Objects Interaction

Christian A. Rodríguez

Object Oriented Programming

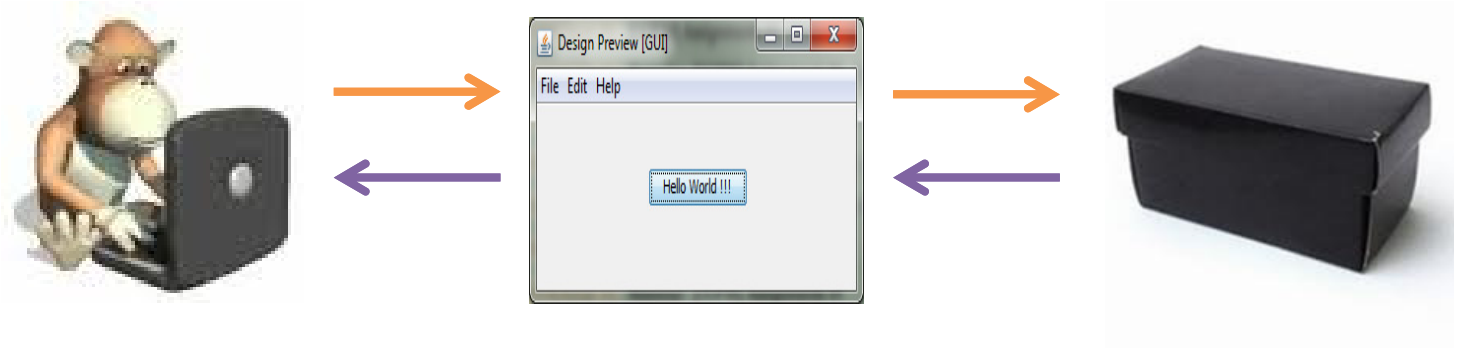
How software works?

User interface triggers

Objects collaboration

Software layers

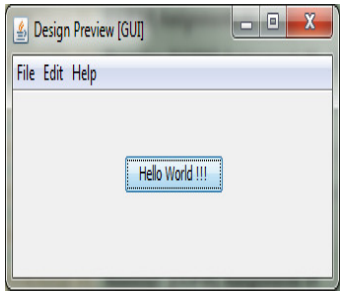
User perspective and the “black box”



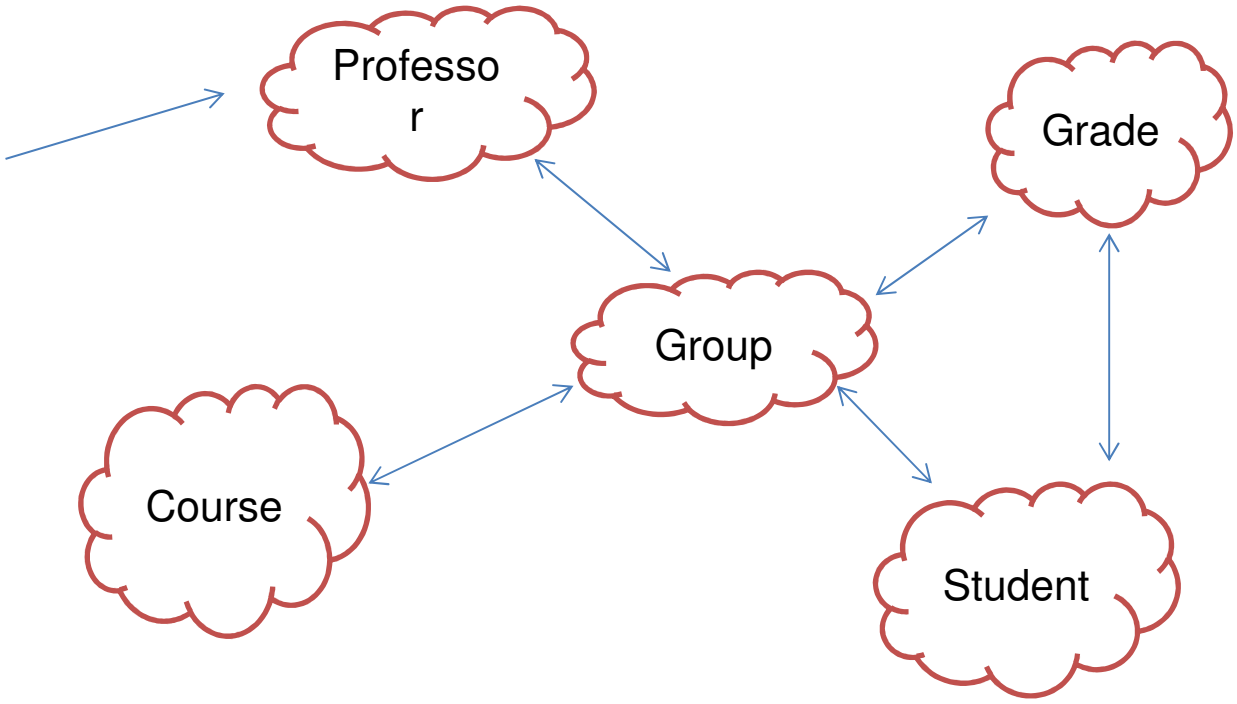
Users do not know how software internally works

UI used to be a bridge between user and the “black box”

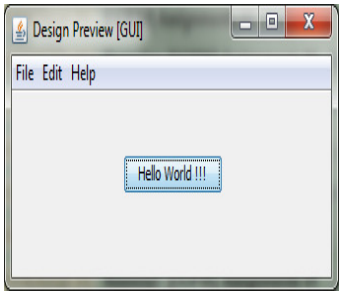
Black box contains networks of related objects



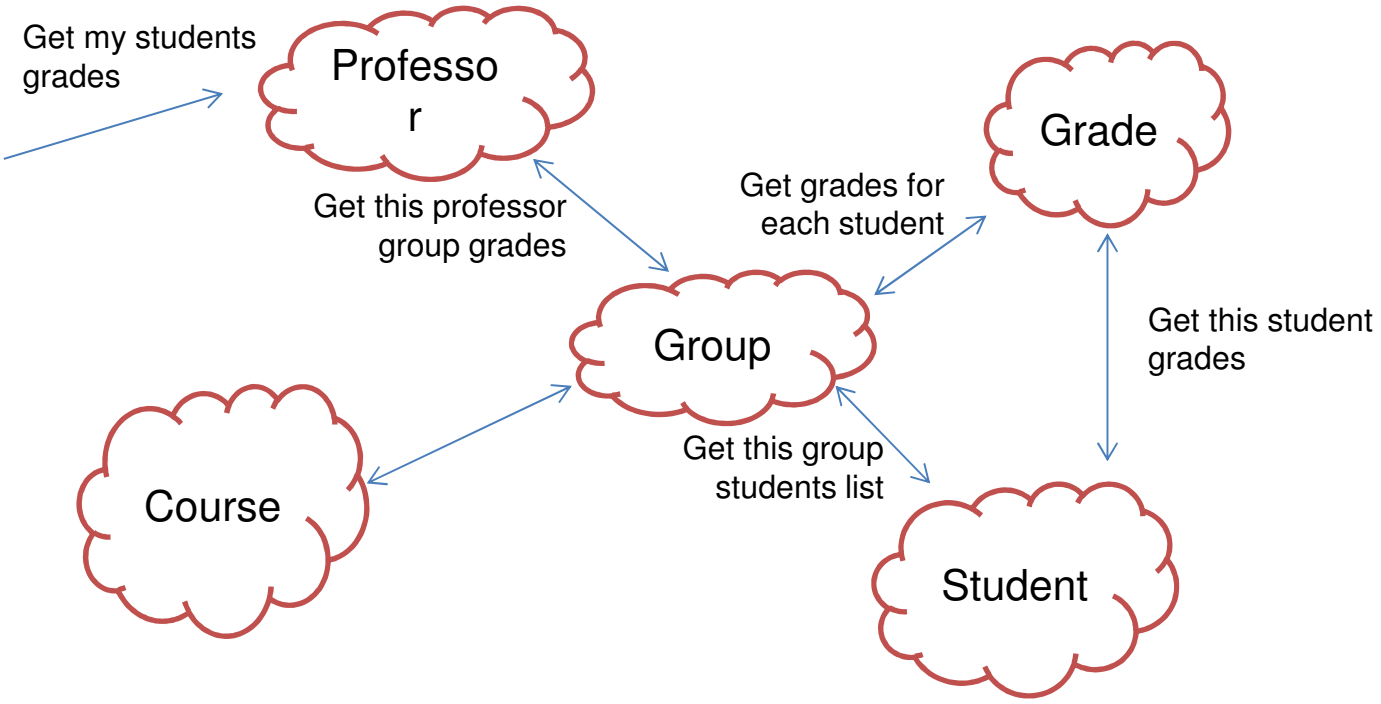
UI Click



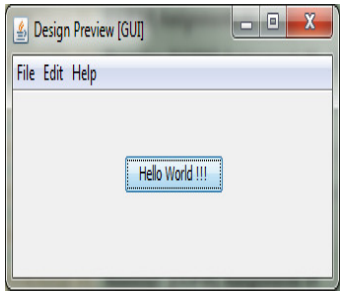
Objects collaborate and delegate tasks



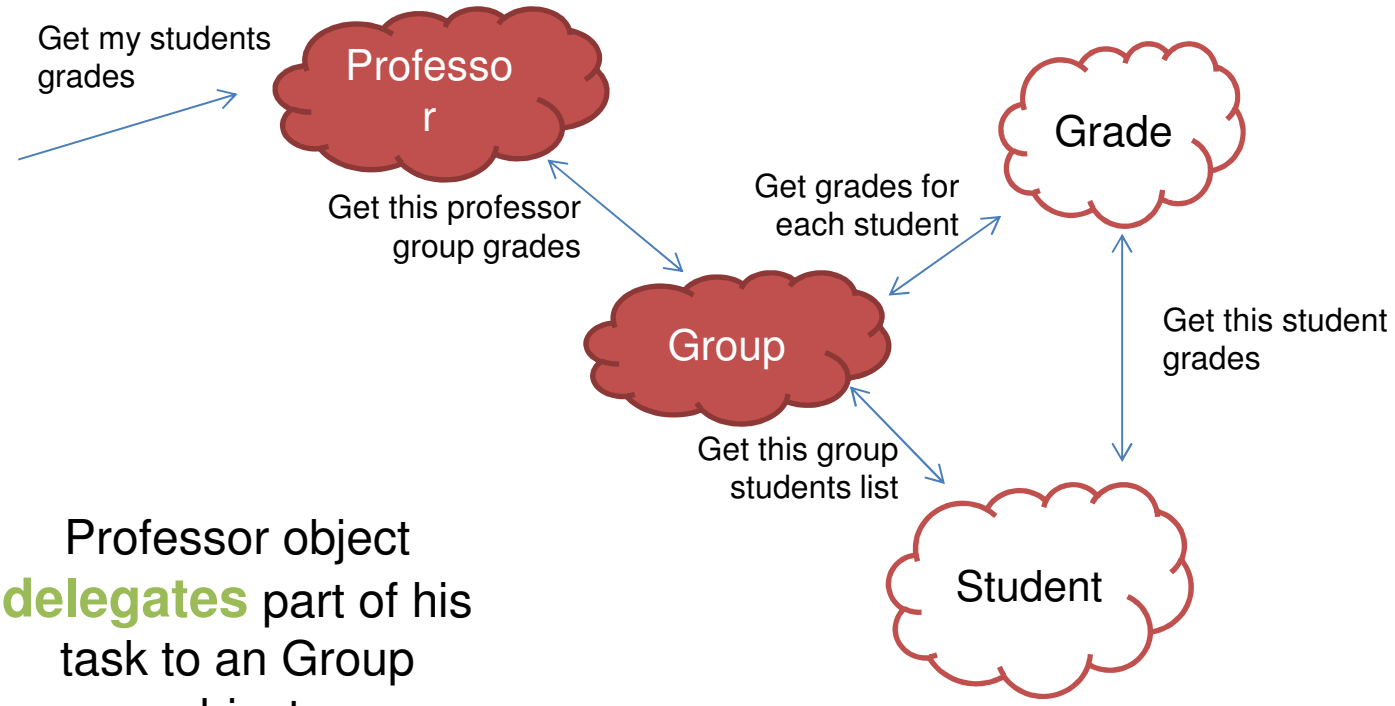
UI button
Click action



Delegation



UI button Click



Professor object **delegates** part of his task to an Group object

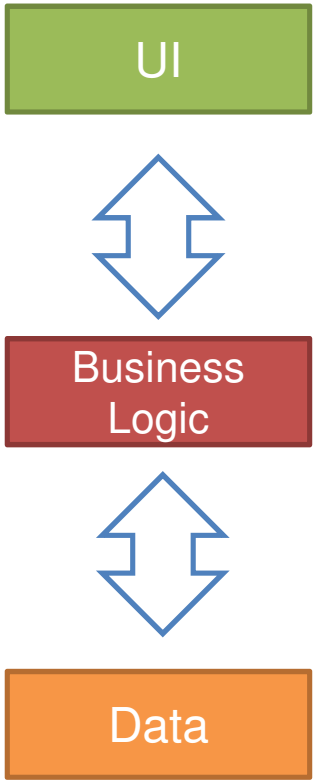
Delegation

Delegation is **invisible** for the object user

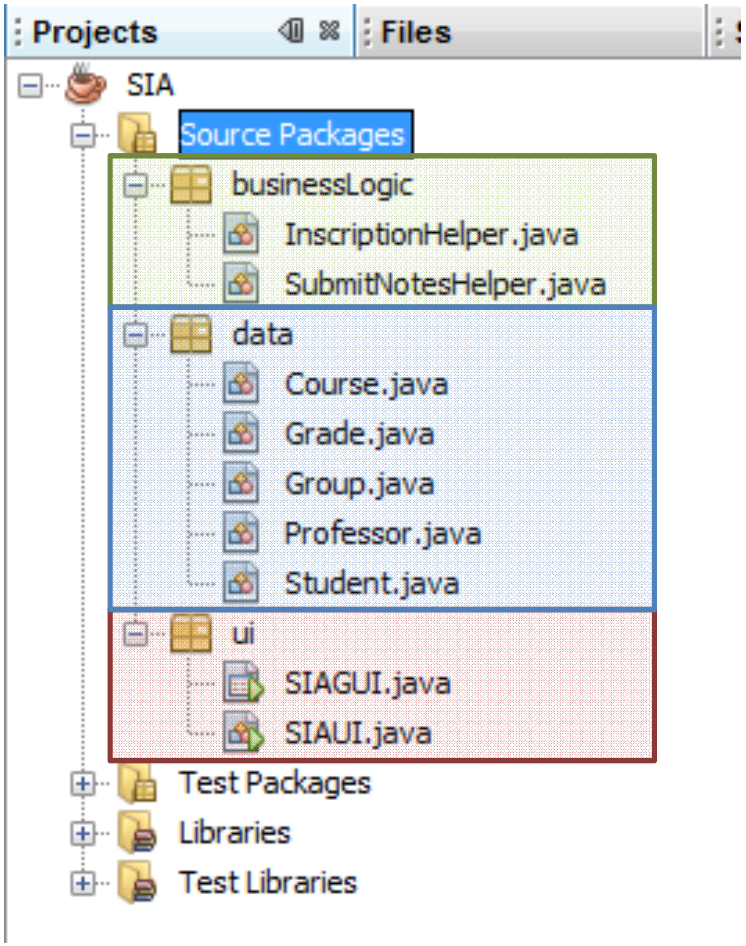
Delegation among software objects is exactly the same as delegation between people in the real world

How to organize all those objects?

- Software is generally structured in layers:
  - UI Layer
  - Business Logic Layer
  - Data Layer
- Each layer is responsible for a set o related tasks



How to organize all those objects?



Classes can be organized in packages

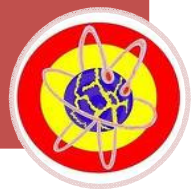
Each layer can be represented by a package

Classes in different packages must be imported in other classes

Layer responsibilities

- Contains encapsulated classes definitions.
- Contains data integrity validations.
- Do not contain business logic code.

Data layer



- Contains the “functional code”.

Business Logic layer

- Contains the UI code, forms, applets, web pages, etc.
- Do not contain business logic code.

UI Layer

Defining Methods

Declaring methods

Parameters and arguments

Method signatures & Methods overloading

Declaring methods - Knowing services

Object A: You

Object B: Your pet

Knowing services

You need to know which of your pet's services (methods) you want your pet to perform.

- ✓ Sit
- ✓ Fetch
- ✓ Stay
- ✓ Dance

Declaring methods -Passing data

Object A: You

Object B: Your pet

Passing data

Depending on the service request, object you may need to give your pet some additional information so that your pet knows exactly how to proceed

- Fetch beer
- Fetch stick
- Fetch newspaper

Declaring methods - Expecting something?

Object A: You

Object B: Your pet

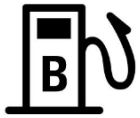
Expecting something?

Your pet in turn needs to know whether you expects your pet to report back the outcome of what it has been asked to do.

- Are you expecting your pet give you the beer?
- Are you expecting your pet give you the stick
- Are you expecting your pet give you the newspaper?

Declaring methods - Java perspective

Knowing services?

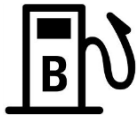


Passing data?

```
public static Course createCourse(String name, int number) {

    Course course = new Course();

    course.setName(name);
    course.setNumber(number);

    return course;
}
```

Expecting something?

Declaring methods in Java terms



Knowing services

Passing data

```
public static Course createCourse(String name, int number) {

    Course course = new Course();

    course.setName(name);
    course.setNumber(number);

    return course;
}
```

Expecting something?

Passing data

How it is the **passing** data process?

Parameters and arguments

Parameter are **variables**

Parameter

```
public void setFirstName(String firstName) {
    this.firstName = firstName;
}
```

Argument are **values**

```
Student student = new Student();
student.setFirstName("Bruce Wayne");
```

Argument

Parameters examples

```
public static Course createCourse(String name, int number) {
```

```
public static Grade createGrade(Group group, Student student, double Grade) {
```

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

Arguments examples

```
createCourse("Kung Fu", 265481032);
```

```
createGrade(group, student, 4.5);
```

Returning data

How it is the **returning** data process?

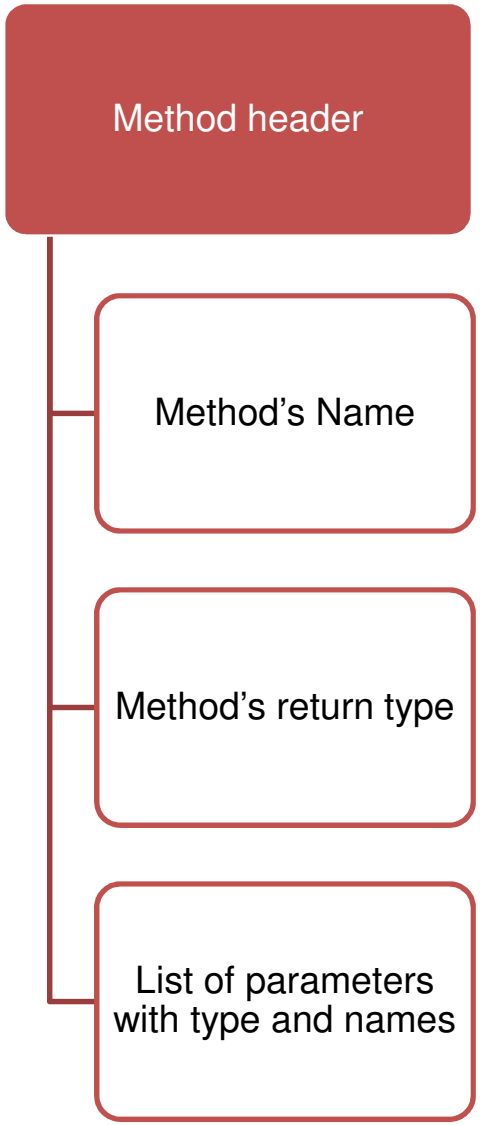
Methods can define zero or many returning points

```
public static Course createCourse(String name, int number) {

    if (name.length() == 0) {
        return null;
    } else {
        Course course = new Course();

        course.setName(name);
        course.setNumber(number);

        return course;
    }

}
```

Clients can use or not returning data

```
Student student = createStudent(266999,
        Calendar.getInstance().getTime(), "Bruce", "Wayne", "bwayne");

createStudent(266999, Calendar.getInstance().getTime(),
        "Bruce", "Wayne", "bwayne");
```

Method header



Java definition for a method

```
public static Course createCourse(String name, int number) {
```

This method header is:

```
Course createCourse(String name, int number)
```

Method signatures

Methods have signatures which indicates

- Method's Name
- Order, types and number of parameters

```
public static Course createCourse(String name, int number) {
```

This Method signature is:

`createCourse(String , int)`

The method **createCourse** declares **two** parameters of type **String** and **int** respectively

Method signature is unique

Methods overloading

Methods from the **same class** can be offered with a **unique name** but with **different signature**

println overloading

```
println()                 void
println(Object o)         void
println(String string)    void
println(boolean bln)      void
println(char c)           void
println(char[] chars)     void
println(double d)         void
println(float f)          void
println(int i)            void
println(long l)           void
```

```
public void println() {
```

```
public void println(char[] chars) {
```

```
public void println(long l) {
```

```
public void println(String string) {
```

```
public void println(boolean bln) {
```

```
public void println(char c) {
```

```
public void println(int i) {
```

```
public void println(double d) {
```

```
public void println(float f) {
```

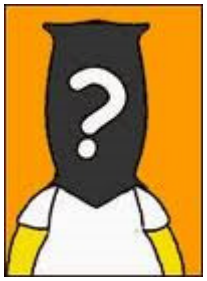
```
public void println(Object o) {
```

println overloading

```
System.out.println("String");
System.out.println(1);
System.out.println(1.2);
System.out.println(1.5f);
System.out.println();
System.out.println('c');
System.out.println(true);
System.out.println(student);
```

```
println()                    void
println(Object o)            void
println(String string)       void
println(boolean bln)         void
println(char c)              void
println(char[] chars)        void
println(double d)            void
println(float f)             void
println(int i)               void
println(long l)              void
```

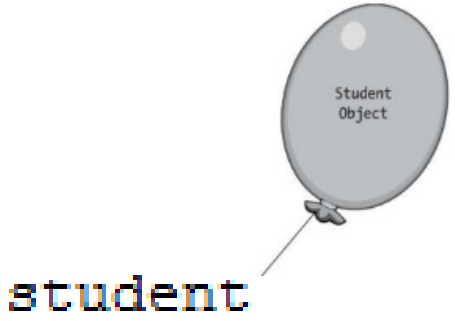
Compiler choose the correct method checking the list of arguments passed to parameters



Constructors

Do you remember how to instantiate?

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



Constructors

This is the Student class constructor

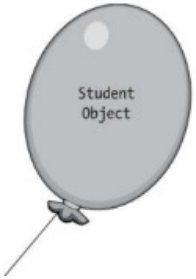
```
Student student = new Student();
```

Invoking a constructor serves as a request to the JVM to construct (instantiate) a brand-new object

Constructors

Constructors are special type of procedures which **are responsible to ask the JVM to inflate a new helium balloon**

JVM



Default constructor

```java
public class Student {

    private String name;
    private int age;

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

=

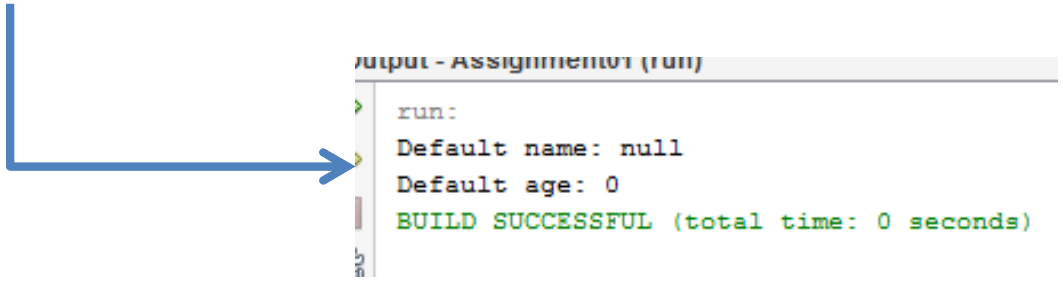
```java
public class Student {

    private String name;
    private int age;

    public Student() {
    }

    public int getAge() {
        return age;
    }

    public String getName() {
        return name;
    }
}
```

If there is no defined constructors, the JVM use the by **default constructor**

Default constructor

Default constructors setting all attributes to their zero-equivalent default values

```
public class StudentTest {

    public static void main(String[] args) {

        Student myStudent = new Student();

        System.out.println("Student name: " + myStudent.getName());
        System.out.println("Student age: " + myStudent.getAge());
    }
}
```



Explicit constructors

```
public class Student {
    // ...


    // ...
    public Student(String name, int age){
        // Code code code
    }
    // ...
```

We use **explicit constructors** if we wish to do something more “interesting” to initialize an object when it is first instantiated

Explicit constructors rules

Constructor's name **must be exactly the same** as the name of the class for which we're writing the constructor

Constructors works like another method, can define a **list of parameters**

```
public class Student {
    // ...

    // ...
    public Student(String name, int age){
        // Code code code
    }
    // ...
```

We **cannot specify a return type** for a constructor; by definition, a constructor returns a **reference to a newly created object of the class type**

Passing parameters to constructors

```java
public class Student {

    private String name;
    private int age;

    public Student(String name, int age){
        this.setAge(age);
        this.setName(name);
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }
    // ...
```

Class definition

```java
public class StudentTest {

    public static void main(String[] args) {

        Student myStudent = new Student("Bob", 31);

        System.out.println("Student name: "
                + myStudent.getName());
        System.out.println("Student age: "
                + myStudent.getAge());
    }
}
```

Test Class

```
run:
Student name: Bob
Student age: 31
BUILD SUCCESSFUL (total time: 0 seconds)
```

Be careful

If there is at least one constructor defined we **cannot** use the default constructor

```
public class StudentTest {

    public static void main(Strin

        Student myStudent = new Student();
        //..
```

```
cannot find symbol
  symbol:   constructor Student()
  location: class lesson.Student
--
(Alt-Enter shows hints)
```

Replacing the Default Parameterless Constructor

```
// ...
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}
// ...
```

```
public class StudentTest {

    public static void main(String[] args) {
        Student myStudent = new Student();
        System.out.println("Student name: "
                + myStudent.getName());
        System.out.println("Student age: "
                + myStudent.getAge());
    }
}
```

```
run:
Student name: UNDEFINED
Student age: -1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Overloading Constructors

It is possible to overload Constructors like any other method

```
//
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}
```

Constructor 1 signature

`Student ( )`

```
public Student(String name){
    this.setName(name);
    this.setAge(-1);
}
```

Constructor 2 signature

`Student ( String )`

```
public Student(String name, int age){
    this.setAge(age);
    this.setName(name);
}
// ...
```

Constructor 3 signature

`Student ( String , int )`

Constructors reuse

```
// ...
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}

public Student(String name){
    this.setName(name);
    this.setAge(-1);
}

public Student(String name, int age){
    this.setAge(age);
    this.setName(name);
}
// ...
```

Code duplication

Constructors reuse

It is possible to reuse Constructors using the keyword **this**

```
// ...
public Student(){
    this.setName("UNDEFINED");
    this.setAge(-1);
}

public Student(String name){
    this.setName(name);
    this.setAge(-1);
}

public Student(String name, int age){
    this.setAge(age);
    this.setName(name);
}
// ...
```

```
// ...
public Student() {
    this("UNDEFINED", -1);
}

public Student(String name) {
    this(name, -1);
}

public Student(String name, int age) {
    this.setAge(age);
    this.setName(name);
}
// ...
```

Reusing constructors can avoid duplication of code

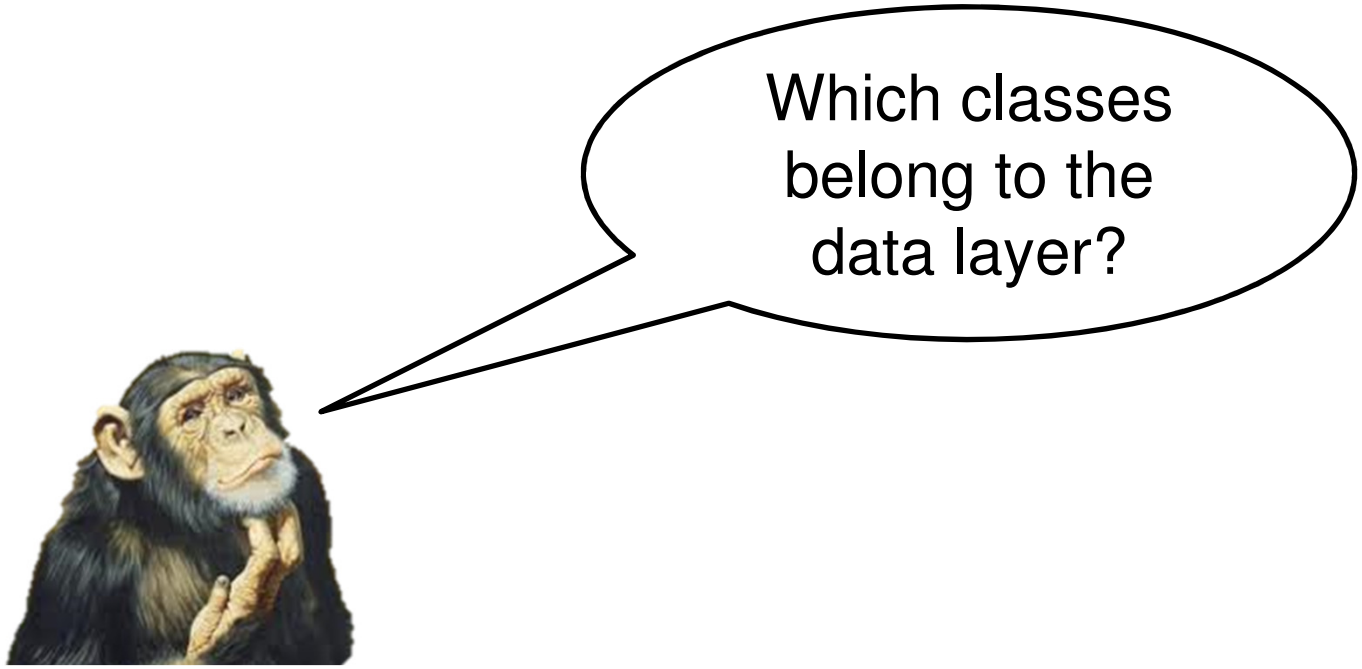
Review: Tic Tac Toe

Layers and packages

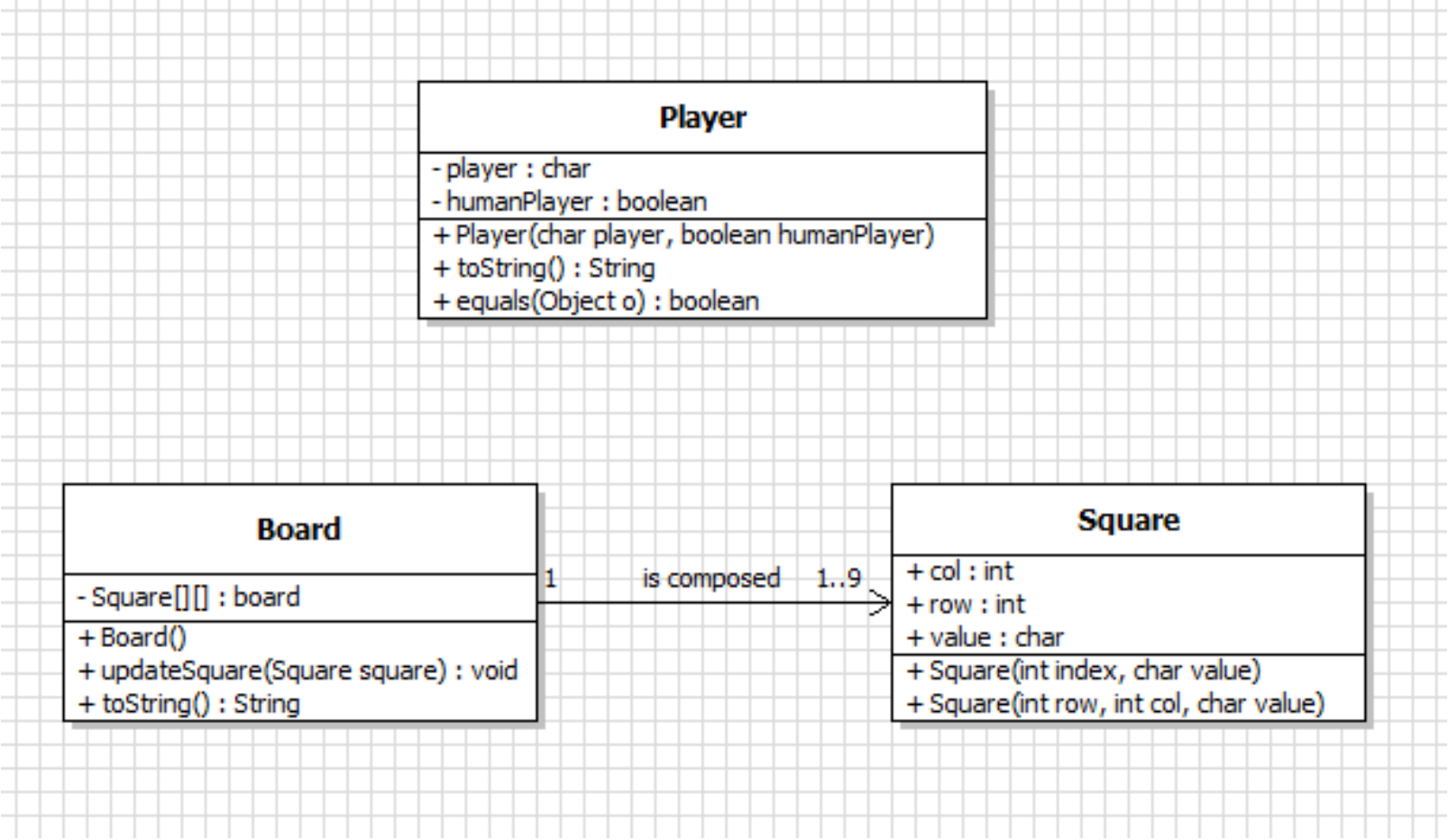
Constructors and access modificators

Relation between objects

Data Layer



Data layer


Player
- player : char
- humanPlayer : boolean
+ Player(char player, boolean humanPlayer)
+ toString() : String
+ equals(Object o) : boolean
Board
- Square[][] : board
+ Board()
+ updateSquare(Square square) : void
+ toString() : String
1
is composed
1..9
Square
+ col : int
+ row : int
+ value : char
+ Square(int index, char value)
+ Square(int row, int col, char value)

Board class

Overriding default constructor

```
public class Board {

    private Square[][] board;

    public Board() {

        char value = '0';
        board = new Square[3][3];

        for (int row = 0; row < board.length; row++) {
            for (int col = 0; col < board.length; col++) {
                Square square = new Square(row, col, (char) (++value));
                board[row][col] = square;
            }
        }
    }

    public Square[][] getBoard() {...}

    public void updateSquare(Square square) {...}

    @Override
    public String toString() {...}
}
```

Printing user defined objects method

```
public static void printBoard(Board board) {
    System.out.println(board);
}
```

```
tictactoe.data.Board@530daa
```

toString method

toString method allow us **override the default way how the object is printed**

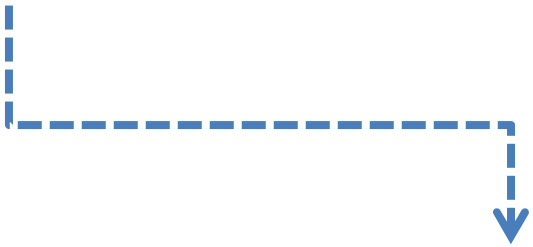
Important, do not forget it

```
@Override
public String toString() {
    String printBoard = "\n";

    for (int row = 0; row < board.length; row++) {
        printBoard = printBoard.concat("\t|");
        for (int col = 0; col < board.length; col++) {
            printBoard = printBoard.concat(
                    String.valueOf(board[row][col]).concat("|"));
        }
        printBoard = printBoard.concat("\n");
    }
    return printBoard;
}
```

The return type is String

Overriding toString method

```
public static void printBoard(Board board) {
    System.out.println(board);
}
```



```
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |
```

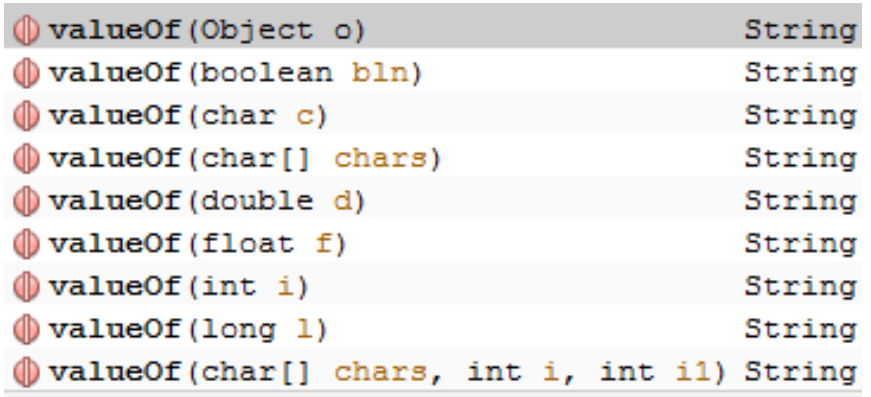
```
public class Square {

    private int row;
    private int col;
    private char value;

    public Square(int row, int col, char value) {
        this.row = row;
        this.col = col;
        this.value = value;
    }

    public Square(int index, char value) {
        this.row = (index - 1) / 3;
        this.col = (index - 1) % 3;
        this.value = value;
    }

    public int getCol() {...}

    public void setCol(int col) {...}

    public int getRow() {...}

    public void setRow(int row) {...}

    public char getValue() {...}

    public void setValue(char value) {...}

    @Override
    public String toString() {...}
}
```

Overloading constructors

Overriding toString method

```
@Override
public String toString() {
    return String.valueOf(this.getValue());
}
```

String.valueOf( ) is used to cast variables to String



Player class

```
public class Player {

    private char player;
    private boolean humanPlayer;

    public Player(char player, boolean humanPlayer) {...}

    public char getPlayer() {...}

    public void setPlayer(char player) {...}

    public boolean isHumanPlayer() {...}

    @Override
    public String toString() {...}

    @Override
    public boolean equals(Object obj) {...}
}
```

Overriding toString method

```
@Override
public String toString() {

    String printPlayer = "I am the player "
            + String.valueOf(this.getPlayer()) + " and I am ";

    if (this.isHumanPlayer()) {
        printPlayer = printPlayer.concat("HUMAN");
    } else {
        printPlayer = printPlayer.concat("A ROBOT");
    }
    return printPlayer;
}
```

How do you know if two user defined objects are equal?

Overriding equals method

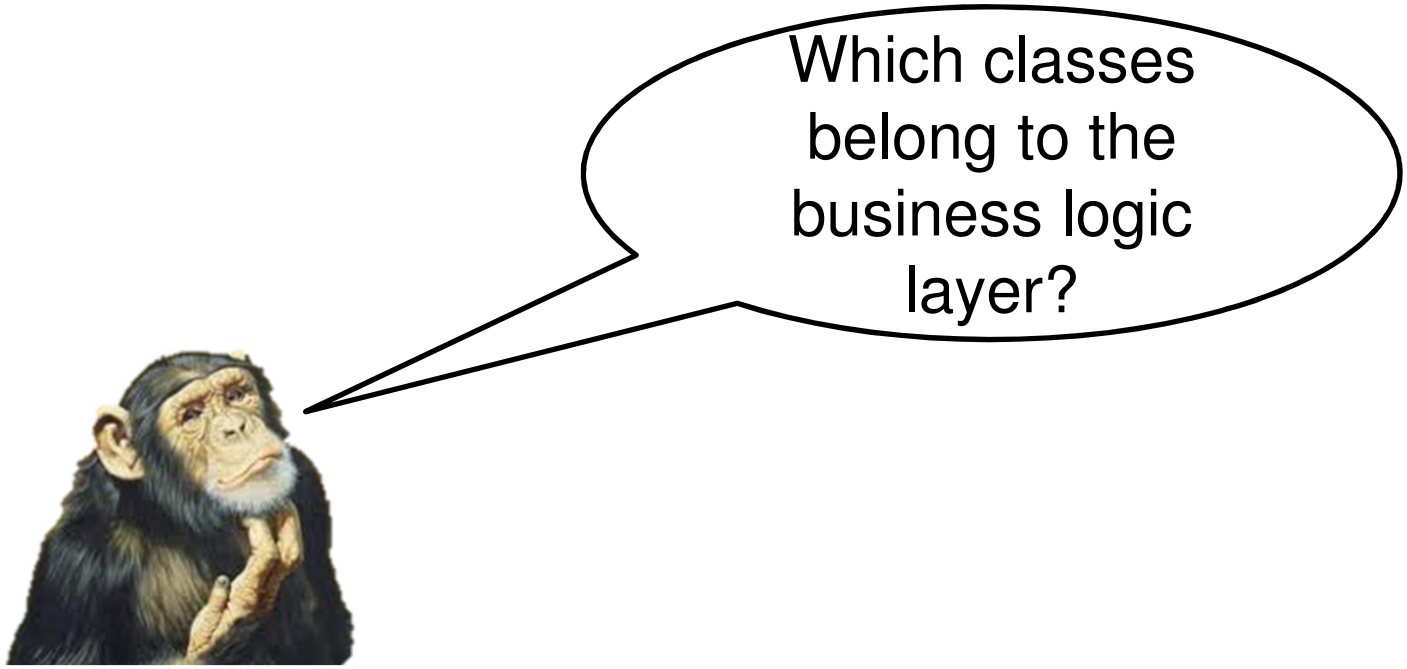
equals method allow us to **override the default way how two user defined objects are equals or not**

Important, do not forget it

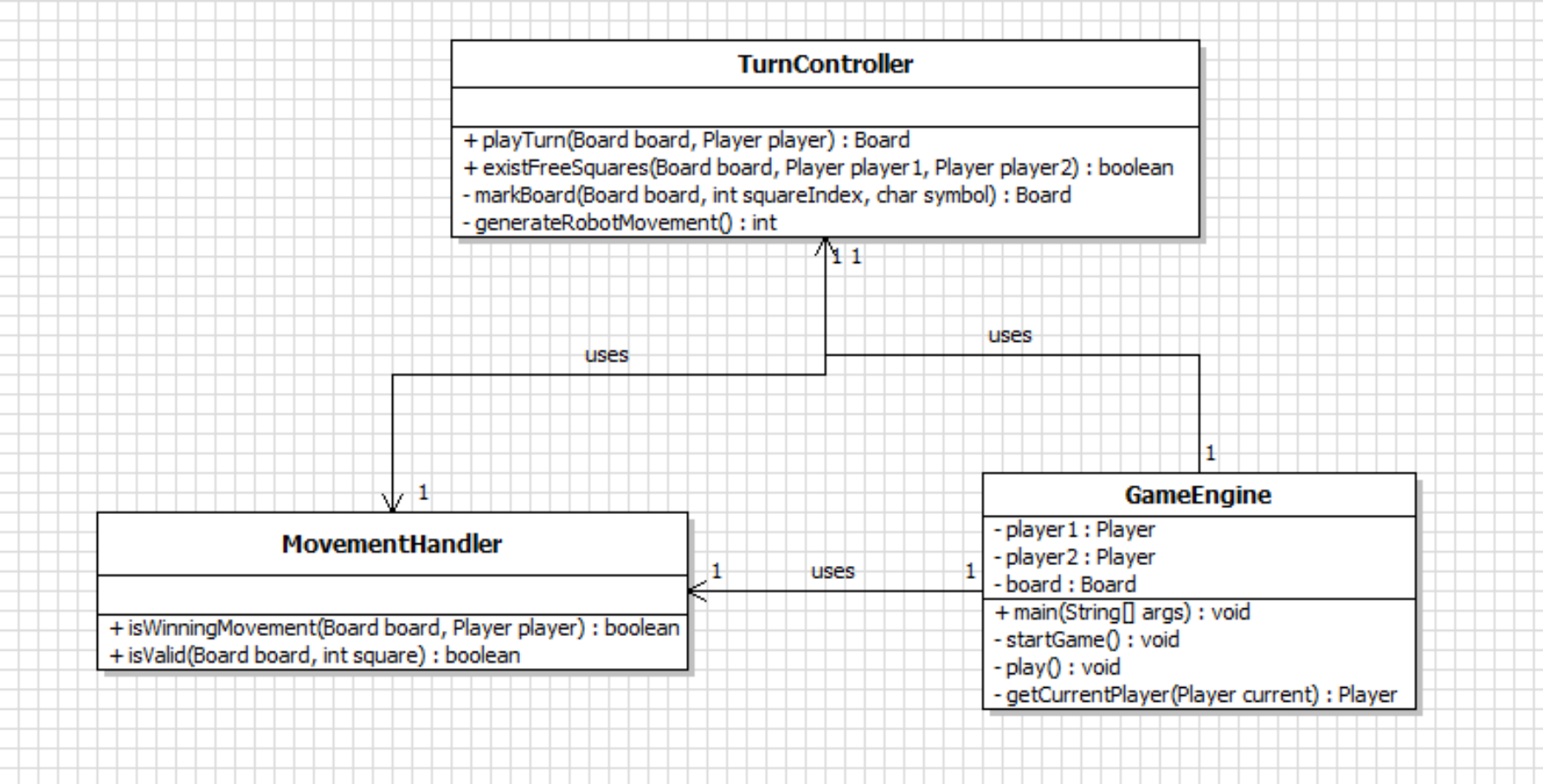
```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Player other = (Player) obj;
    if (this.player != other.player) {
        return false;
    }
    return true;
}
```

If the two objects have the same player (Symbol) are equals

Business logic Layer



Business logic layer



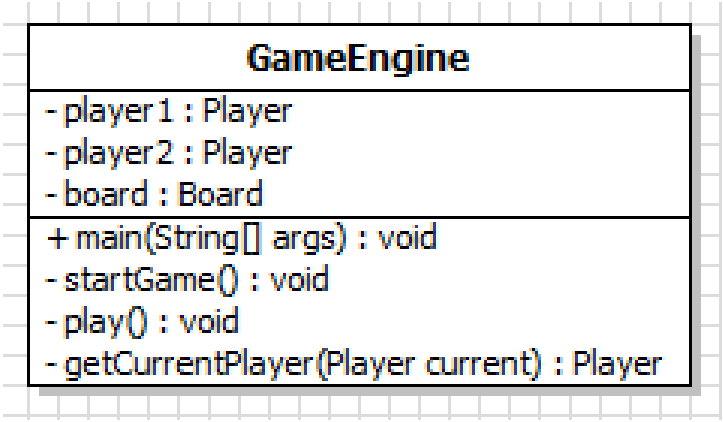
GameEngine Class

This is the game starting point

```
import tictactoe.data.Board;
import tictactoe.data.Player;
import tictactoe.ui.UI;

public class GameEngine {

    private static Player player1;
    private static Player player2;
    private static Board board;

    public static void main(String[] args) {...}

    private static void startGame() {...}

    private static void play() {...}

    private static Player getCurrentPlayer(Player current) {...}
}
```

Methods can be private

GameEngine Class

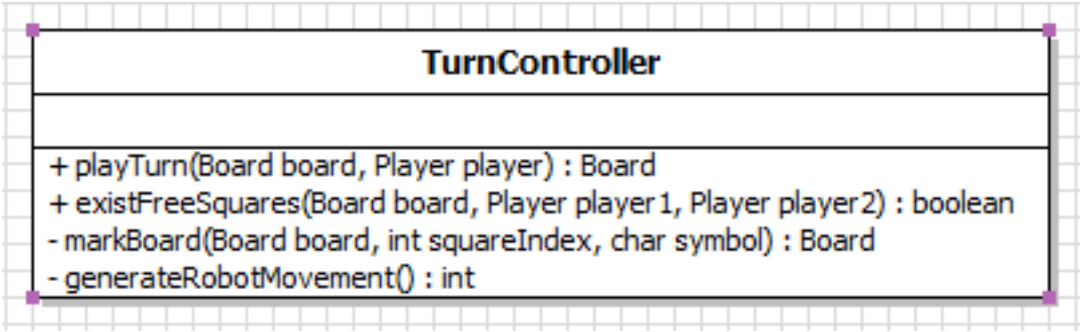


Class	Method signature	Function
GameEngine	startGame ( )	Initialize objects and variables Call the method play
	play ( )	Iterate until win or finish condition is reached
	getCurrentPlayer (Player)	Change the current player at the end of each turn

TurnController Class

```
import java.util.Random;
import tictactoe.data.Board;
import tictactoe.data.Player;
import tictactoe.data.Square;
import tictactoe.ui.UI;

public class TurnController {

    public static Board playTurn(Board board, Player player) {...}

    private static Board markBoard(Board board, int squareIndex, char symbol) {...}

    public static boolean existFreeSquares(Board board, Player player1, Player player2) {...}

    private static int generateRobotMovement() {...}
}
```

TurnController Class

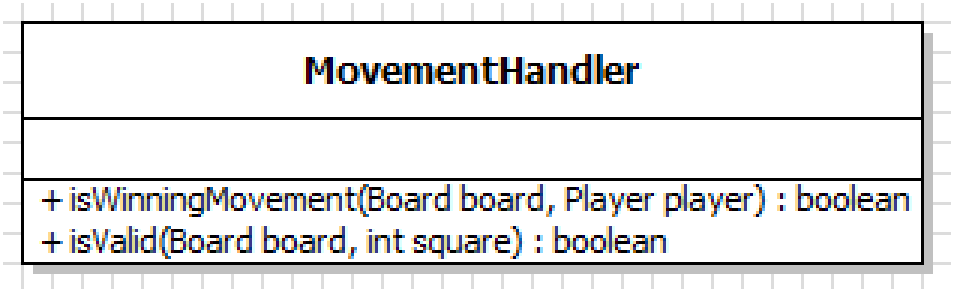


Class	Method signature	Function
TurnController	playTurn (Board , Player)	Handle movement turn Call movement validator Call board modifier
	markBoard (Board , int , char)	Modify board after a valid play
	existFreeSquares (Board , Player , Player)	Chek if there are available squares to play
	generateRobotMovement ( )	Generate random robot movement

MovementHandler Class

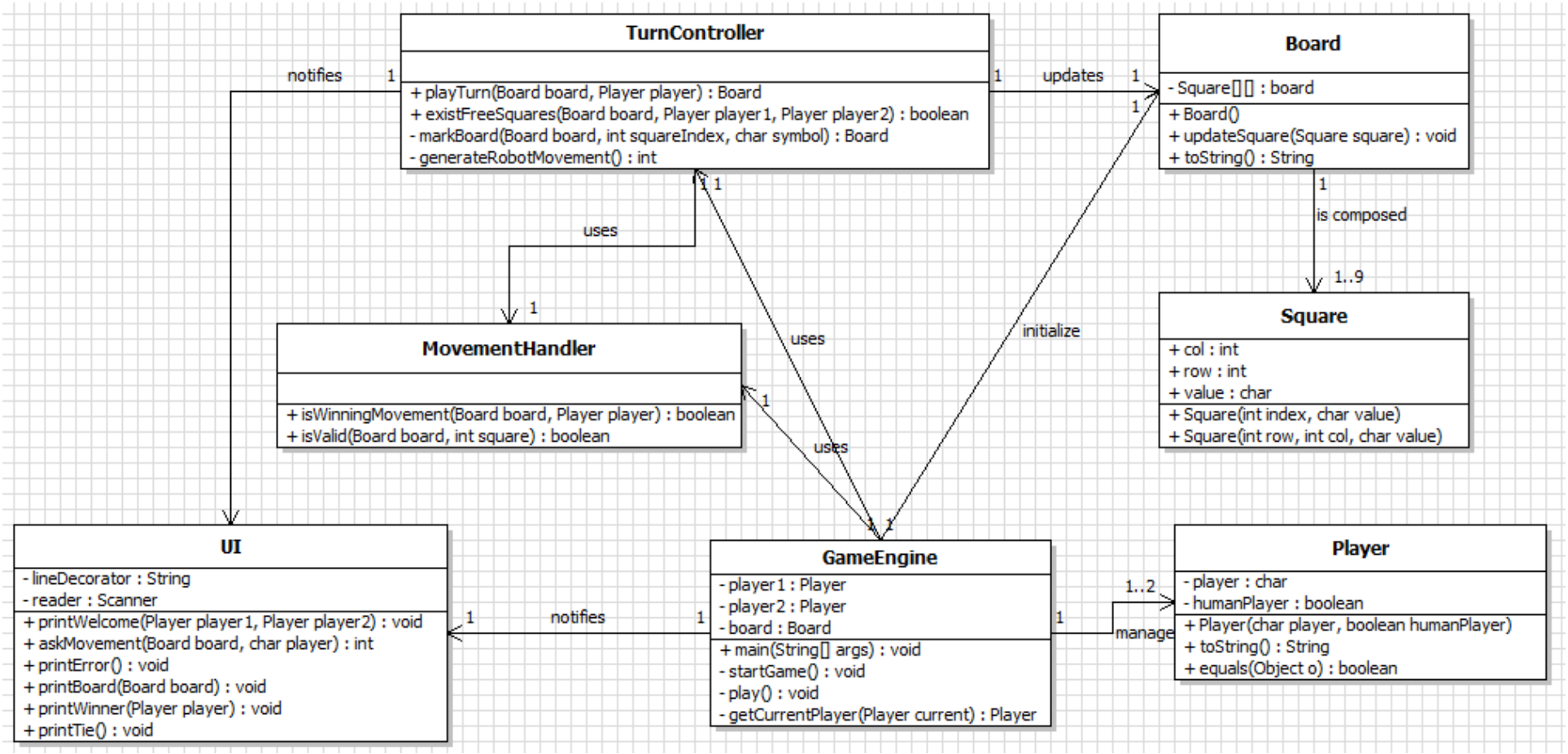
```
import tictactoe.data.Board;
import tictactoe.data.Player;
import tictactoe.data.Square;

public class MovementHandler {

    public static boolean isValid(Board board, int square) {...}

    public static boolean isWinningMovement(Board board, Player player) {...}
}
```

MovementHandler Class

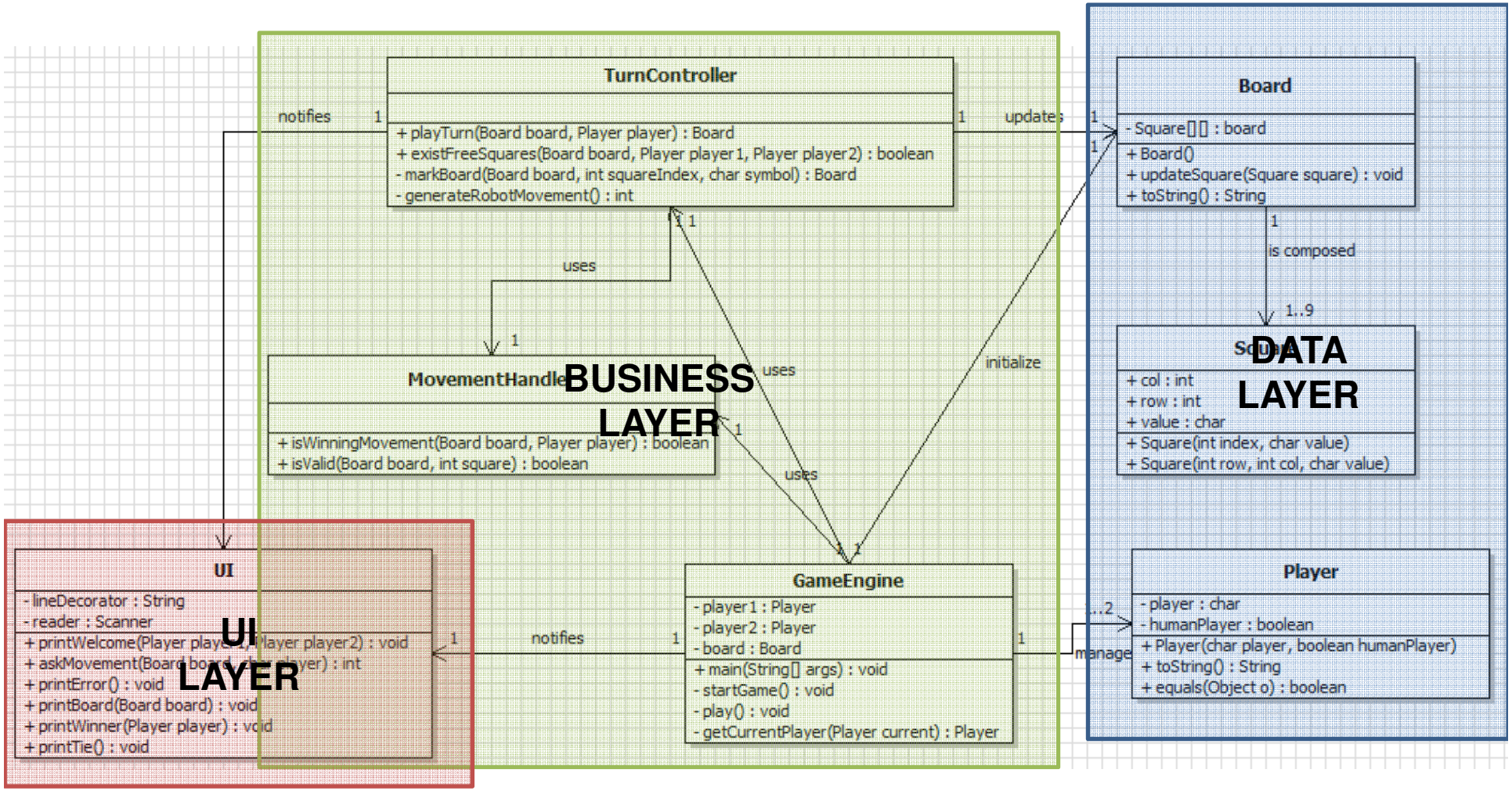


Class	Method signature	Function
MovementHandler	isValid (Board , int)	Check if a square selection is available to be marked
	isWinningMovement (Board , Player)	Check if the last movement causes the player's victory

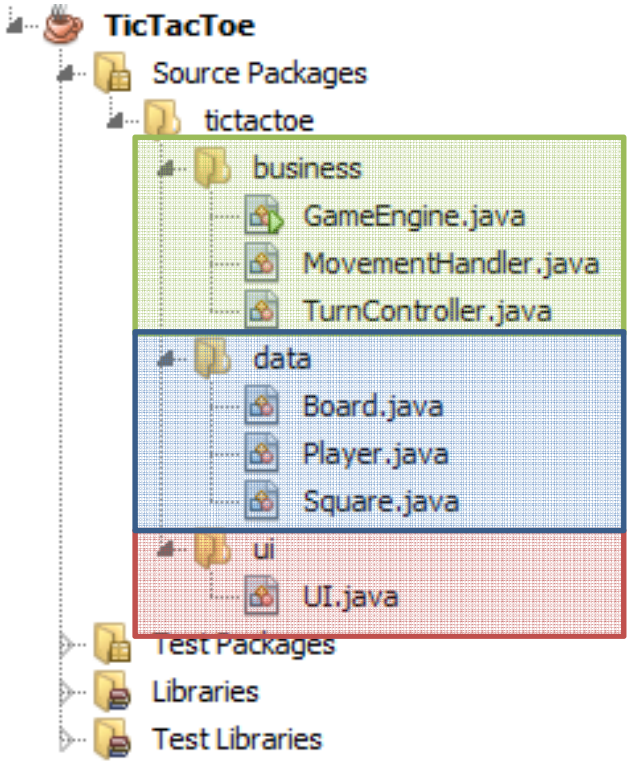
UML Class diagram



UML Class diagram



Each layers is represented as a package



- Three layers
  - Data
  - Business logic
  - UI

Tic Tac Toe

Check the code here

References

- [Barker] J. Barker, *Beginning Java Objects: From Concepts To Code*, Second Edition, Apress, 2005.