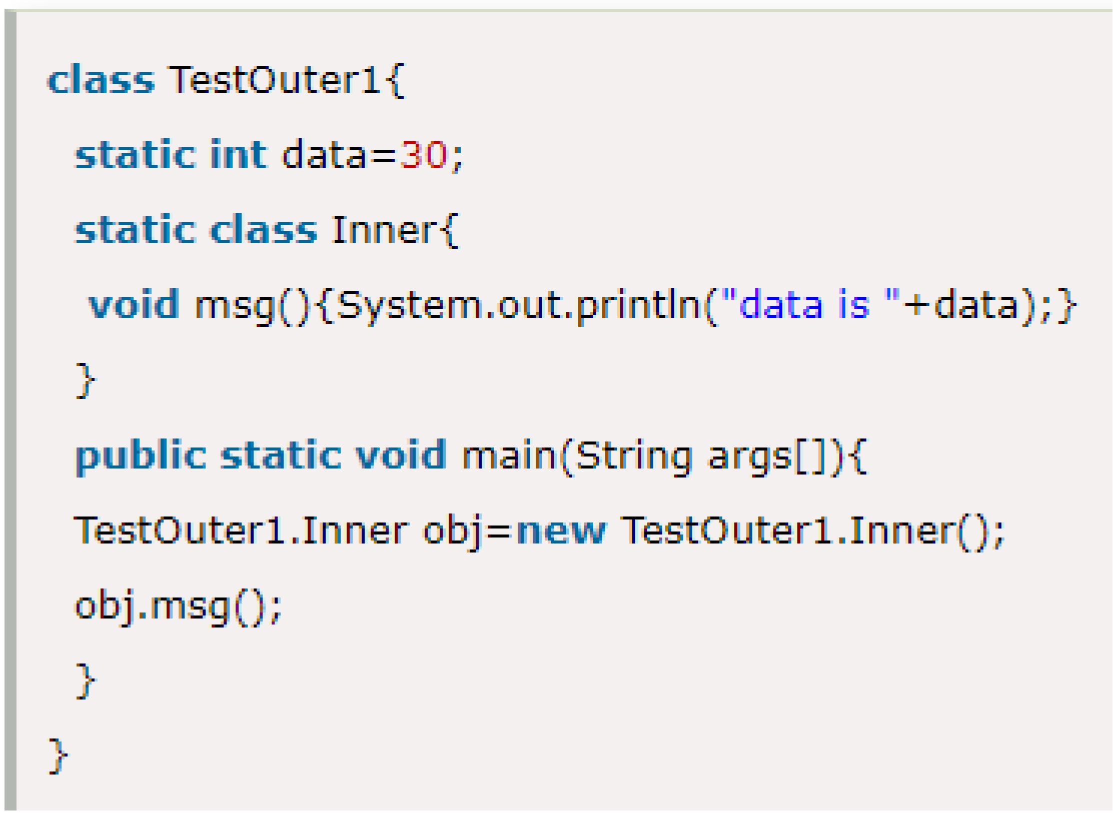
- A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface.



```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Static Nested Class

- A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.



```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

Summary

- A Java method is a set of statements grouped together for performing a specific operation.
- Parameters are the list of variables specified in a method declaration, whereas arguments are the actual values that are passed to the method when it is invoked.
- The variable argument feature is used in Java when the number of a particular type of arguments that will be passed to a method is not known until runtime.
- Access specifiers are used to restrict access to fields, methods, constructor, and classes of an application.
- Java comes with four access specifiers namely, public, private, protected, and default.
- Using method overloading, multiple methods of a class can have the same name but with different parameter lists.
- Java provides the ‘this’ keyword which can be used in an instance method or a constructor to refer to the current object, that is, the object whose method or constructor is being invoked.

Summary

- A member class is a non-static inner class. It is declared as a member of the outer or enclosing class.
- An inner class defined within a code block such as the body of a method, constructor, or an initializer, is termed as a local inner class.
- An inner class declared without a name within a code block such as the body of a method is called an anonymous inner class.
- A static nested class cannot directly refer to instance variables or methods of the outer class just like static methods but only through an object reference.

FPT UNIVERSITY

Student Activities

Employee

- name: string
- year_of_birth: int
- salary: int

- Employee()
- Employee(string, int, int)
- recruit()
- print()

Payment

- day int
- money int

- payment()
- timeTrack()

Payment is nest class

- payment(): print bill, set day =0; money=10
- timeTrack(): day=day+1
  salary = day*money

Main: declare 1 employee, set 2 day work and print his info, get money, print his info again

FPT UNIVERSITY

Student Activities

Person
- private name: string - private year_of_birth: int - private money : int
- public Person() - public input() - public output()

BankAccount
- private ID string - private money int
- public withdraw(int) - public transferMoney(BankAccount) - public deposit(int money)

BankAccount is nest class

- withdraw(int): decrease money of bank account and person
- transferMoney(BankAccount): decrease money of bank account and increase another
- deposit(BankAccount): increase money of bank account and person

Main: declare 1 person, 2 bank accounts, try withdraw, tranfer and deposite, print his info

ĐẠI HỌC FPT CẦN THƠ

cantho.fpt.edu.vn