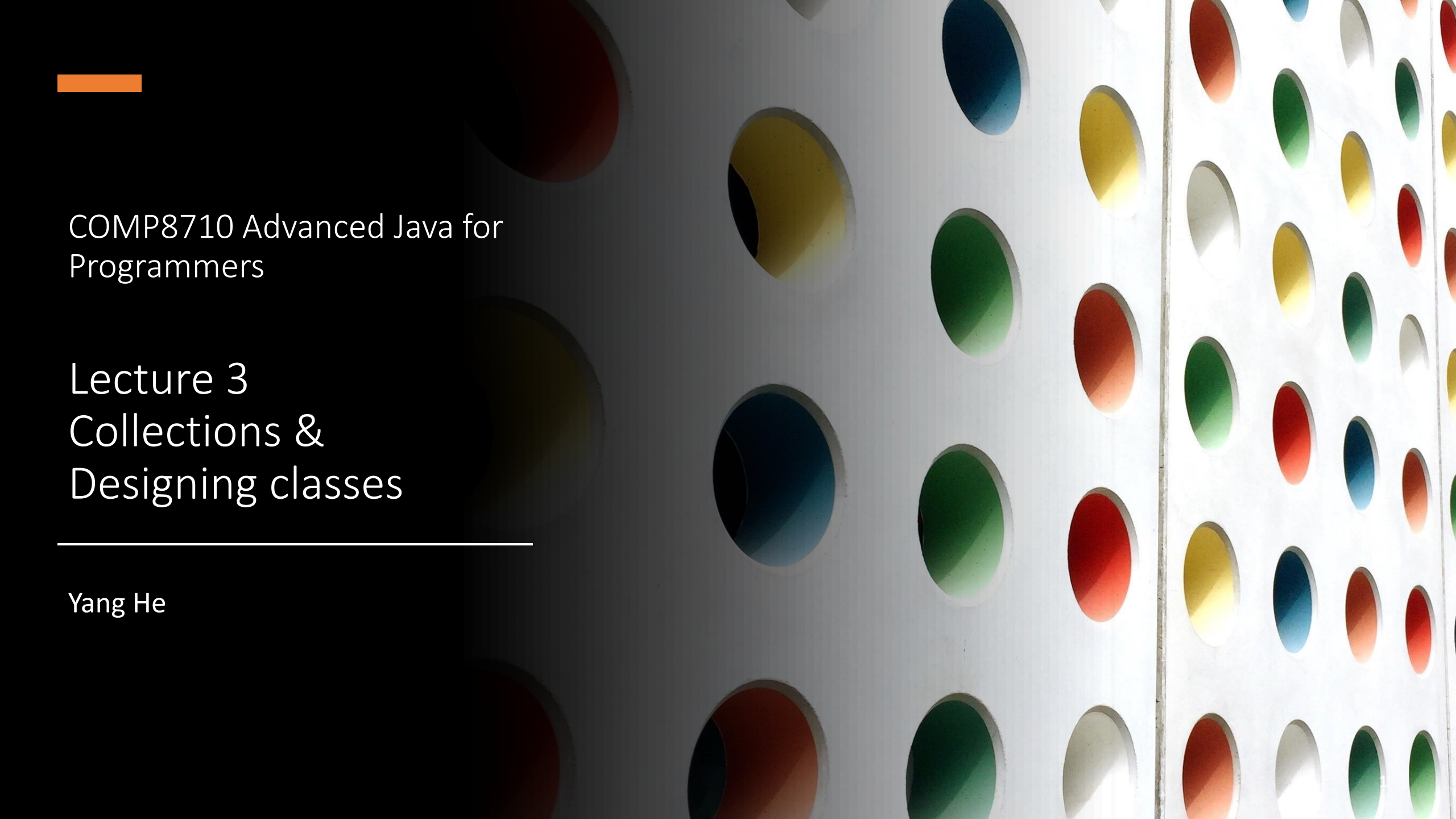


COMP8710 Advanced Java for Programmers

Lecture 3
Collections & Designing classes

Yang He

Topics

- Recap collections
  - ArrayList
  - HashMap
  - Iterator
  - Array
- Designing classes

Recap – Collections

Java API

- Example classes
  - `ArrayList`
  - `TreeSet`
  - `HashMap`
  - `TreeMap`
  - `Iterator`
  - `Arrays`
  - `Collections`
  - `Random`
  - `StringBuilder`
  - `Scanner`

Java 21 API:
https://docs.oracle.com/en/java/javase/21/docs/api/index.html

Main collections

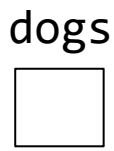
- ArrayList
- HashMap
- Iterator
- Array

ArrayList (1)

- ArrayList<E>
- Useful methods: add, size, get, set, isEmpty, contains
- E.g. an ArrayList of Dog objects

1) Declare a variable to store ArrayList of Dog

```
ArrayList<Dog> dogs;
```



2) Instantiate the variable dogs with an empty list

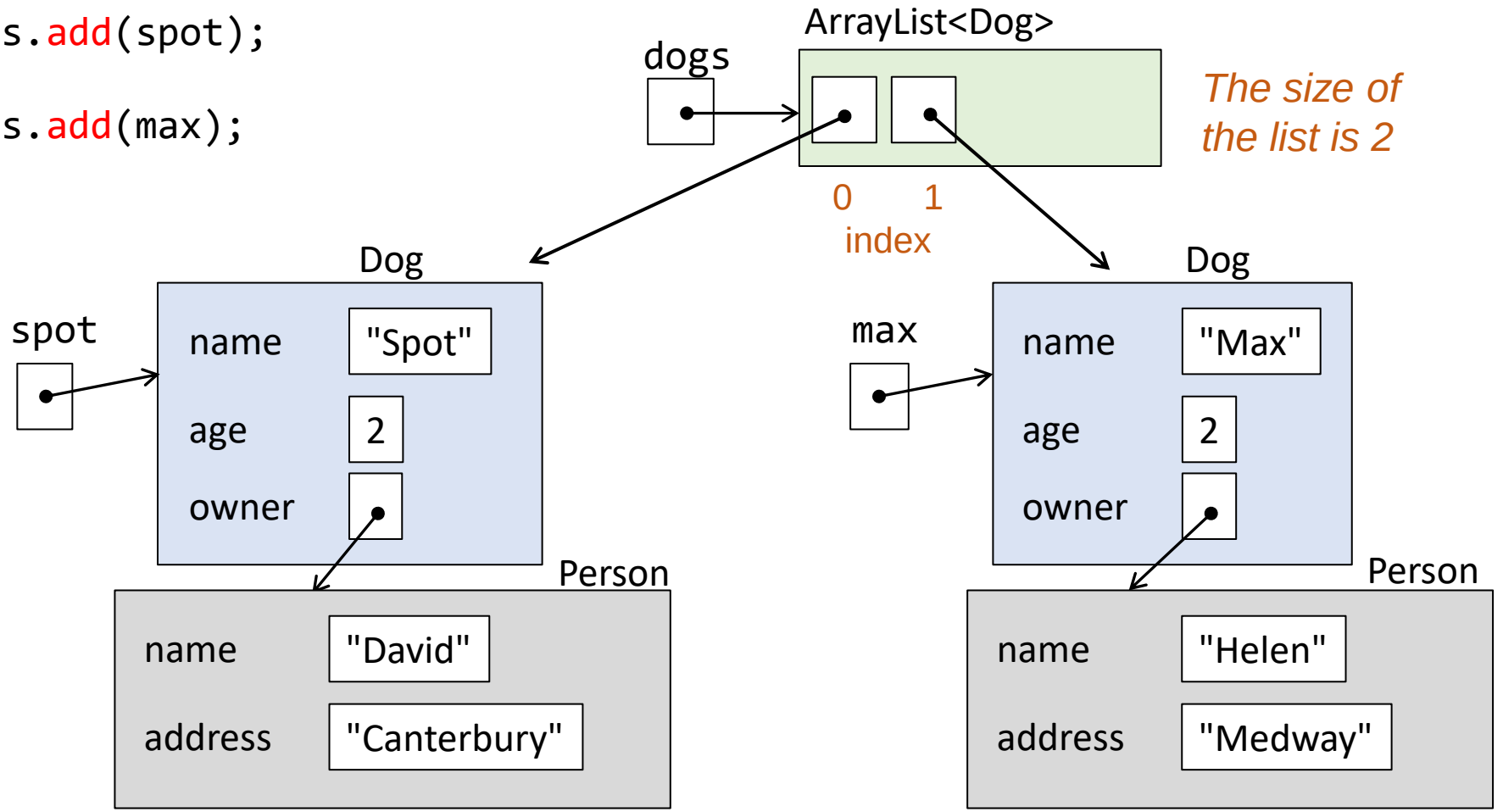
```
dogs = new ArrayList<>();
```

The size of the list is 0

ArrayList (2)

3) Add two Dog objects, spot and max, into the list **dogs**

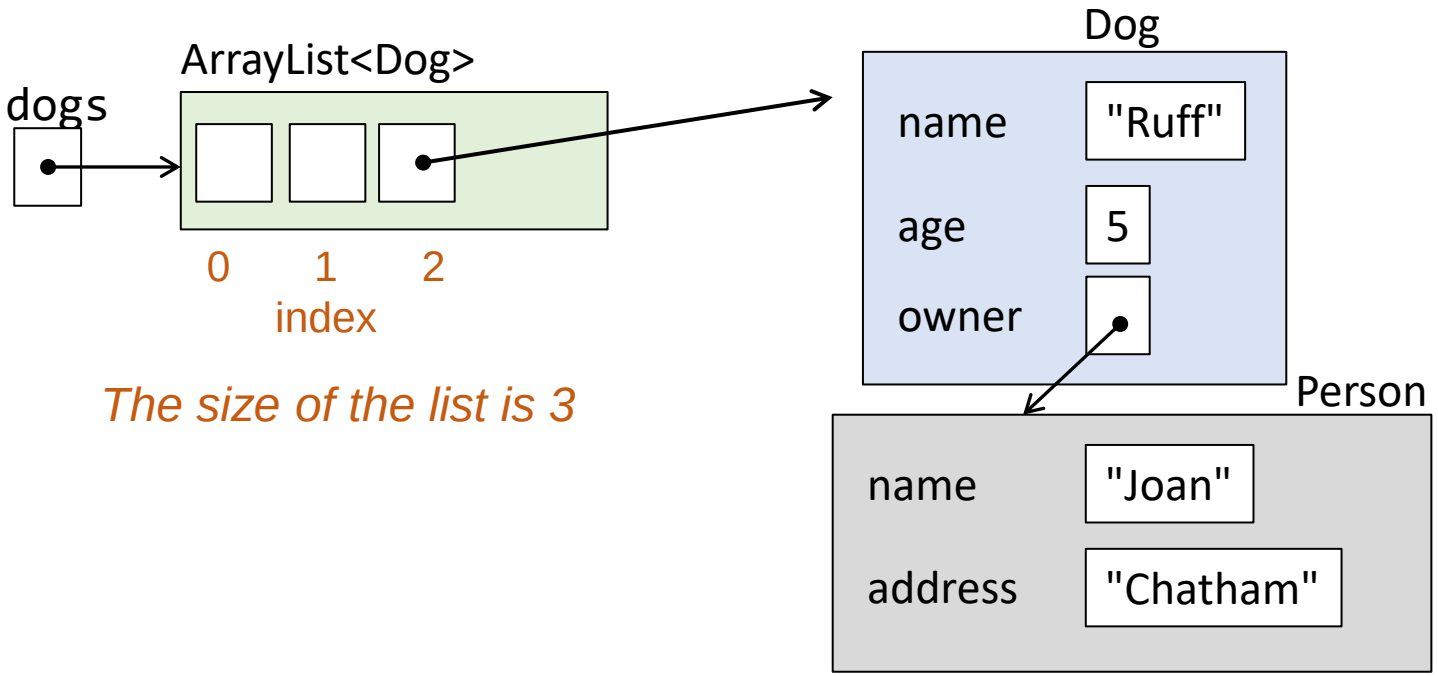
```
dogs.add(spot);

dogs.add(max);
```



ArrayList (3)

4) Add an anonymous Dog object into the list **dogs**

```
dogs.add( new Dog("Ruff", 5, new Person("Joan", "Chatham") );
```



HashMap

- **`HashMap<K,V>`**
- Useful methods: put, size, get, keyset, isEmpty, containsKey
- E.g. Use a HashMap to record the marks of students

1) Declare & instantiate a HashMap variable

```
HashMap<String,Integer> marks = new HashMap<>();
```

2) Add the records into the HashMap

```
students.put("Sam", 60);
students.put("Jo", 75);
```

Iterator class (1)

- `Iterator<E>`
- Useful methods: hasNext, next, remove
- Java collections have an iterator() method that returns an object of Iterator

Iterator class (2)

- Normally we should use an iterator explicitly when we want to modify a collection
- E.g. Remove all dogs that are older than 10 years old.

```
Iterator<Dog> it = dogs.iterator();

while (it.hasNext()) {
    var d = it.next();
    if(d.getAge() > 10) {
          it.remove();
    }
}
```

Homework:
Define a method that takes one parameter, age of int type, and removes all dogs that are older than the given age. It returns a list of dogs removed.

Sorting a list

- The `sort` method of `Collections` class
- `TreeSet`
- `TreeMap`

Java 21 API: https://docs.oracle.com/en/java/javase/21/docs/api/index.html

Array (1)

- An array has a fix size
- It can store primitive types or references (objects)
- Access is fast

- E.g. Use an array to record the number of visitors in each month:

```
int[] visitors = { 18, 25, 22, 51, 25, 65, 73, 82, 77, 32, 12, 48 };
```

Array (2)

- E.g. Find out which month has the least number of visitors.

```
int[] visitors = {18, 25, 22, 51, 25, 65, 73, 82, 77, 32, 12, 48};

var month = 1;
var least = visitors[0];
for (var k=1; k < visitors.length; k++) {
   if ( visitors[k] < least ) {
      least = visitors[k];
      month = k + 1;
   }
}
System.out.println("Month " + month);
```

Two-dimensional array (1)

- Array of array
- E.g. 4 assessment marks of 6 students

Student No.	A1	A2	A3	A4
1	66	43	78	54
2	32	51	64	40
3	85	78	67	70
4	65	41	54	48
5	43	25	36	30
6	52	68	75	74

Two-dimensional array (2)

- Using a 2-D array:

```
int[][] marks = {
                {66,43,78,54}, {32,51,64,40}, {85,78,67,70},
                {65,41,54,48}, {43,25,36,30}, {52,68,75,74}
            };
for (var row = 0; row < 6; row++) {
    for (var col = 0; col < 4; col++) {
        System.out.print(marks[row][col] + " ");
    }
    System.out.println();
}
```

```
66 43 78 54
32 51 64 40
85 78 67 70
65 41 54 48
43 25 36 30
52 68 75 74
```

Homework (challenging)

- Define a method that takes an array of String and returns a TreeMap<Integer, ArrayList<String>>, where the key is a number, and the value is a list of words containing the given number of letters.

Designing classes

Software

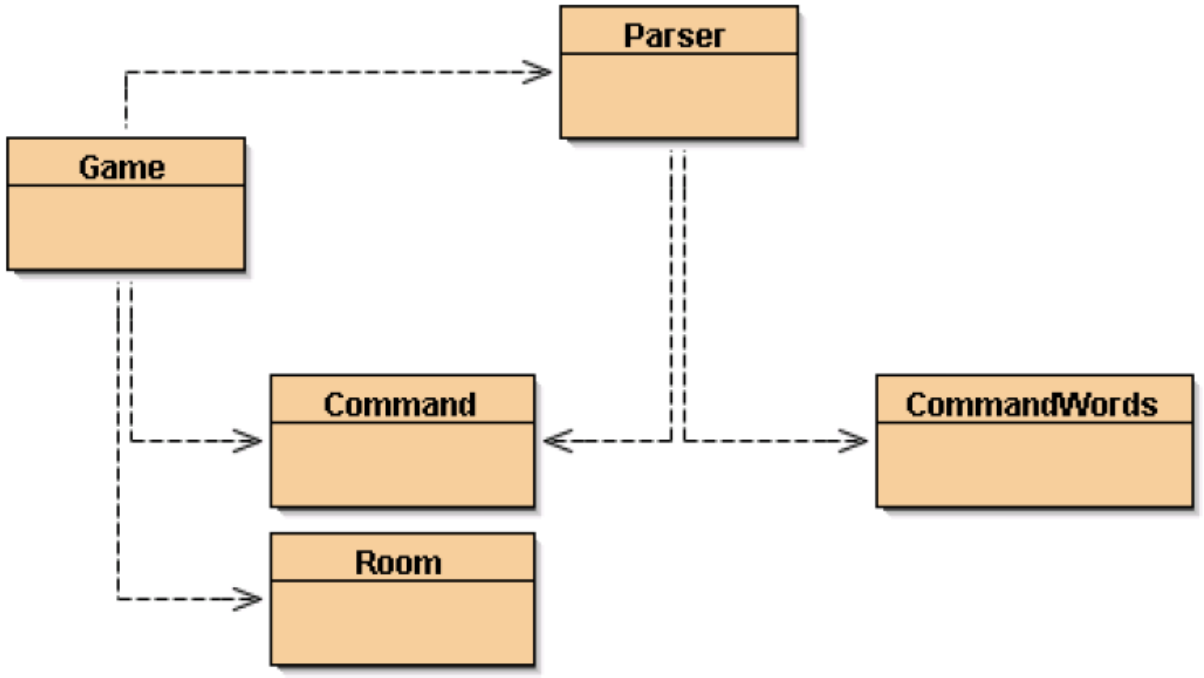
- Software changes
  - It is not like a novel that is written once and then remains unchanged
  - Software is extended, corrected, maintained, ported, adapted
  - The work is done by different people over time, often decades
- Change or die
  - There are only two options for software in the long term
    - Either it is continuously maintained
    - Or it dies
- Software that cannot be maintained will be thrown away

Software design

- In OOP, we aim to write classes that they are easy to
  - Understand
  - Maintain
  - Reuse
- Main concepts to be covered:
  - Responsibility-driven design
  - Coupling
  - Cohesion
  - Refactoring

World of Zuul

- Class diagram for zuul-bad:



Code duplication (1)

- Code duplication
  - Is an indicator of bad design
  - Makes maintenance harder
  - Can lead to introduction of errors during maintenance
- It often arises through copy-paste programming
  - Copying similar code from A to B, and then modifying B
  - Instead of abstracting the commonality and have A and B as instances of that common pattern

Code duplication (2)

- In zuul-bad: code duplication in Game class
  - printWelcome()
  - goRoom()

- We should introduce a new method instead
- E.g.

```
private void printLocationInfo()
```

Making extensions

- Well-designed code is easier to extend than poorly designed code
- In zuul-bad: a player can go north, east, south or west
- Suppose we want multi-level buildings and want to add directions up and down, what do we need to change?

```
In Game lass:
    createRoom()
    printWelcome()
    goRoom()
```

```
In Room class:
    setExits()
```

Code quality

- Two important metrics for code quality:
  - Coupling
  - Cohesion

Coupling (1)

- Coupling refers to links between separate units of a program
- If two classes depend closely on many details of each other, we say they are tightly coupled, otherwise we say they are loosely coupled
- We aim for loose coupling
- Loose coupling makes it possible for us to:
  - Understand one class without reading others
  - Change one class without affecting others

  Therefore, it improves maintainability

Coupling (2)

- In zuul-bad: many places where all exits are enumerated – bad design
- A better way: use a HashMap rather than separate fields for exits, allow any number of exits
  - Mapping: direction → room
- This change should *only* affect the *implementation* of the Room class and not its *interface*
  - Changing the implementation of one class should not affect other classes — loose coupling
  - Do not change the method signatures (interface) so that other parts of the code do not require modifications

Encapsulation

- Encapsulation is a mechanism that helps with maintaining a loose coupling between classes
- In zuul-bad: The exit fields in Room are public! – bad design
- A better way:
  - Hide implementation from the outside
  - Use methods not public fields
- Reveal *what* it does, *not how* it does it

Responsibility-driven design

- Question: where should we add a new method (which class)?
- Each class should be responsible for manipulating its own data
- The class that owns the data should be responsible for processing it
- Responsibility-driven design (RDD) leads to low coupling

Localizing changes

- One aim of reducing coupling and responsibility-driven design is to localize change
- When a change is needed, as few classes as possible should be affected

Implicit coupling??

- Are there any ways in which different parts of the zuul-bad code are coupled other than through duplicated code, knowledge of the internal structure of other classes, etc?
- How easy would it be to convert zuul-bad so that it could be played in another language, such as French?
- How many changes would you have to make?
- How could you solve this problem?
  - Check: https://docs.oracle.com/javase/tutorial/i18n/index.html

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for
- If each unit is responsible for *one single* logical task, we say it has high cohesion, otherwise we say it has low ochesion
- Cohesion applies to classes and methods
- We aim for high cohesion

High cohesion

- High cohesion makes it easier for us to understand what a class or method does
- A class should represent one *single*, well-defined entity
- A method should be responsible for one and *only one* well defined task, e.g. printWelcome

Benefits of high cohesion

- Readability
  - Easier to read means easier to maintain
  - Easier to find where to start
- Reuse
  - With just one responsibility, it is possible to use a class in different contexts

Thinking ahead

- When designing a class, we try to think of what changes are likely to be made in the future
- We aim to make those changes easy

Refactoring

- When classes are maintained, often code is added
- Classes and methods tend to become longer
- Every now and then, classes and methods should be refactored to maintain high cohesion and loose coupling

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes
  - First do the *refactoring only*, without changing the functionality
  - Test *before* and *after* refactoring to ensure that nothing was broken
- IDEs like IntelliJ, Eclipse, Netbeans offer extensive support for refactoring

Design questions

- Common questions
  - How long should a class be?
  - How long should a method be?

  We can answer these questions in terms of cohesion and coupling
- Design guidelines
  - A class is too complex if it represents more than one logical entity
  - A method is too long if it does more than one logical task

Review

- Programs are continuously changed
- It is important to make this change possible
- Quality of code requires much more than just performing correctly at one time
- Code must be understandable and maintainable
- Good quality code avoids duplication, displays high cohesion, low coupling
- Coding style (commenting, naming, layout, etc.) is also important
- There is a big difference in the amount of work required to change poorly structured and well-structured code