(Lab 7) Problem Set 5:
Containers: Objects & Classes

P3 Solutions limited in scope to:		
● P1 Concepts ● P2 Concepts ● P3 Concepts ● P4 Concepts	● Designing Objects ○ instance variables ○ instance methods ○ constructors ○ UML diagrams	● Using Objects ○ Instantiation ○ Storing objects into variables ○ invoking instance methods ○ using Java API

Submission Rules:

1. Submissions must be zipped into a **handin.zip** file. Each problem must be implemented in its own class file. Use the name of the problem as the class name.

2. You must use standard input and standard output for ALL your problems. It means that the input should be entered from the keyboard while the output will be displayed on the screen.

3. Your source code files should include a comment at the beginning including your name and that problem number/name.

4. The output of your solutions must be formatted exactly as the sample output to receive full credit for that submission.

5. Compile & test your solutions before submitting.

6. Each problem is worth up to 10 points total. The breakdown is as follows: 2 points for compiling, 3 points for correct output with sample inputs, 5 points for additional inputs.

7. This lab is worth a max total of: 40 points.

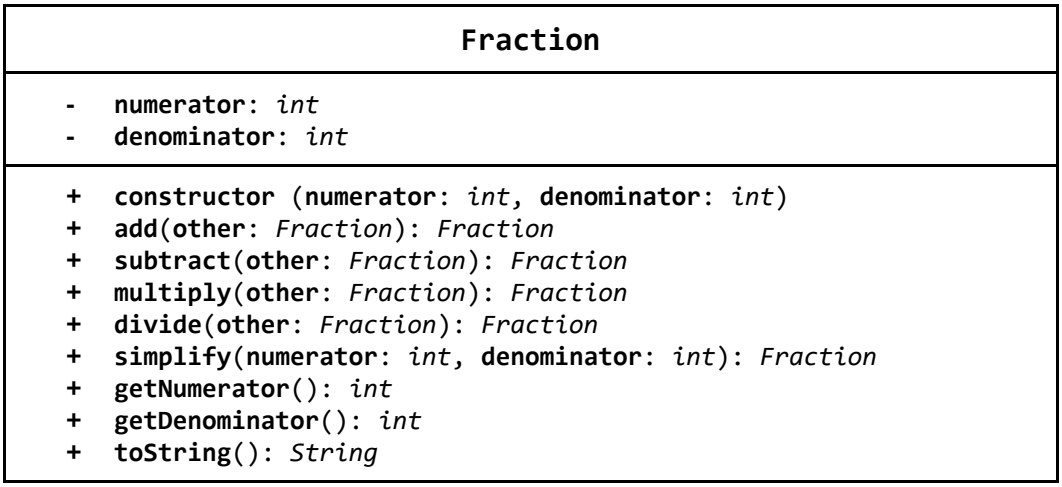
8. Submission:

- You have unlimited submission attempts until the deadline passes
- You'll receive your lab grade immediately after submitting

Problem 1: Fraction (10 points)

(Mathematics) Create a new class that represents fractional numbers as a pair of integers: numerator and denominator. Fractional numbers should support the basic arithmetic operations: multiply, divide, add, subtract. Fractional numbers should also simplify their integer representations as the smallest possible values.

UML Class Diagram:



Fraction Constructor Summary:

Constructor	Description
`Fraction(int numerator, int denominator)`	Creates a fraction object from two integer values. The numerator should carry the +/- sign and the denominator should be positive

Fraction Method API:

Modifier and Type	Method and Description
`Fraction`	**`add`**`(Fraction other)` Returns new fraction object based on this fraction added to other fraction
`Fraction`	**`divide`**`(Fraction other)` Returns new fraction object based on this fraction divided by other fraction
`int`	**`getDenominator`**`()` Returns the denominator
`int`	**`getNumerator`**`()` Returns the numerator
`Fraction`	**`multiply`**`(Fraction other)` Returns new fraction object based on this fraction multiplied by other fraction
`Fraction`	**`simplify`**`(int numerator, int denominator)` Returns a fraction of simplified form using the given integer arguments
`Fraction`	**`subtract`**`(Fraction other)` Returns new fraction object based on this fraction subtracted from other fraction
`String`	**`toString`**`()` Returns a text representation of fraction in form of: "(numerator/denominator)"

Facts

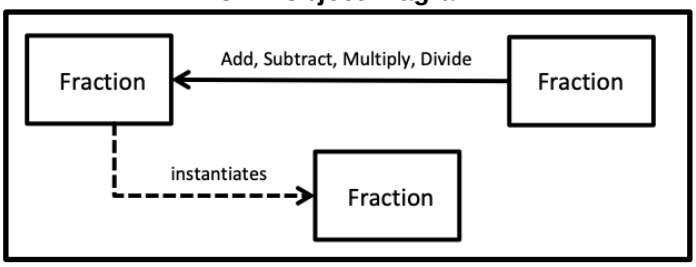
- **`Fraction`** should contain arithmetic methods: **`add,subtract,multiply,divide`**
  - **`add:`**
    - `numerator = numerator1 * denominator2 + numerator2 * denominator1`
    - `denominator = denominator1 * denominator2`
    - `Then simplify`
  - **`subtract:`**
    - `numerator = numerator1 * denominator2 - numerator2 * denominator1`
    - `denominator = denominator1 * denominator2`
    - `Then simplify`
  - **`multiply:`**
    - `numerator = numerator1 * numerator2`
    - `denominator = denominator1 * denominator2`
    - `Then simplify`
  - **`divide:`**
    - `numerator = numerator1 * denominator2`
    - `denominator = denominator1 * numerator2`
    - `Then simplify`

- **`Simplify`** should iterate on the value of the **`denominator`** to check if that value is evenly divisible for both the **`denominator`** and **`numerator`**, if so then divide them both by that value. Continue this process until you decrement down to 1.

Software Architecture:

The **`Fraction`** class is designed to be instantiated by an external application file to generate new Fraction objects.

UML Object Diagram



Tester Files:

Use the *`FractionTester.java`* file to test your implementation. Compare your results with the *`fractions.txt`* file.

Sample Method Calls	Sample Method Results

`Fraction f1 = new Fraction(1,2);`
`Fraction f2 = new Fraction(1,3);`
`Fraction f3 = f1.add(f2);`
`System.out.printf("%s+%s=%s")`
	`(1/2)+(1/3)=(5/6)`

Problem 2: Dungeon Crawl (10 points)

(Game Dev) Build a Room class that can be used to construct dungeons that allows a player to navigate a text-based world and explore its environments. Use the Unified Modeling Language (UML) diagram below for constructing your class file.

UML Class Diagram:

Room
- **name**: *String* - **description**: *String* - **north**: *Room* - **east**: *Room* - **west**: *Room* - **south**: *Room*
+ **constructor** (**name**: *String*, **description**: *String*) + **setExits**(**n**:*Room*, **e**:*Room*, **w**:*Room*, **s**:*Room*) + **getExits**(): *String* + **getName**(): *String* + **getNorth**(): *Room* + **getEast**(): *Room* + **getWest**(): *Room* + **getSouth**(): *Room* + **toString**(): *String*

Room Constructor Summary:

Constructor	Description
`Room(String name, String description)`	Creates a Room object from two Strings.

Room Method API:

Modifier and Type	Method and Description
`Room`	**`getEast`**`()` Returns the room east of this one, if no such room exists, then null is returned.
`String`	**`getExits`**`()` Returns a String of this room's exits with both the exit direction & the room's name
`String`	**`getName`**`()` Returns a String with this room's name.
`Room`	**`getNorth`**`()` Returns the room north of this one, if no such room exists, then null is returned
`Room`	**`getSouth`**`()` Returns the room south of this one, if no such room exists, then null is returned
`Room`	**`getWest`**`()` Returns the room west of this one, if no such room exists, then null is returned
`void`	**`setExits`**`(Room n, Room e, Room w, Room s)` Sets all four exits for this room: north, east, west, south
`String`	**`toString`**`()` Returns a text representation of this room with name, description, and exits

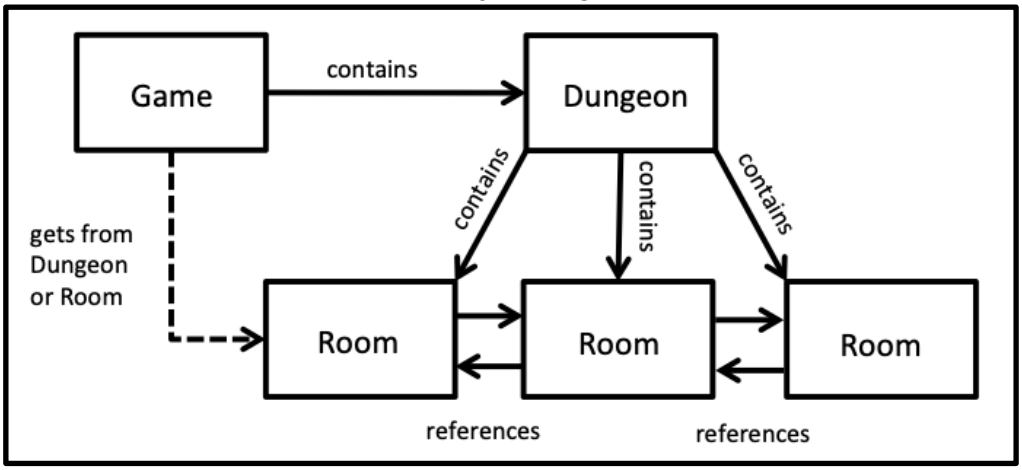
Facts

- Each **`Room`** object contains references to up to four other **`Room`** objects
- **`toString()`** should include the rooms's name, room's description, and the method getExits()

Software Architecture:

The **`Room`** class is designed to be instantiated by **`Dungeon`** class which is used by the **`Game`** class. See the UML object diagram below:

UML Object Diagram



Tester Files:

Use the *`RoomTester.java`* file to test your implementation. Use the *`Game.java`* app to play a simple version of a text-based exploration game using your **`Room`** class.

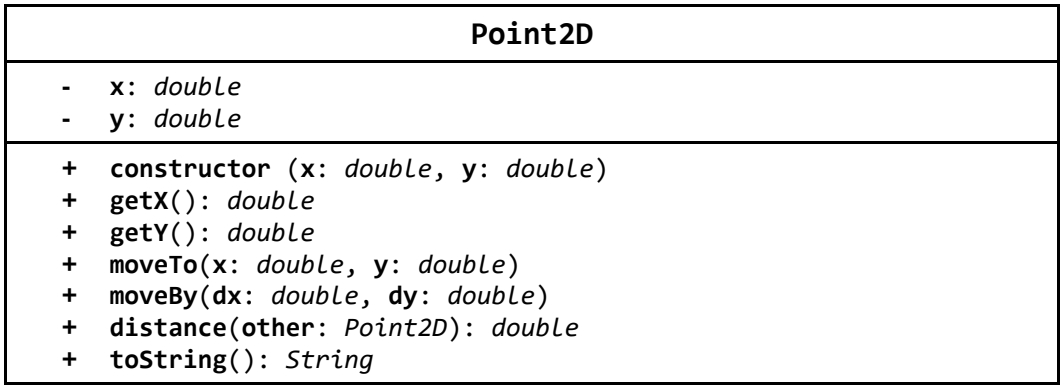
Sample Method Calls	Sample Method Results

`Room hall = new Room("Hall", "Its Dark.");`
`Room bed = new Room("Bed", "Tiny room.");`
`Room bath = new Room("Bath", "Toilets here.");`
`Room dine = new Room("Dining", "Table & chairs");`
`hall.setExits(bed, bath, dine, null);`
`System.out.println(hall)`
	`[Hall]`
	`Its Dark.`
	`[N]orth: Bed`
	`[E]ast:  Bath`
	`[W]est:  Dining`

Problem 3: Point2D (10 points)

(Data Analytics) A core modeling tool used by data scientists is that of 2d points. Points are most common in generating graphs and are used to plot data across two axis, where given some observation x we can map to a result y. But as common as points might be, they aren't a Java primitive data type. Build a **`Point2D`** class that represents a geometric two dimensional point with an x-coordinate and a y-coordinate. You should be able to move and calculate its distance from another point.

UML Class Diagram:



Point2D Constructor Summary:

Constructor	Description
`Point2D(double x, double y)`	Creates a Point2D object from two double values.

Point2D Method API:

Modifier and Type	Method and Description
`double`	**`distance`**`(Point2D other)` Returns the distance between between two points, uses the distance formula.
`double`	**`getX`**`()` Returns this point's x-coordinate.
`double`	**`getY`**`()` Returns this point's y-coordinate.
`void`	**`moveBy`**`(double dx, double dy)` moves point by dx units on x-axis & dy units on y-axis
`void`	**`moveTo`**`(double x, double y)` moves point to new position with given x, y values
`String`	**`toString`**`()` Returns a text representation of this point formatted as: (x,y) Decimal precision should be limited to the tenths place.

Facts

- distance formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)}^2$

Software Architecture:

The **`Point2D`** class is designed to be instantiated by external classes and interact with other point objects. See the UML object diagram below:

Tester Files:

Use the *`Point2DTester.java`* file to test your implementation.

Sample Method Calls	Sample Method Results

`Point2D p1 = new Point2D(1,1);`
`System.out.printf("point1: %s\n",p1);`
`Point2D p2 = new Point2D(4,3);`
`System.out.printf("point2: %s\n",p2);`
`double distance = p1.distance(p2);`
`System.out.printf("distance: %f\n",distance);`
	`point1: (1.0,1.0)`
	`point2: (4.0,3.0)`
	`distance: 3.605551`

Problem 4: ATM Banking (10 points)

(Software Engineering) Digital account management systems in the dominant approach that most companies provide services to their clients. You use such account based login systems for social media, streaming entertainment services, shopping, banking, etc. Each of these systems must manage multiple user accounts. You must create a **`Account`** class for use in an ATM system with the given specification .

UML Class Diagram:

`Account`
- **count**: *int* (static) - **name**: *String* - **pin**: *String* - **id**: *int* - **balance**: *double*
+ **constructor** (**name**: *String*, **pin**: *String*) + **getName**(): *String* + **getID**(): *int* + **getBalance**(): *double* + **isPin**(**attempt**: *String*): *boolean* + **deposit**(**amount**: *double*) + **withdraw**(**amount**: *double*) + **toString**(): *String*

Account Constructor Summary:

Constructor	Description
`Account(String name, String pin)`	Creates Account object with given name and pin. Set balance to 0 & set id to count, then increment count.

Account Method API:

Modifier and Type	Method and Description
`void`	**`deposit`**`(double amount)` Adds amount to balance provided it is a positive value.
`double`	**`getBalance`**`()` Returns the dollar balance of this account
`int`	**`getID`**`()` Returns this account's id number
`String`	**`getName`**`()` Returns the person's name associated with this account
`boolean`	**`isPin`**`(String attempt)` Returns true if attempt matches pin otherwise false
`double`	**`withdraw`**`(double amount)` Subtracts amount from balance provided it is positive value and less than balance
`String`	**`toString`**`()` Returns a text representation of this account, formatted as: "Name: %s, Account ID: %d, Balance: $%.02f"

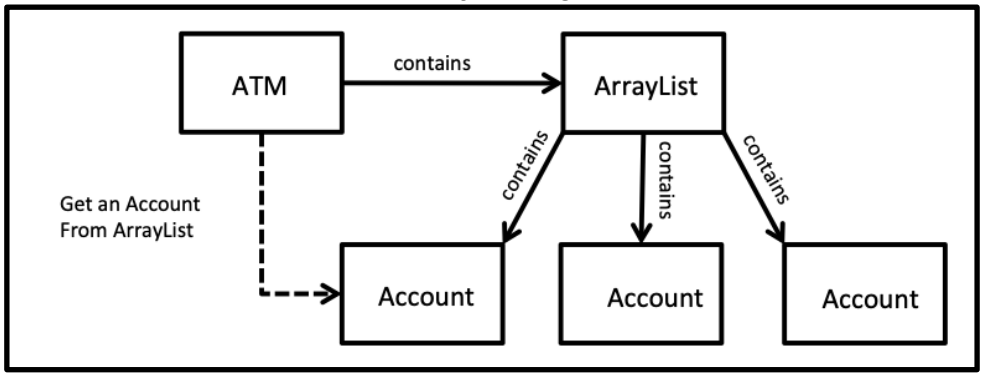
Facts

- Make sure to check that the inputs for **`deposit`** and **`withdraw`** are valid.
- Consider using **`String.format()`** in your **`toString()`**

Software Architecture:

The **`Account`** class is designed to be instantiated by **`ATM`** class which is used to make account objects. See the UML object diagram below:

UML Object Diagram



Tester Files:

Use the *`AccountTester.java`* file to test your implementation. Use the *`ATM.java`* app to play a simple simulation of an ATM banking system using your **`ATM`** class.

Sample Method Calls	Sample Method Results

`Account tim = new Account("Tim", "1234");`
`System.out.println(tim);`
`Account ted = new Account("Ted", "9999");`
`System.out.println(ted);`
	`Name: Tim, Account ID: 0, Balance: $0.00`
	`Name: Ted, Account ID: 1, Balance: $0.00`

Problem 5: Monster Factory (10 points)

(Game Dev) Create a Monster class that maintains a count of all monsters instantiated and includes a static method that generates a new random monster object. In software engineering, a method that generates new instances of classes based on configuration information is called the Factory pattern.

UML Class Diagram:

Monster
- **name**: *String* - **health**: *int* - **strength**: *int* - **xp**: *int*
+ **spawn(type:***String***): *Monster*** + **constructor** (**name**: *String*, **health**: *int*, **strength**: *int*, **xp**: *int*) + **getName**(): *String* + **getHealth**(): *int* + **getStrength**(): *int* + **getXP**(): *int* + **takeDamage**(**damage**: *int*) + **attack**(**hero**: *Player*) + **toString**(): *String*

Monster Constructor Summary:

Constructor	Description
Monster(String name, int health, int strength, int xp)	Creates Monster object with given name, health, strength. and xp.

Monster Method API:

Modifier and Type	Method and Description
void	**attack**(Player hero) Monster attacks player, where player takes damage equal to monster strength. Display message: "%s attacks player for %d damage", name, strength
int	**getHealth**() Returns this monster's health
String	**getName**() Returns this monster's name
int	**getStrength**() Returns this monster's strength
int	**getXP**() Returns this monster's experience
static Monster	**spawn(**String type**)** Returns a monster object of given type for "goblin", "orc", or "troll"
void	**takeDamage**(int damage) health is decreased by given damage, but can't be a negative
String	**toString**() Returns a text representation of this account, formatted as: "[%s] HP: %d, STR: %d", name, health, strength

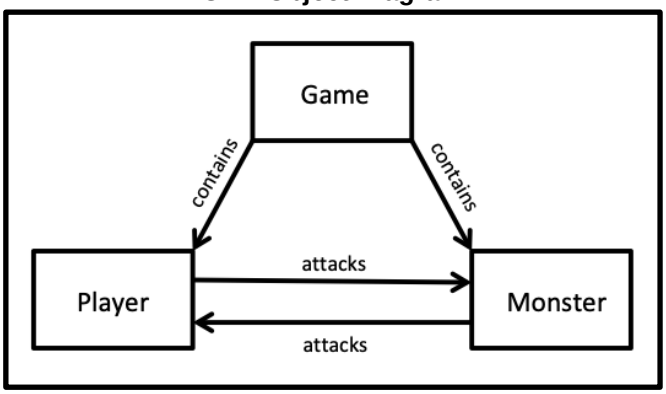
Facts

- Monster types that may be spawned with following attribute values:
    - `goblin, name="goblin", health=60, strength=8, xp=1`
    - `orc, name="orc", health=100, strength=12, xp=3`
    - `troll, name="troll", health=150, strength=15, xp=5`
- `monster's` **`attack()`** should invoke player's **`takeDamage()`** method.
- Consider using **`String.format()`** in your **`toString()`**

Software Architecture:

The **`Monster`** class is designed to be instantiated by **`Game`** class which is used to make Monster objects and Player objects. See the UML object diagram below:

UML Object Diagram



Tester Files:

Use the *`MonsterTester.java`* file to test your implementation. Use the *`Game.java`* app to play a simple game using your **`Monster`** class.

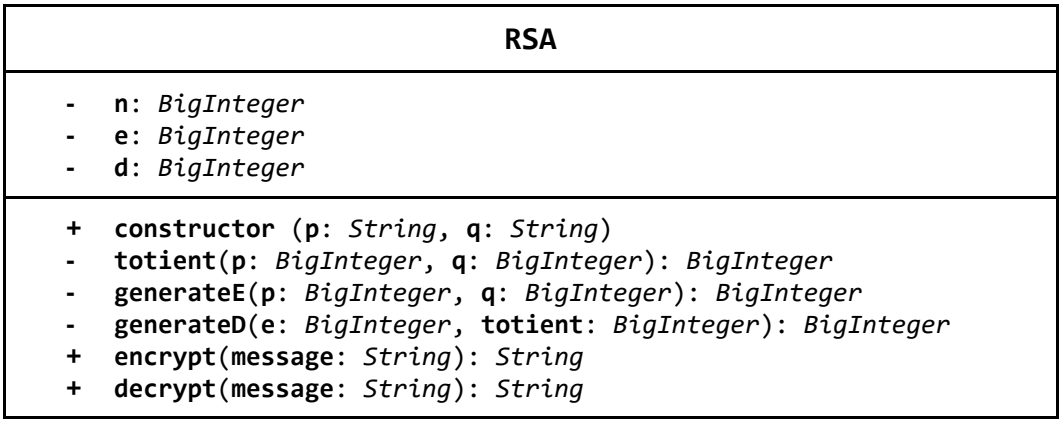
Sample Method Calls	Sample Method Results

`Monster goblin = Monster.spawn("goblin");`
`Monster orc = Monster.spawn("orc");`
`Monster troll = Monster.spawn("troll");`
`System.out.println(goblin);`
`System.out.println(orc);`
`System.out.println(troll);`
	`[goblin] HP:60, STR:8`
	`[orc] HP:100, STR:12`
	`[troll] HP:150, STR:15`

Problem 6: RSA (10 points)

(Cyber Security) Use BigInteger API to implement RSA encryption. RSA is an asymmetric encryption scheme, where a public key is used to encrypt data and a different, private key decrypts that data. RSA public/private keys are generated from two prime numbers, typically very large ones.

UML Class Diagram:



RSA Constructor Summary:

Constructor	Description
`RSA(String p, String q)`	Converts p and q into Big Integers and sets the following attributes: n = p * q e = generate E (p, q) d = generate D (e, totient)
`RSA(String p, String q, String e)`	Converts p,q,e into Big Integers and sets the following attributes: n = p * q, d = generate D (e, totient)

RSA Method API:

Modifier and Type	Method and Description
`BigInteger`	**`totient`**`(BigInteger p, BigInteger q)` Returns the totient which is calculated as: (p-1) * (q - 1), The totient represents the approximate number of primes that occur before the composite n = p * q
`BigInteger`	**`generateE`**`(BigInteger p, BigInteger q)` Returns a randomly-generated value for e, e must be prime and less than totient
`BigInteger`	**`generateD`**`(BigInteger e, BigInteger totient)` Returns a valid value for d, uses the multiplicative inverse to find d such that: (e * d) % totient == 1, Note: use an iterative approach to find a valid value for d
`String`	**`encrypt`**`(String message)` Returns an encrypted message using the following approach: Take each char in message, convert char to integer c, then find encrypted char = c^e % n
`String`	**`decrypt`**`(String message)` Returns a decrypted message using the following approach: Take each char in message, convert char to integer c, then find decrypted char = c^d % n

Facts

- **`BigInteger API:`** https://docs.oracle.com/javase/10/docs/api/java/math/BigInteger.html

- **`BigInteger`** contains arithmetic methods: **`add,subtract,multiply,mod,modPow`**
- **`BigInteger`** contains **`equals`** that checks for equality between two BigInteger objects
- **`BigInteger`** contains **`compareTo`** that compares two different BigInteger objects as follows:
  - BigInteger < BigInteger → -1
  - BigInteger = BigInteger → 0
  - BigInteger > BigInteger → 1
- **`BigInteger`** contains **`nextProbablePrime`** that returns the next prime after the given number
- **`BigInteger`** contains **`intValue`** that converts a BigInteger value into an integer value
- **`BigInteger`** contains prebuilt constant values: **`BigInteger.ONE`**
- **`String`** contains **`toCharArray`** that converts a String into an array of char values

Software Architecture:
The RSA class is designed to be instantiated by an external application file to generate new RSA encryption keys.

Tester Files:
Use the RSATester.java file to test your implementation.

Sample Method Calls	Sample Method Results

`RSA rsa = new RSA("19","79", "17");`
`String message = "Hello World";`
`message = rsa.encrypt(message);`
`message = rsa.decrypt(message);`
	`ˠꞴꞴ˫ʜ˙˫Ꞵʢ`
	`Hello World`