Practice II

Lecture 8

Testing

- Testing
  - A way of showing the correctness of software
- Phases
  - **Unit testing**
    - To test each module (unit, or component) independently
    - Mostly done by developers of the modules
  - **Integration and system testing**
    - To test the system as a whole
    - Often done by separate testing or QA team
  - **Acceptance testing**
    - To validate system functions for (and by) customers or user

Unit testing

Definitions

- *Testing* is the process of showing that a program works for certain inputs.
- *A unit* is a module or a small set of modules.
  - In Java, a unit is a class or interface, or a set of them, e.g.,
  - An interface and 3 classes that implement it, or
  - A public class along with its helper classes.
- *Unit testing* is testing of a unit.

Why Unit Testing?

- *Code isn't right if it's not tested*.
- Practical
  - Most programmers rely on testing, e.g., Microsoft has 1 tester per developer.
  - You could get work as a tester.
- Divide-and-conquer approach
  - Split system into units.
  - Debug unit individually.
  - Narrow down places where bugs can be.
  - Don't want to chase down bugs in other units.

Why Unit Testing? (Cont.)

Support regression testing

- You can make changes to lots of code and know if you broke something.
- Can make big changes with confidence.

The main idea

- Build systems in layers
  - Starts with classes that don't depend on others.
  - Continue testing building on already tested classes.
- Benefits
  - When testing a module, ones it depends on are reliable.

Program to Test

```
public final class IMath {

  /**
   * Returns an integer approximation to the square root of x.
   */
  public static int isqrt(int x) {
     int guess = 1;
     while (guess * guess < x) {
        guess++;
     }
     return guess;
  }
}
```

Conventional Testing

```
/** A class to test the class IMath. */
public class IMathTestNoJUnit {
   /** Runs the tests. */
   public static void main(String[] args) {
       printTestResult(0);
       printTestResult(1);
       printTestResult(2);
       printTestResult(3);
       printTestResult(4);
       printTestResult(7);
       printTestResult(9);
       printTestResult(100);
  }
 private static void printTestResult(int arg) {
       System.out.print("isqrt(" + arg + ") ==>  ");
       System.out.println(IMath.isqrt(arg));
  }
}
```

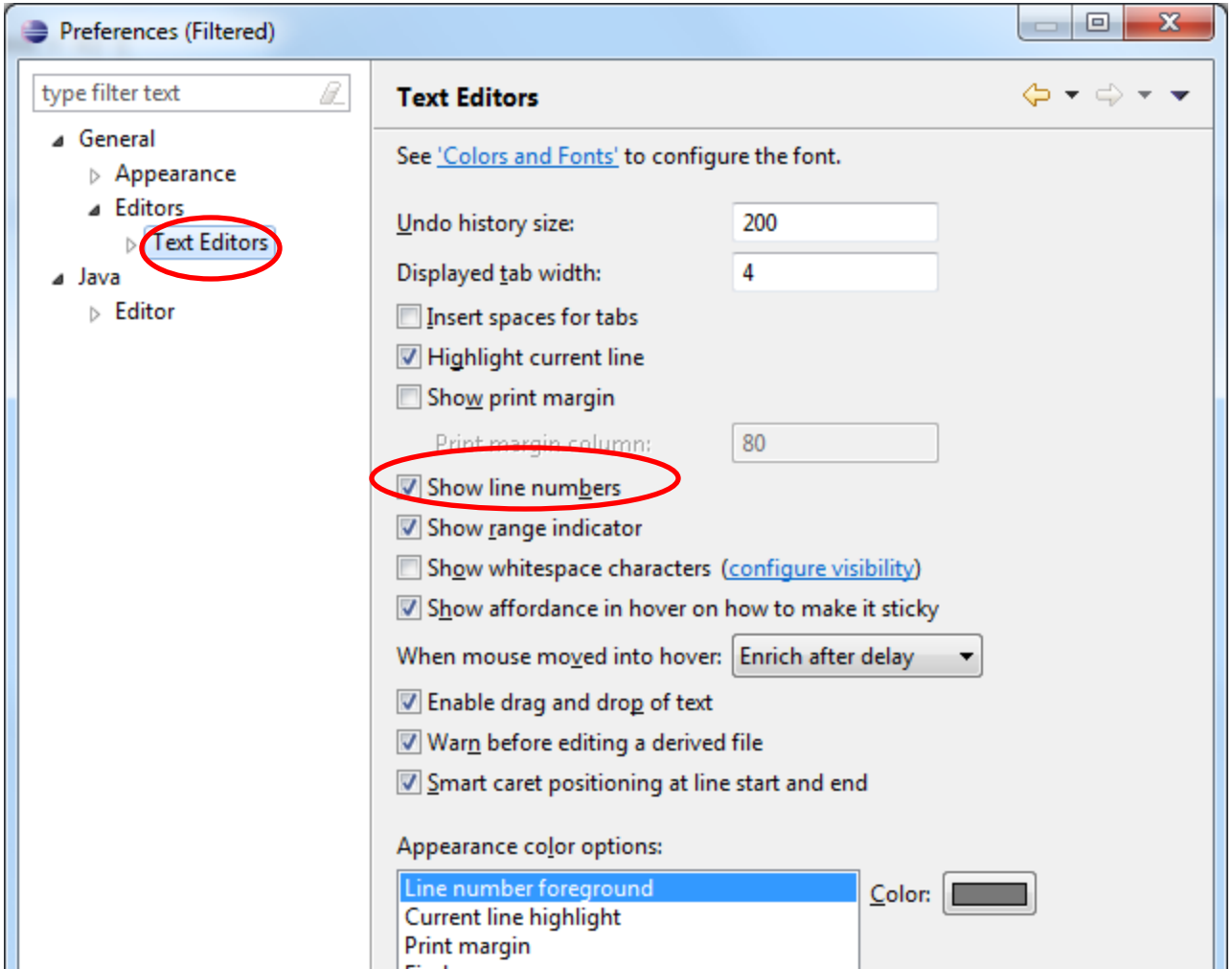
Conventional Test Output

```
Isqrt(0) ==> 1
Isqrt(1) ==> 1
Isqrt(2) ==> 2
Isqrt(3) ==> 2
Isqrt(4) ==> 2
Isqrt(7) ==> 3
Isqrt(9) ==> 3
Isqrt(100) ==> 10
```

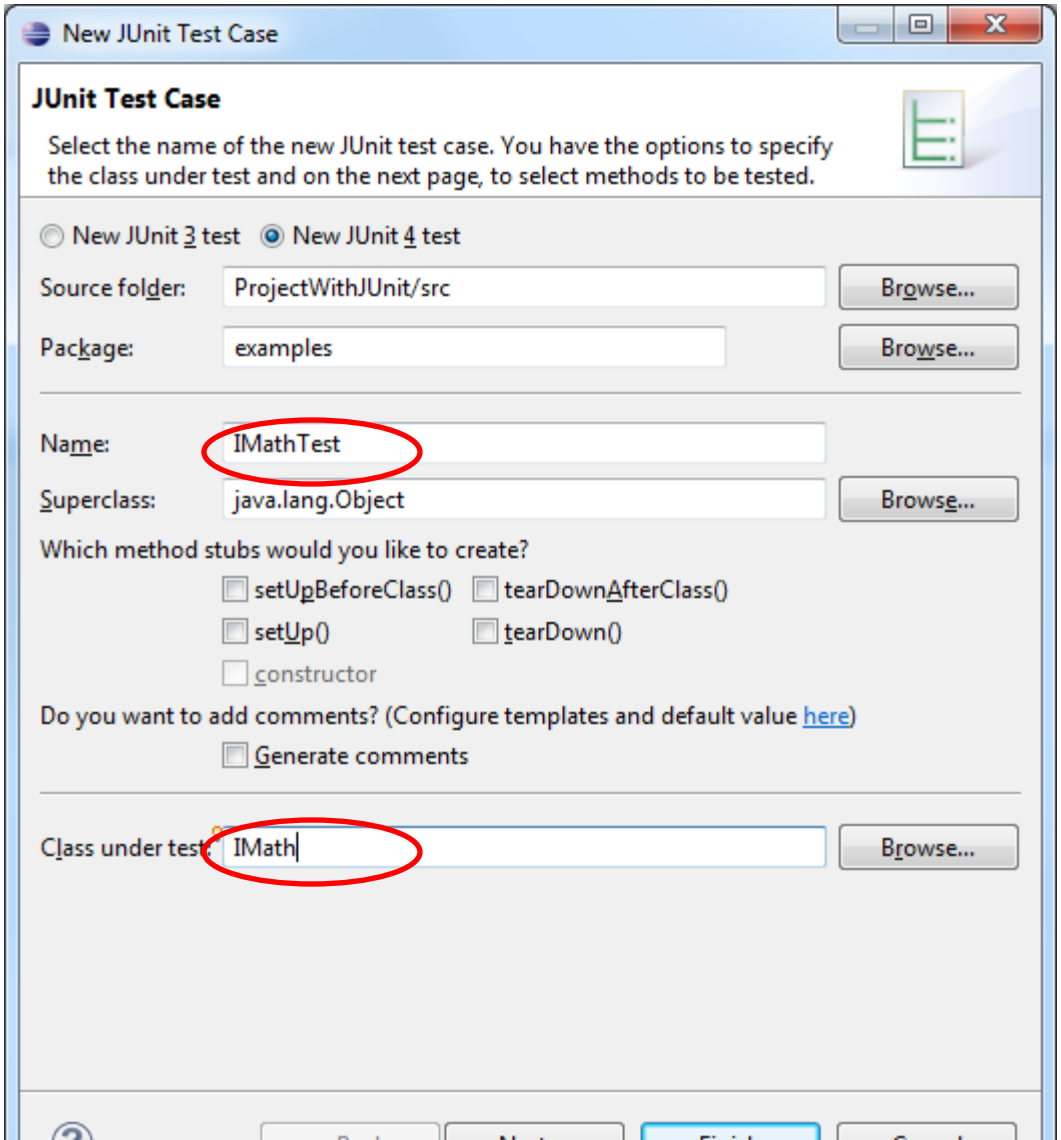
- What does this say about the code? Is it right?
- What's the problem with this kind of test output?

To enable line numbers in Eclipse

Rclick -> Preferences



Adding New -> JUnit test case



Testing with JUnit: Test case

```
package examples;

import static org.junit.Assert.*;
import org.junit.Test;
public class IMathTest {

@Test
/** Tests isqrt. */
  public void testIsqrt() {
    assertEquals(0, IMath.isqrt(0)); // line 13
    assertEquals(1, IMath.isqrt(1));
    assertEquals(1, IMath.isqrt(2));
    assertEquals(1, IMath.isqrt(3));
    assertEquals(2, IMath.isqrt(4));
    assertEquals(2, IMath.isqrt(7));
    assertEquals(3, IMath.isqrt(9));
    assertEquals(10, IMath.isqrt(100));
  }
}
```

Test suite

- Includes all the unit tests in the project
- Add -> New ->Other -> Junit->Junit Test Suite

Test Suite syntax

```
package examples;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ IMathTest.class })
public class AllTests {

}
```

Run as -> JUnit test

Exercise

Write a JUnit Test Case to test the following class

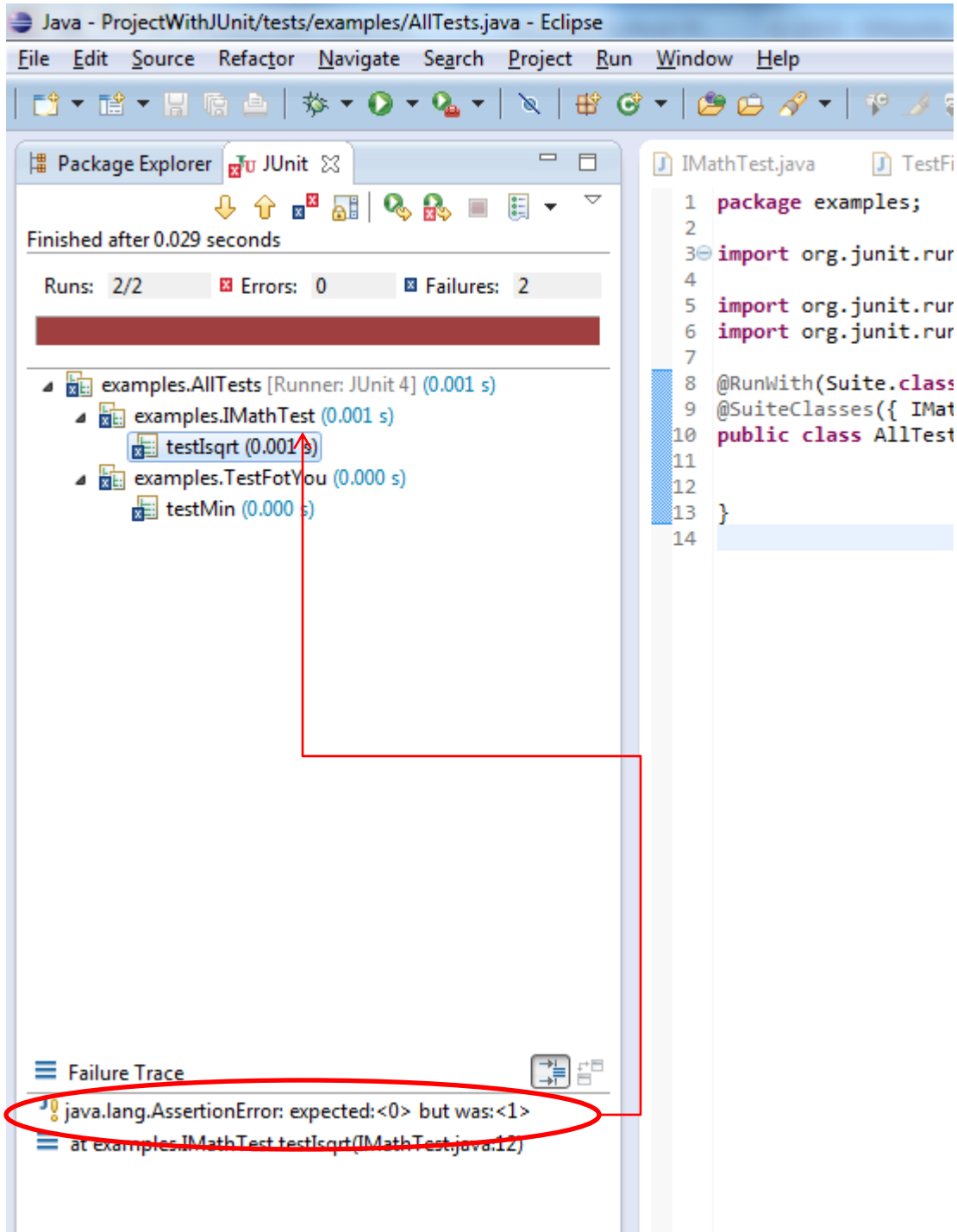
```
public class ForYou {
  /** Return the minimum of x and y. */
   public static int min(int x, int y) { ... }
}
```

Testing with different inputs

```
package examples;

import static org.junit.Assert.*;
import org.junit.Test;

public class TestForYou {

        @Test
        public void testMin() {
                assertEquals(0, ForYou.min(0,1)); // line 7
                assertEquals(-1, ForYou.min(-1,1));
                assertEquals(-1, ForYou.min(1,-1));
        }

}
```

Adding new test to the Test Suit

```
package examples;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ IMathTest.class ,TestForYou.class})
public class AllTests {


}
```



Displaying errors

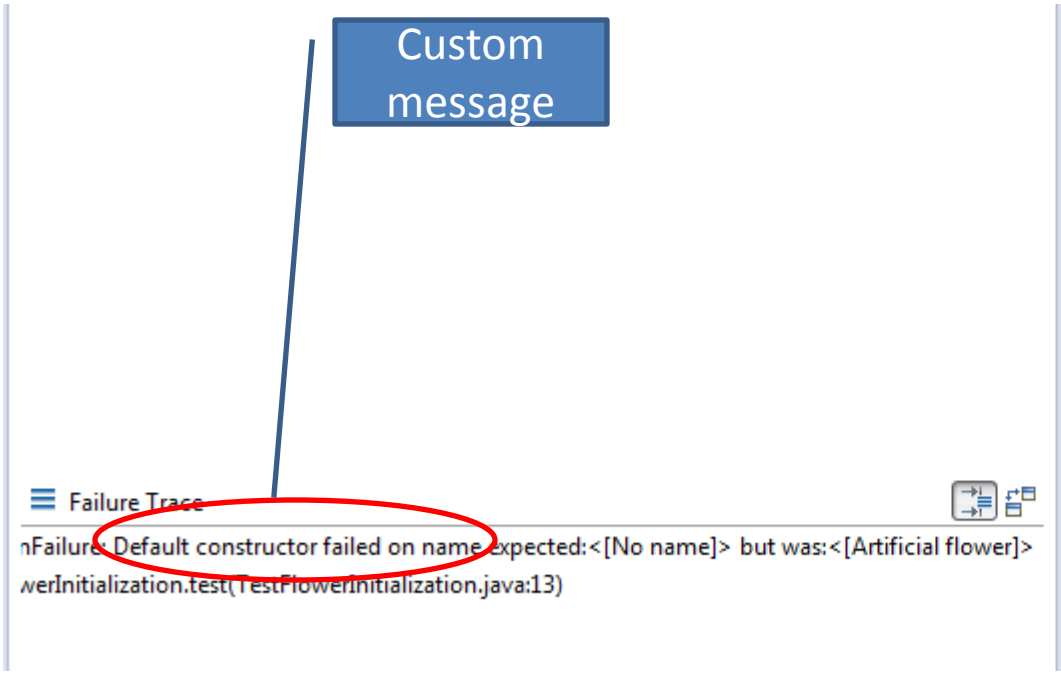
Home quiz 2.2: fix the code to pass JUnit tests

```
package examples;

public class Flower {
        int _petalCount = 0;
        String _name = "No name";

        Flower(int petalCount) {
                _petalCount = petalCount;
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }

        Flower(String name) {
                this();
                _name = name;
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }

        Flower(String name, int petalCount) {
                this(petalCount);
                //! this(name); // Can't call two!
                this._name = name; // Another use of "this"
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }

        Flower() {
                this("Artificial flower", 2);
                System.out.println("Created flower "+ _name+" with "+_petalCount+" petals");
        }
}
```

JUnit Code

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestFlowerInitialization {
    @Test
    public void test() {
        Flower flower=new Flower(); //test default constructor
        assertEquals("Default constructor failed on name", "No name", flower._name);
        assertEquals("Default constructor failed on petals", 0,  flower._petalCount);

        flower=new Flower(2); //test constructor with petals only
        assertEquals("Constructor(int) failed on name","No name", flower._name);
        assertEquals("Constructor(int) failed on petals",2, flower._petalCount);

        flower=new Flower("Rosa"); //test constructor with name only
        assertEquals("Constructor(String) failed on name","Rosa", flower._name);
        assertEquals("Constructor(String) failed on petals",0, flower._petalCount);

        flower=new Flower("Rosa",5); //test constructor with name and petals
        assertEquals("Constructor(String,int) failed on name","Rosa", flower._name);
        assertEquals("Constructor(String,int) failed on petals",5, flower._petalCount);
    }
}
```

After running tests:




```
18          assertEquals("Constructor(int) fai
19          assertEquals("Constructor(int) fai
20
21          //test constructor with name only
22          flower=new Flower("Rosa");
23          assertEquals("Constructor(String)
24          assertEquals("Constructor(String)
25
26          //test constructor with name and p
27          flower=new Flower("Rosa",5);
28          assertEquals("Constructor(String,i
29          assertEquals("Constructor(String,i
30      }
31
32  }
33
```

Assertion Methods

Method	Description
assertEquals(a,b)	Test if a is equal to b
assertFalse(a)	Test if a is false
assertNotSame(a, b)	Test if a and b do not refer to the identical object
assertNull(a)	Test if a is null
assertSame(a,b)	Test if a and b refer to the identical object
assertTrue(a)	Test if a is true
assertEquals(a,b)	Test if a and b are equal

- Static methods defined in junit.framework.Assert
- Variations taking string error messages

Write JUnit Test Case for class

```
public class Fields {
	private String name;
	private String nickname;
	private boolean stateOK;

	public void setName(String name){
		this.name=name;
	}
	public String getName(){
		return this.name;
	}
	public void setNickName(String nickname){
		this.nickname=nickname;
	}
	public String getNickName(){
		return this.nickname;
	}
	public void setState(boolean value){
		this.stateOK=value;
	}
	public boolean getState(){
		return this.stateOK;
	}
}
```

What to test

1. Create new ***Fields*** object
2. //test initial fields values

nickName is Null, name is Null, state is false

3. //set values for String fields and test on null

4. //set the same name and nickname and test on equals

5. //set state to different boolean values and test using assertTrue

Submit listings:

- Corrected Flower class
- Test case class

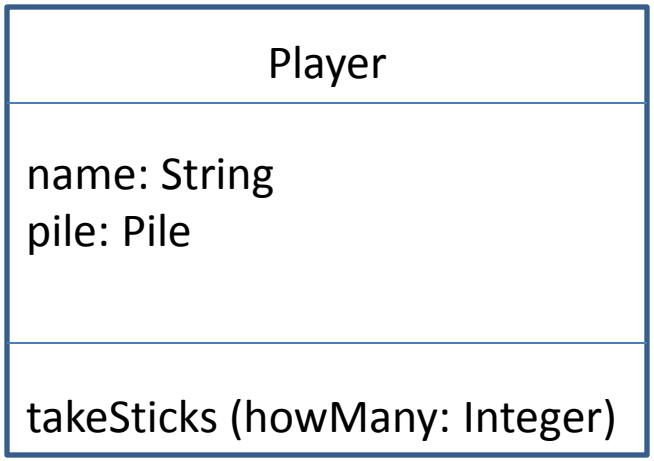
Designing interacting classes: Game 1

Variation of a nim game: subtraction game:

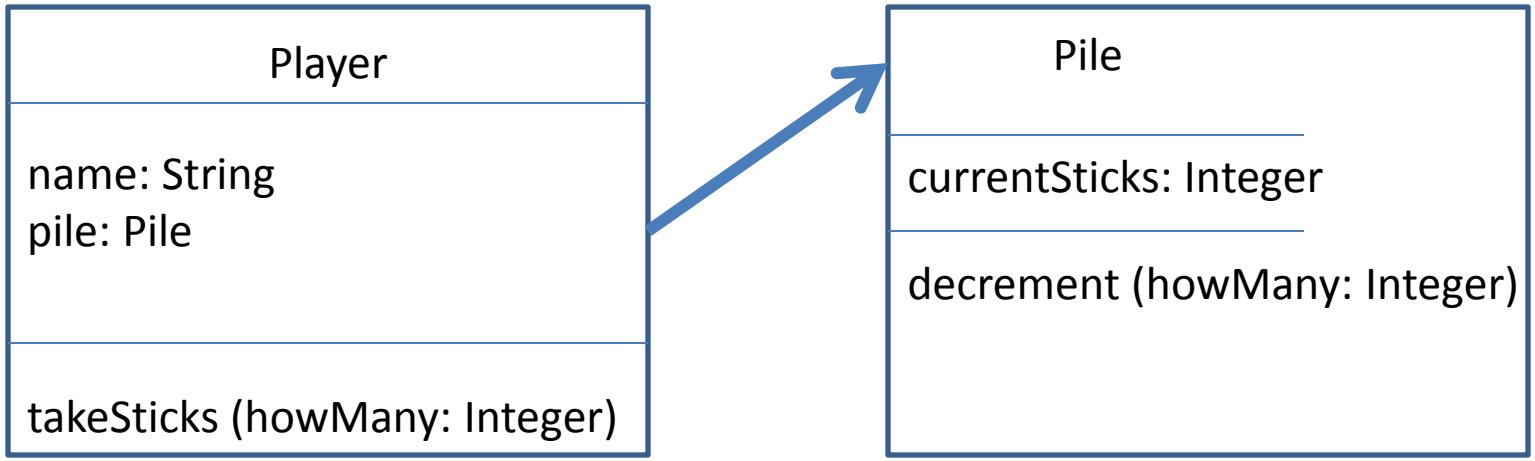
- http://en.wikipedia.org/wiki/Nim 4.1

In this game, players take turns removing sticks from a pile. Each player can remove 1,2, or 3 sticks. The player who removes the last stick wins.

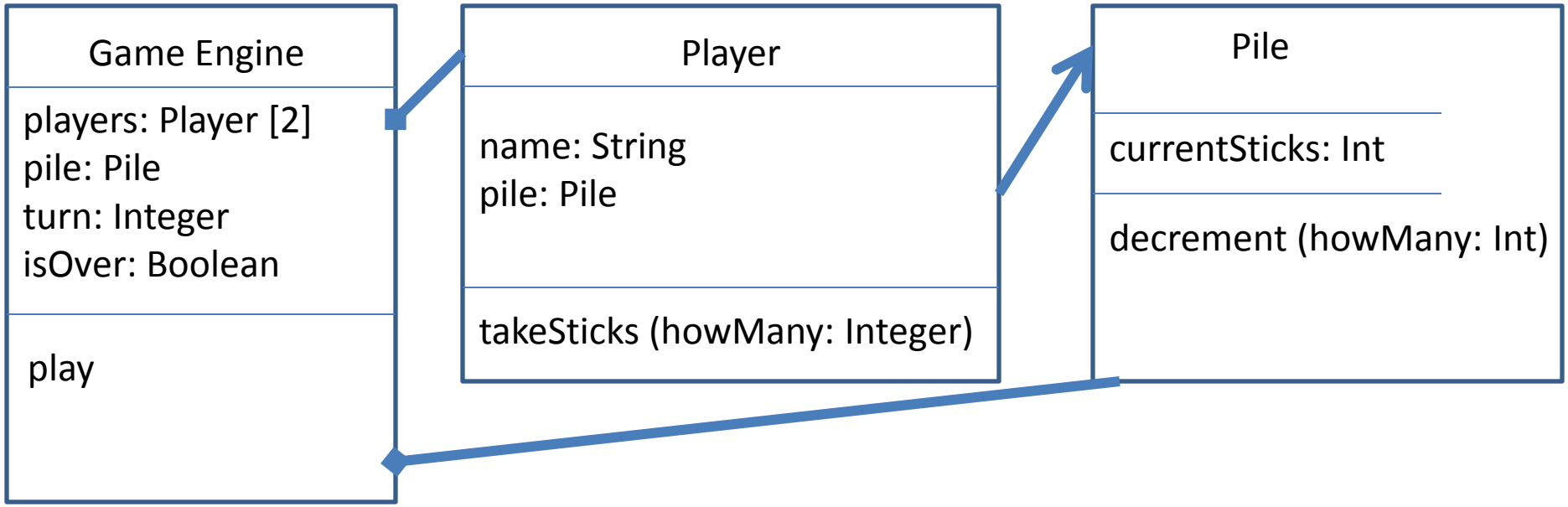
Modeling Player



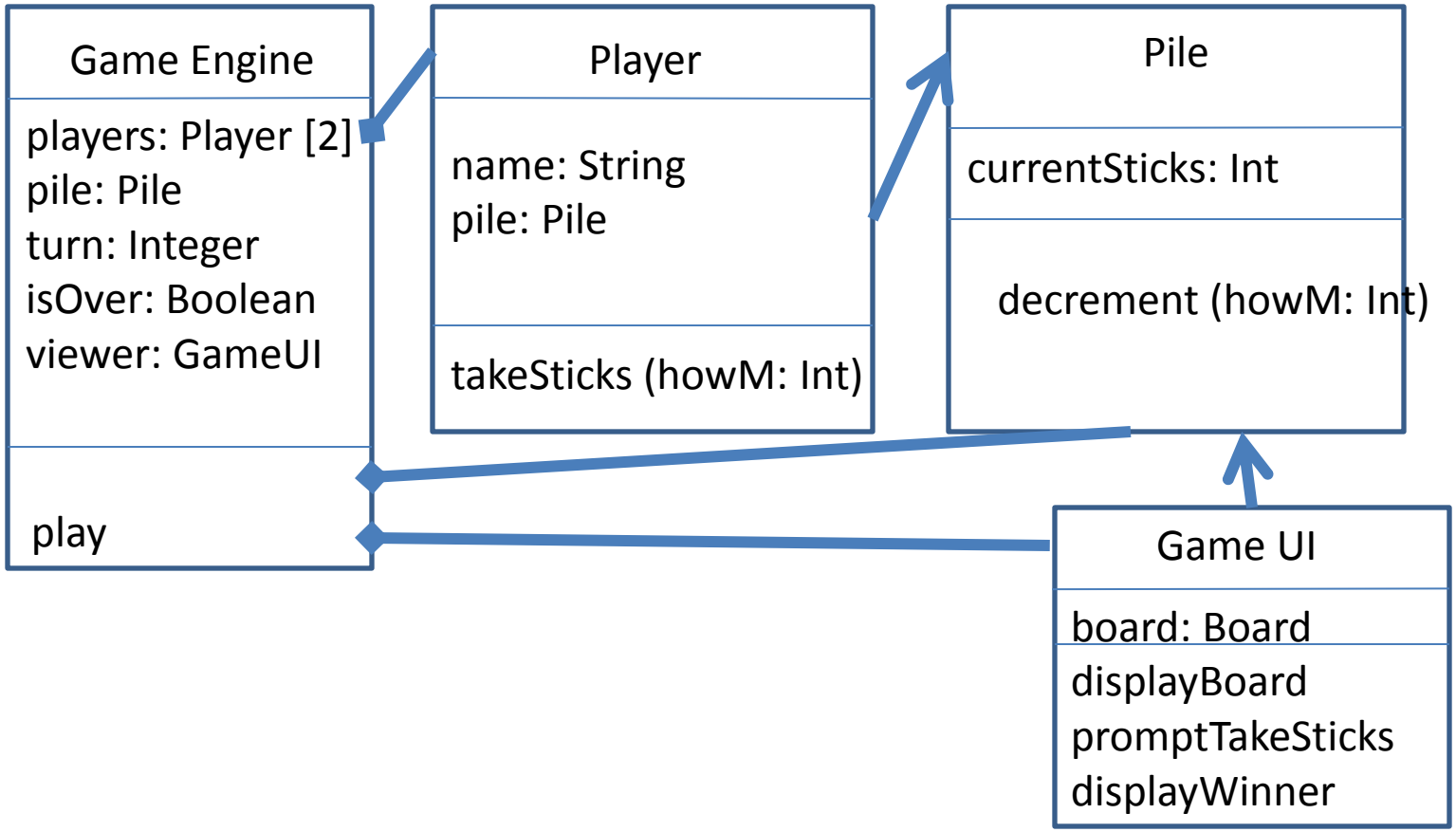
Modeling Pile



Modeling client – a consumer of Player and Pile



Modeling viewer



Implementing Player

```
public class Player
{
        String _name;   //package access variables
        Pile _pile;

        public Player (String name,  Pile pile)  //association with Pile
        {
                _name=name;
                _pile=pile;
        }

        public void takeSticks (int howMany)
        {
                _pile.decrement(howMany);
        }
}
```

Implementing Pile

```
public class Pile
{
        private int _currentSticks;
        public Pile (int initialsticks)
        {
                _currentSticks=initialsticks;
        }

        public void decrement (int howMany)
        {
                _currentSticks-=howMany;
        }

        public int getCurrentSticks()
        {
                return _currentSticks;
        }
}
```

Implementing Client: Constructor

```
public class Game
{
        Player [] _players;
        int _turnIndex;
        GameUI _ui;
        boolean _isOver;
        Pile _pile;

        public Game (String player1Name, String player2Name, int initialsticks)
        {
                _players=new Player[2];
                _pile=new Pile(initialsticks);
                _players[0]=new Player(player1Name,_pile);
                _players[1]=new Player(player2Name,_pile);
                _ui=new GameUI(_pile);
                _isOver=false;
                _turnIndex=0;
        }

}
```

Implementing Client: game logic

```
public class Game {
        public void play()
        {
                while(!_isOver)
                {
                        _ui.displayBoard();
                        int howMany=_ui.promptTakeSticks(_players[_turnIndex]);
                        _players[_turnIndex].takeSticks(howMany);
                        if(_pile.getCurrentSticks()==0)
                        {
                                _isOver=true;
                                _ui.displayWinner(_players[_turnIndex]);
                        }
                        else
                                toogleTurns();
                }
        }
}
```

Implementing client: main

```
public static void main (String [] args){
        if(args.length<3)
        {
                System.out.println("To run subtractiongame.Game <Player1 name> <Player1
                                                                name> <initial pile size>");
                System.exit(0);
        }
        try
        {
                int initialPileSize=Integer.parseInt(args[2]);
                Game game=new Game(args[0],args[1],initialPileSize);
                game.play();
        }
        catch (NumberFormatException nfe)
        {
                System.out.println("Invalid argument 3");
                System.exit(0);
        }
}
```

Implementing Viewer

```
public void displayBoard()
{
        System.out.println();
        System.out.println("**********Subtraction game board**********");
        for(int i=0;i<_pile.getCurrentSticks();i++)
                System.out.print("|");
        System.out.print(System.getProperty("line.separator"));
}
```

Implementing Viewer- Prompt

```
public int promptTakeSticks (Player currentPlayer)
{
        int howMany=-1;  //not defined, must initialize
        int maxSticksAllowed=_pile.getCurrentSticks();

        System.out.println("Turn Player "+currentPlayer._name );
        System.out.println("How many sticks do you want to remove - enter a number
                between 1 and "+Math.min(maxSticksAllowed,3));
**
        return howMany;
}
```

Implementing Viewer- Prompt (Cont.)

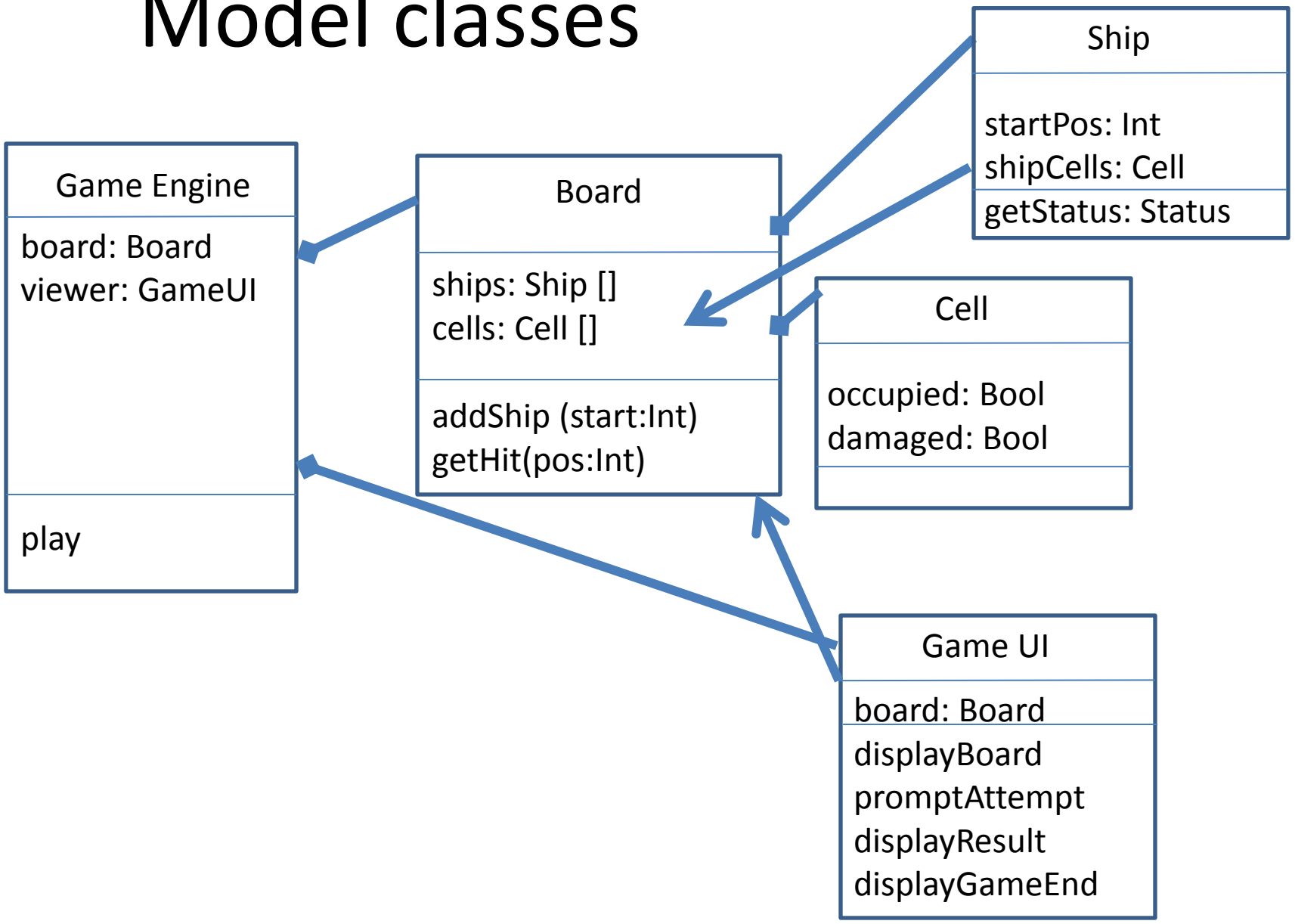
```
public int promptTakeSticks (Player currentPlayer){
        **try{
                BufferedReader bufferRead = new BufferedReader(new InputStreamReader(System.in));
                String s = bufferRead.readLine();
                howMany=Integer.parseInt(s);
                if(howMany<1 || howMany>3 || howMany>maxSticksAllowed){                    {
                        System.out.println("You entered invalid number.
                                Please review the rules of the game and try again." );
                        System.exit(0);
                }
        }
        catch (IOException ioe){
                System.out.println("Unexpected error: "+ioe.getMessage() );System.exit(0);
        }
        catch (NumberFormatException nfe){
        System.out.println("You entered invalid number. Please review the rules of the game
                                                                and try again." );
                System.exit(0);
        }
}
```

Designing interacting classes: Battleship Game - Mini version

- Before play begins, each player (in our case – the computer) secretly arranges their ships on their primary grid.
- Each ship occupies a number of consecutive squares on the grid, arranged either horizontally or vertically. The number of squares for each ship is determined by the type of the ship.
- The ships cannot overlap (i.e., only one ship can occupy any given square in the grid), and they cannot touch each other.

This is a 1-player 1-dimensional version, the length of each ship is 3 cells (submarine)

Model classes



Syntax: Java constants

Constants

public static final int SHIP_LENGTH=3;

Can be accessed by:

Ship. SHIP_LENGTH

Syntax: Java Enums - *Status*

```
public enum Status
{
        OK, DAMAGED, SUNK;  //; is required here.

 @Override public String toString() {
   //only capitalize the first letter
   String s = super.toString();
   return s.substring(0, 1) + s.substring(1).toLowerCase();
 }
}
```

Syntax: Java Enums - *Result*

```
public enum Result {
        MISS, HIT, KILL;

 @Override public String toString() {
   //only capitalize the first letter
   String s = super.toString();
   return s.substring(0, 1) + s.substring(1).toLowerCase();
 }
}
```

Syntax: random integer

```
import java.util.Random;

Random rand = new Random();
int nextShipStart=rand.nextInt(boardSize);  //from 0 to boardSize-1
```

Setting reference to elements of an existing array

```
Cell [] shipCells=new Cell[Ship.SHIP_LENGTH];
shipCells[0]=_cells[startPos];  //reference
shipCells[1]=_cells[startPos+1];
shipCells[0]=_cells[startPos+2];
```

Board: Overriding toString() method

```
public String toString()
{
        String ret="";
        for(int i=0;i<_cells.length;i++)
        {
                ret+=i+"\t";
        }
        ret+=System.getProperty("line.separator");

        for(int i=0;i<_cells.length;i++)
        {
                if(_cells[i].isAttempted())
                        ret+="v"+"\t";
                else if(_cells[i].isDamaged())
                        ret+="X"+"\t";
                else
                        ret+="_"+"\t";
        }

        return ret;
}
```

Now you can use:
System.out.println(_board);

Programming assignment 2

Extend a simple Battleship game to a real Battleship game with 1 player and 2D grid.