THE UNIVERSITY OF MELBOURNE

# COMP90041

# Programming and Software Development

## Tutorial 6 The Anatomy of Classes & Objects(cont.)

Chuang(Frank) Wang


THE UNIVERSITY OF MELBOURNE

# Overview

1. Immutable & Mutable objects
2. Privacy leak & Copy objects
3. Exercise


THE UNIVERSITY OF MELBOURNE

# 1. Immutable & Mutable objects

# Immutable Objects

```java
public class Player {
    private final String firstName;
    private final String lastName;
    private final int score;

    public Player(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.score = 0;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getScore() {
        return score;
    }
}
```

accessors

```
Player player1 = new Player("Frank", "Wang");
```

## Why `private`?

- the instance variable cannot be accessed by name outside of the class definition
- force the users of those class to use methods to access them.

## Why `final`?

- Cannot be changed once it is initialized

## Why `accessors`?

- Restrict access to read only(not write)
- Adding checks(no plain access)

# Mutable Objects

```
9   public class Player {
10      private String firstName;
11      private String lastName;
12      private int score;
13
14  @   public Player(String firstName, String lastName) {
15          this.firstName = firstName;
16          this.lastName = lastName;
17          this.score = 0;
18      }
19
20      public String getFirstName() {
21          return firstName;
22      }
23
24      public void setFirstName(String firstName) {
25          this.firstName = firstName;
26      }
27
28      public String getLastName() {
29          return lastName;
30      }
31
32      public void setLastName(String lastName) {
33          this.lastName = lastName;
34      }
35  }
```

accessors

mutators

## Mutators

- Change the data in an object
- Add some checks before changing data
- No plain change

# Immutable Objects

```java
public class Player {
    private final String firstName;
    private final String lastName;
    private final int score;

    public Player(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.score = 0;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getScore() {
        return score;
    }
}
```

Which one is legal?

```java
Player player1 = new Player("Frank", "Wang");
```

## Change the object itself

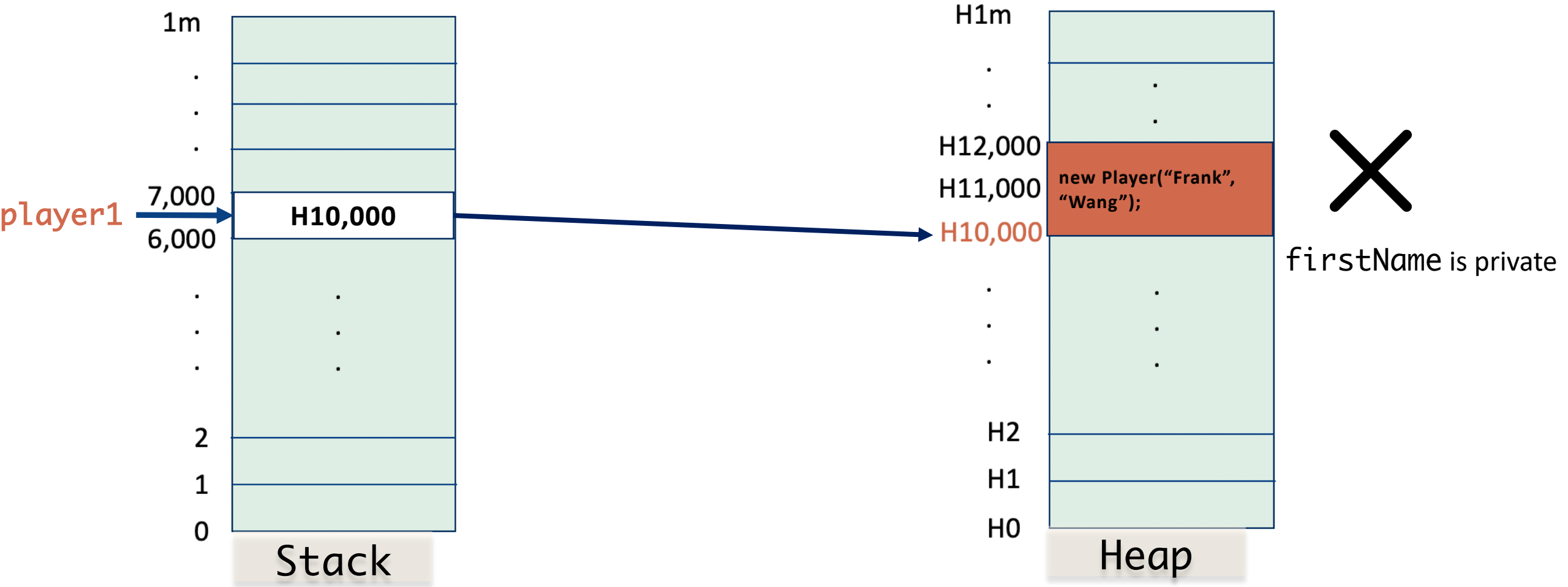
```java
1.      player1.firstName = "Alex";
```

## Dereferencing

```java
2.      player1 = new Player("Alex", "Wang");
```

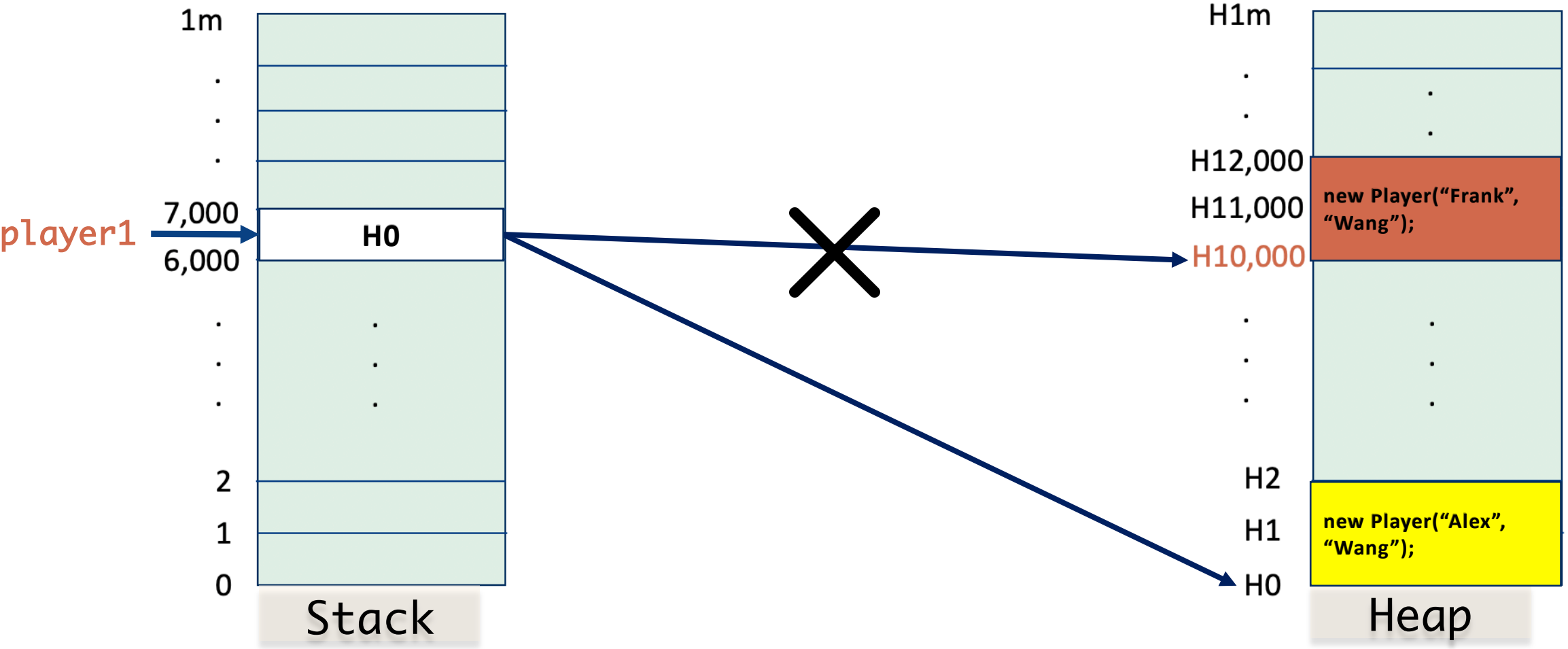
# Change an object

```
Player player1 = new Player("Frank", "Wang");
player1.firstName = "Alex";
```



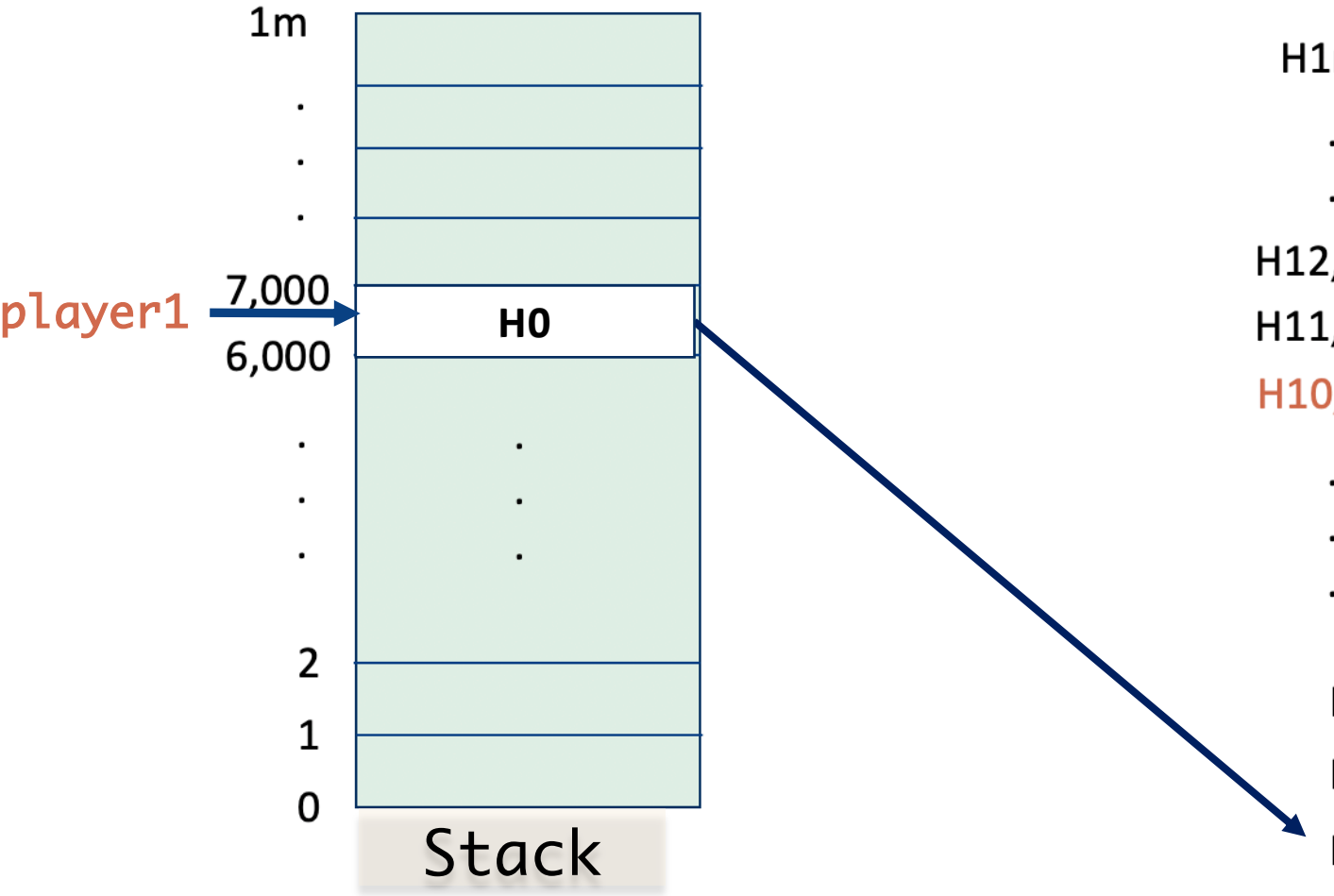
# Dereferencing an object

```
Player player1 = new Player("Frank", "Wang");
player1 = new Player("Alex", "Wang");
```



# Dereferencing(cont.)

```
Player player1 = new Player("Frank", "Wang");
player1 = new Player("Alex", "Wang");
```




**Q:** Where did it go?

**A: Garbage Collector**

# Java Garbage Collector

- ✓ destroying the dead objects from heap memory
- ✓ Reclaim(free up) that memory
- ✓ Return the storage back to OS for future reuse(object allocation)


**Q:** When is an object declared dead?

**A: No reference pointing to the object (Unreachable)**

# Mutable Objects

```java
public class Player {
    private String firstName;
    private String lastName;
    private int score;

    public Player(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.score = 0;
    }

    public String getFirstName() {
        return firstName;
    }

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

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Which one is legal?

```java
Player player1 = new Player("Frank", "Wang");
```

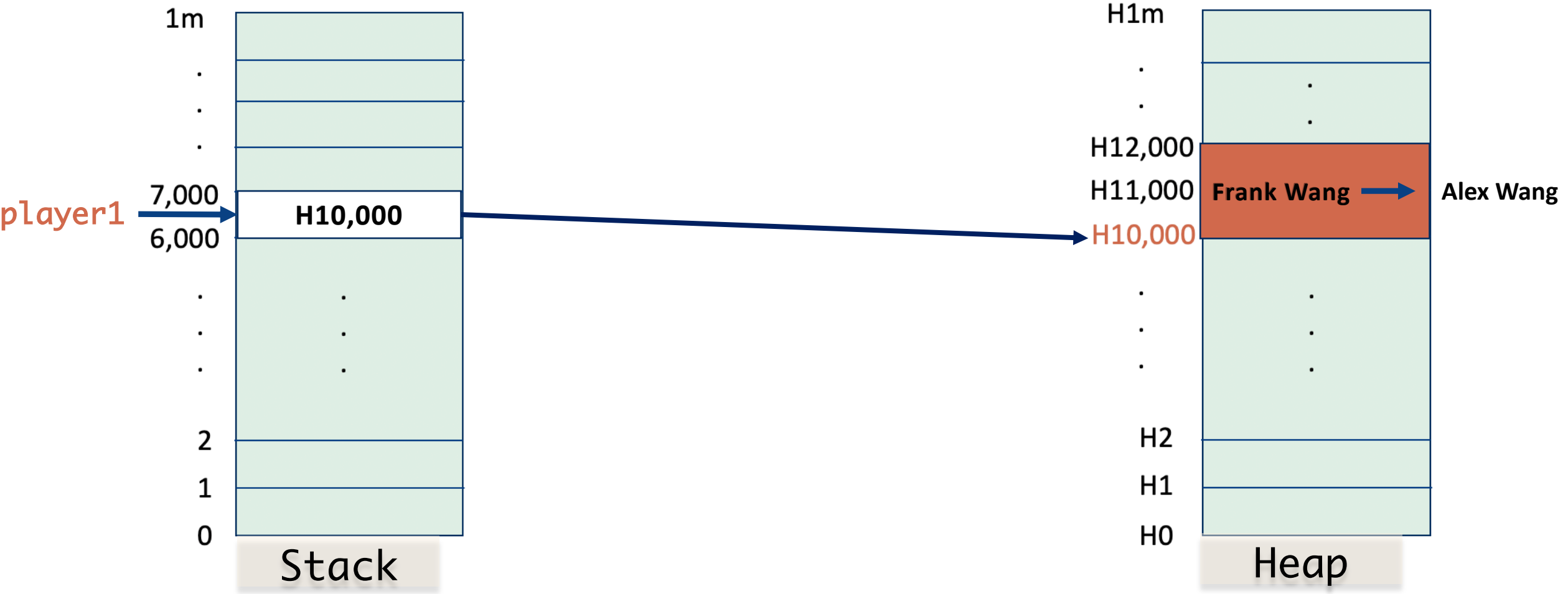
1. `player1.firstName = "Alex";` ✗

2. `player1.setFirstName("Alex");` ✓

3. `player1 = new Player("Alex", "Wang");` ✓

# Change an object

```
Player player1 = new Player("Frank", "Wang");
Player1.setFirstName("Alex");
```




THE UNIVERSITY OF
MELBOURNE

# 2. Privacy leak

# & Copy objects


# Privacy Leak

## Definition

A poorly designed class allows a programmer to circumvent the `private` modifier to access the private instance variable outside that class.

## How to avoid?

Use copy constructor instead of returning the original reference

# Privacy Leak

| Date | |
|---|---|
| month | String |
| day | int |
| year | int |
| setDate(int, int, int) | void |
| setDate(String, int, int) | void |
| setDate(int) | void |
| setYear(int) | void |
| setMonth(int) | void |
| setDay(int) | void |
| getMonth() | int |
| getDay() | int |
| getYear() | int |
| toString() | String |
| equals(Date) | boolean |
| precedes(Date) | boolean |
| readInput() | void |
| dateOK(int, int, int) | boolean |
| dateOK(String, int, int) | boolean |
| monthOK(String) | boolean |
| monthString(int) | String |

| Person | |
|---|---|
| name | String |
| born | Date |
| died | Date |
| set(String, Date, Date) | void |
| toString() | String |
| equals(Person) | boolean |
| datesMatch(Date, Date) | boolean |
| setBirthDate(Date) | void |
| setDeathDate(Date) | void |
| setName(String) | void |
| setBirthYear(int) | void |
| setDeathYear(int) | void |
| getName() | String |
| getBirthDate() | Date |
| getDeathDate() | Date |
| consistent(Date, Date) | boolean |

**Which implementation has privacy leak issue?**

**Option1:**

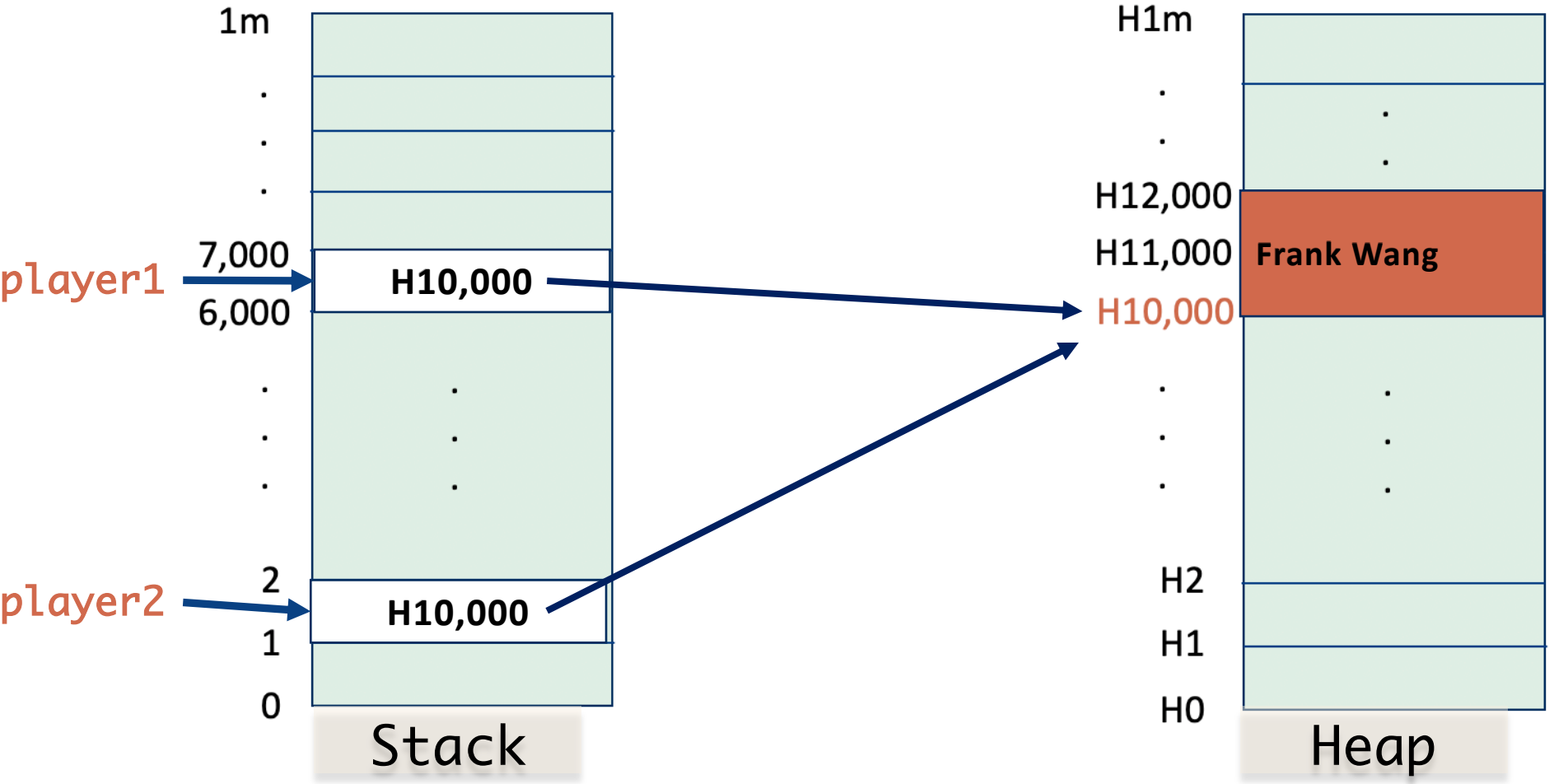
```
public Date getBirthDate( )
{
    return born;
}
```

**Option2:**

```
public Date getBirthDate( )
{
    // deep copy
    // by using copy constructor
    return new Date(born);
}
```

# Copy an object

```
Player player1 = new Player("Frank", "Wang");
Player player2 = player1;
```




**Q:** player1 == player2

**A: True**

**Q:** player1.equals(player2)

**A: True**

# Deep Copy(Copy constructor)

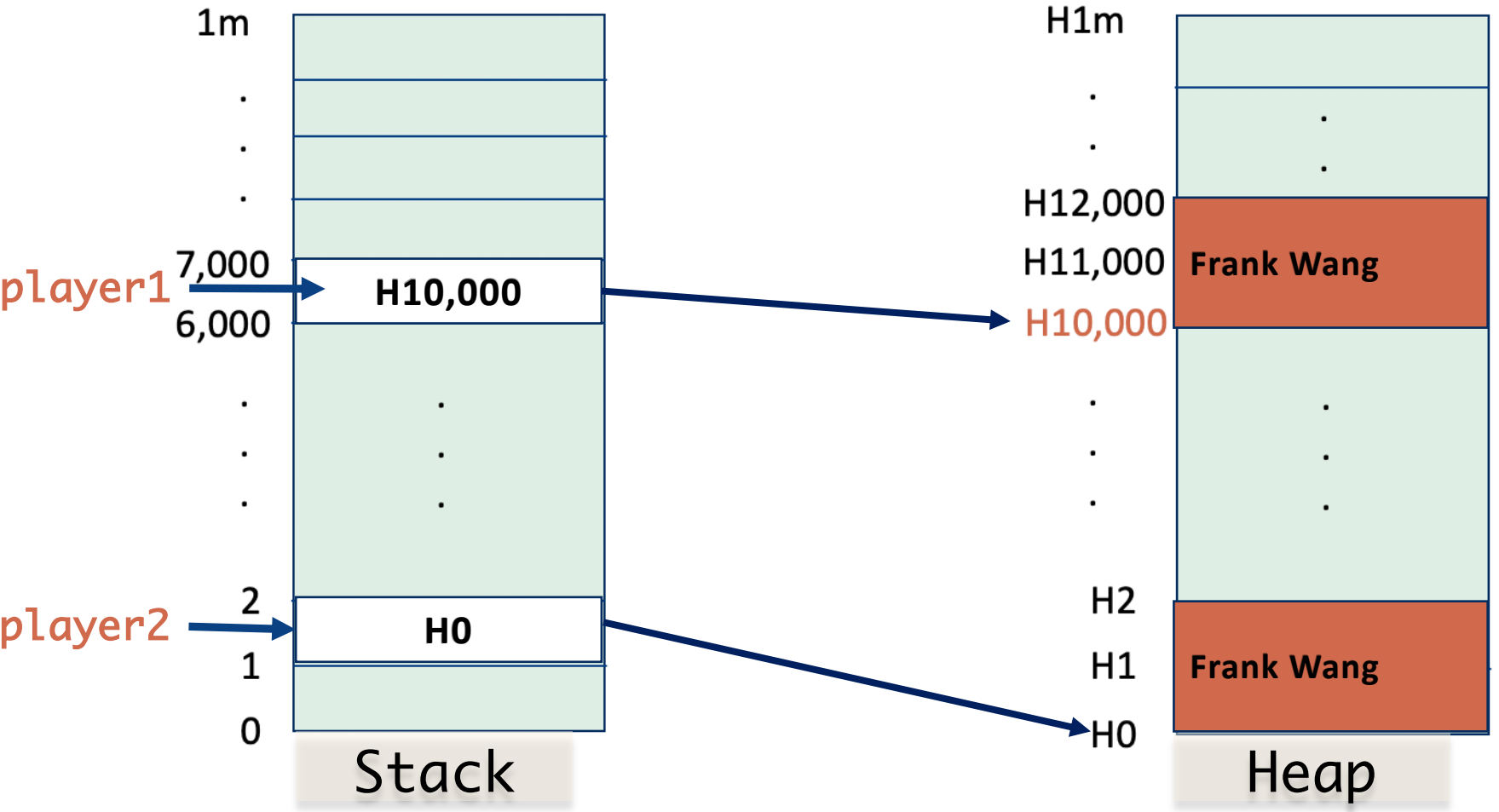
```java
public class Player {
    private String firstName;
    private String lastName;
    private int score;

    public Player(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.score = 0;
    }

    public Player(Player original){
        if (original == null)
        {
            System.out.println("Fatal error.");
            System.exit( status: 0);
        }
        firstName = original.firstName;
        lastName = original.lastName;
        score = original.score;
    }

    public String getFirstName() { return firstName; }

    public String getLastName() { return lastName; }

    public int getScore() { return score; }
}
```

```java
Player player1 = new Player("Frank", "Wang");
```

```java
Player player2 = new Player(player1);
```

# Deep Copy(cont.)

```
Player player1 = new Player("Frank", "Wang");
Player player2 = new Player(player1);
```




**Q:** player1 == player2

**A: False**

**Q:** player1.equals(player2)

**A: True**


THE UNIVERSITY OF
MELBOURNE

# 3. Exercise


# Tutorial Exercise

1. Revise your `Character` class from last lab by adding or modifying and methods so that the class is now mutable. Ensure your class is not subject to privacy leaks.


# Homework

2. Revise your `Movie` class from last week to use your new mutable `Character` class. Minimally adapt your testing code from last week to test your new `Movie` class.

3. Compare the difficulty of avoiding *privacy leaks* when using immutable classes for all instance variables with the difficulty when some instance variables hold mutable objects.

THE UNIVERSITY OF MELBOURNE

# Thank you