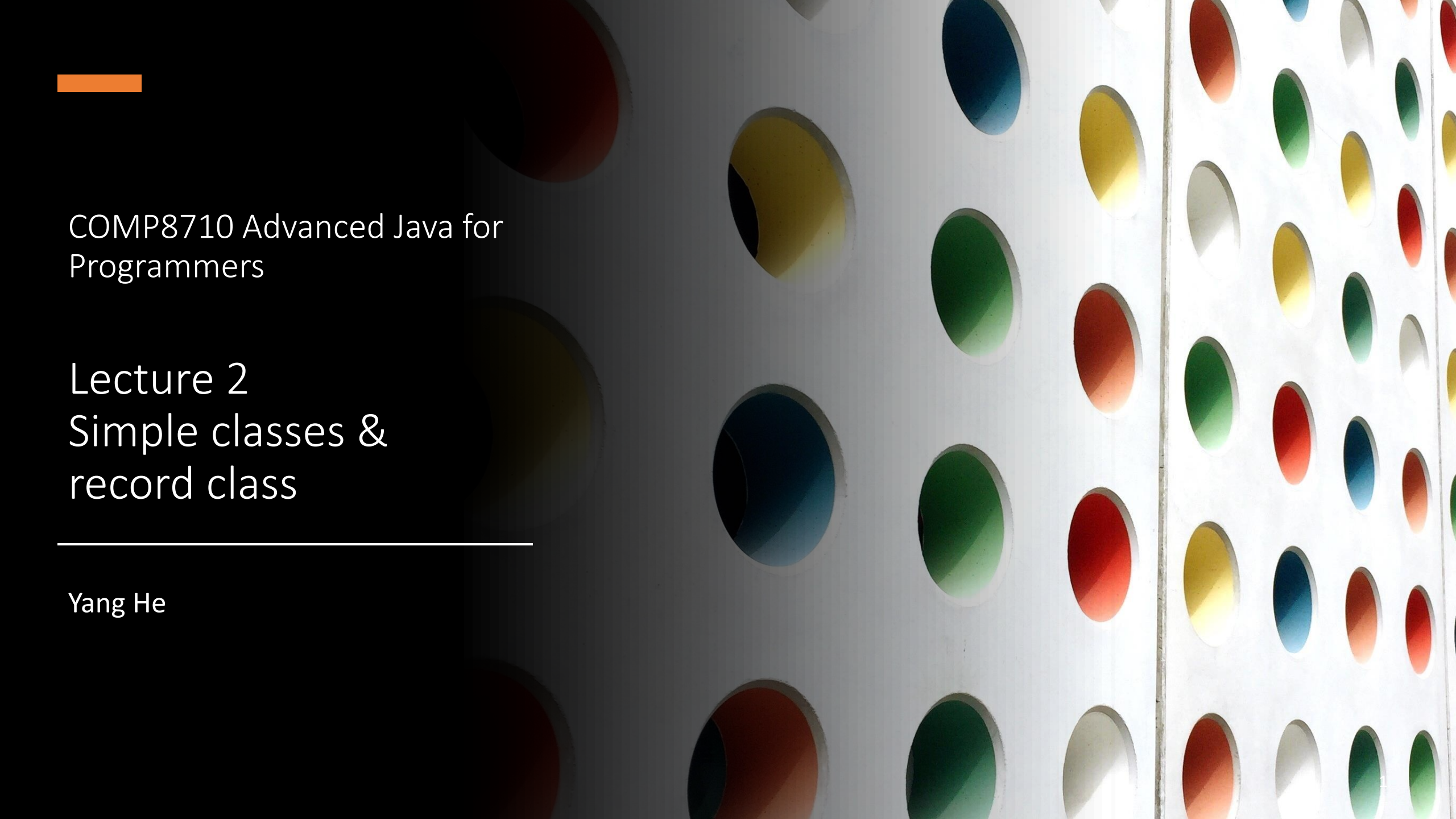


COMP8710 Advanced Java for Programmers

Lecture 2
Simple classes & record class

Yang He

Topics

- Object-oriented (OO) design
- Objects and classes
- Object interaction
- Main principles of OOP
- Components of classes
- Object instantiation & message passing
- Record classes

Object-oriented (OO) design

- The world is made of things that interact
- OO Software design aims to model the real world
- We model things with objects
  - For example, in a model of an online purchasing system, objects might include customers, vendors and bank accounts
- Objects have
  - State, e.g. name, balance, etc.
  - Operations (or methods) , e.g. get a customer's name, deposit or withdraw money, etc.
- We collect objects with the same functionality and the same kind of state (but not necessarily the same value) into a class
  - E.g. the objects representing the bank accounts of 2 different customers are both instances of the Account class, but these instances have vastly different values

Classes and objects

- Those new to Object-oriented programming (OOP) often confuse classes and objects
- One way to think of a class is as a blueprint that spells out how objects are to be manufactured
  - The blueprint specifies how object components (fields and methods) fit together
- Operationally, the data in the fields of one object are different (stored at a different address) from the data of another object

Example – Online purchasing (1)

- A simple on-line purchasing system allows customers to purchase goods from a vendor.
  - The vendor sells only one type of good
  - Customers must provide the vendor with their bank account number and the amount to debit
  - Purchases only succeed if the account has sufficient funds, in which case the cost of the good is transferred from the customer's account to the vendor's bank account

Example – Online purchasing (2)

- Questions:
  - What objects and classes are required for this simple on-line purchasing system?
  - What state should each class maintain?
  - What operations should each class provide?
  - What are the relations between the classes?
  - What messages need to be sent between objects?

Objects interaction

- Things in the real world interact
- We can think of Java objects interacting by exchanging messages
- Suppose a customer wants to purchase 3 goods from a vendor:

```
int cost = 3 * vendor.getPrice();
vendor.sell(account, cost);
```

- What happens?
  - Send a getPrice message to the object vendor, which returns an int
  - Send a sell message with the account object and the cost to the vendor; there's no return value

Object-oriented programming principles

Principles of OOP

- Object oriented programming and design is based on the four main principles:
  - Abstraction
  - Encapsulation
  - Inheritance (hierarchy)
  - Polymorphism

Abstraction

- To represent something in a computer, we need to express its essential characteristics
  - E.g. DVLA might represent a car by the essential characteristics that are important to them: VIN, colour, make, license plate, owner history (and nothing else)
- The process of abstraction is the business of throwing away unimportant details, e.g. service history, interior decor, etc.
- The DVLA record is a data abstraction

Encapsulation

- The step beyond data-abstraction is recognizing that the operations on the data are as important as the data itself
- Encapsulation is the process of coupling data and operations into an indivisible organizational unit so we can say:

  “This is how DVLC represents a car, and these are the only operations that can be applied to a car."
- The classic structured programming device for encapsulation is the Abstract Data Type
  - ADTs fire-wall data (so that the user cannot poke inside)
- Java classes are an example of an ADT (and much more…)

Components of classes (1)

- A class is a collection of data members (fields), methods, and access control mechanisms

By convention:

- *Class names start with an upper-case character*
- *Class members and variable names start with a lower-case character*
- *Class fields are private*
- *Use "camel case" for variable names, but capital letters with _ for constants*

```
public class Room {
    // fields
    private String description;
    private int capacity;

    // constructor
    public Room(String desc, int size) {
        description = desc;
        capacity = size;
    }

    // accessor
    public int getCapacity() {
        return capacity;
    }

    // mutator
    public void setCapacity(int size) {
        capacity = size;
    }
}
```

Components of classes (2)

- Class constructors are used for creating instances of that class
- Every class must have at least one constructor
- Constructors must have the same name as the class itself
- They do not return a value and thus have no return type in their header

```
public class Room {

    public Room(String desc, int size) {
        description = desc;
        capacity = size;
    }
    ...
}
```

Accessor and mutator (1)

- Objects exchange messages (method calls) which may change their state, but we had no direct control of that state (*normally class fields are private*)
- There are two kinds of methods:
  - Accessors get the value of some part of an objects state
    - E.g. in Vendor (giving the vendor's price as an int):

```
public int getPrice() { return price; }
```

  - Mutators modify (mutate) an object's state (in a controlled fashion)
    - E.g. in BankAccount:

```
public void deposit(int amount) {
   if (amount > 0) balance += amount;
}
```

Accessor and mutator (2)

- We want to control access to the internal state of an object
- By making data members (i.e. class fields) private, we forces a user to request a change via a mutator
  - The request can be rejected if it would scramble the state
- Accessor and mutator methods do not expose representation, so encapsulation is preserved

```
private int capacity;

public void setCapacity(int size) {
     if (size < 0)
         throw new IllegalArgumentException("Size must be > 0.");

     capacity = size;
}
```

- *private – accessible only within the class*
- *public – accessible by any classes*

Good practice

- It's a good practice to write all your code in terms of accessors and mutators
  - Safety: the class can control access to state, thus ensuring it is always consistent
  - Easy development: there is only one place that needs to be changed, e.g. if the representation of the state changes
  - Enforcement: use public and private access modifiers to let the compiler enforce this methodology
- We should use accessors and mutators within a class for these reasons, *even though direct access is legal within a class*

Object instantiation and message passing

Declaring variables and instantiating objects

```
int total = 0;      // 0
Room myOffice;      // is null
myOffice = new Room("Office", 2);    // assigns a reference to the new Room object
```

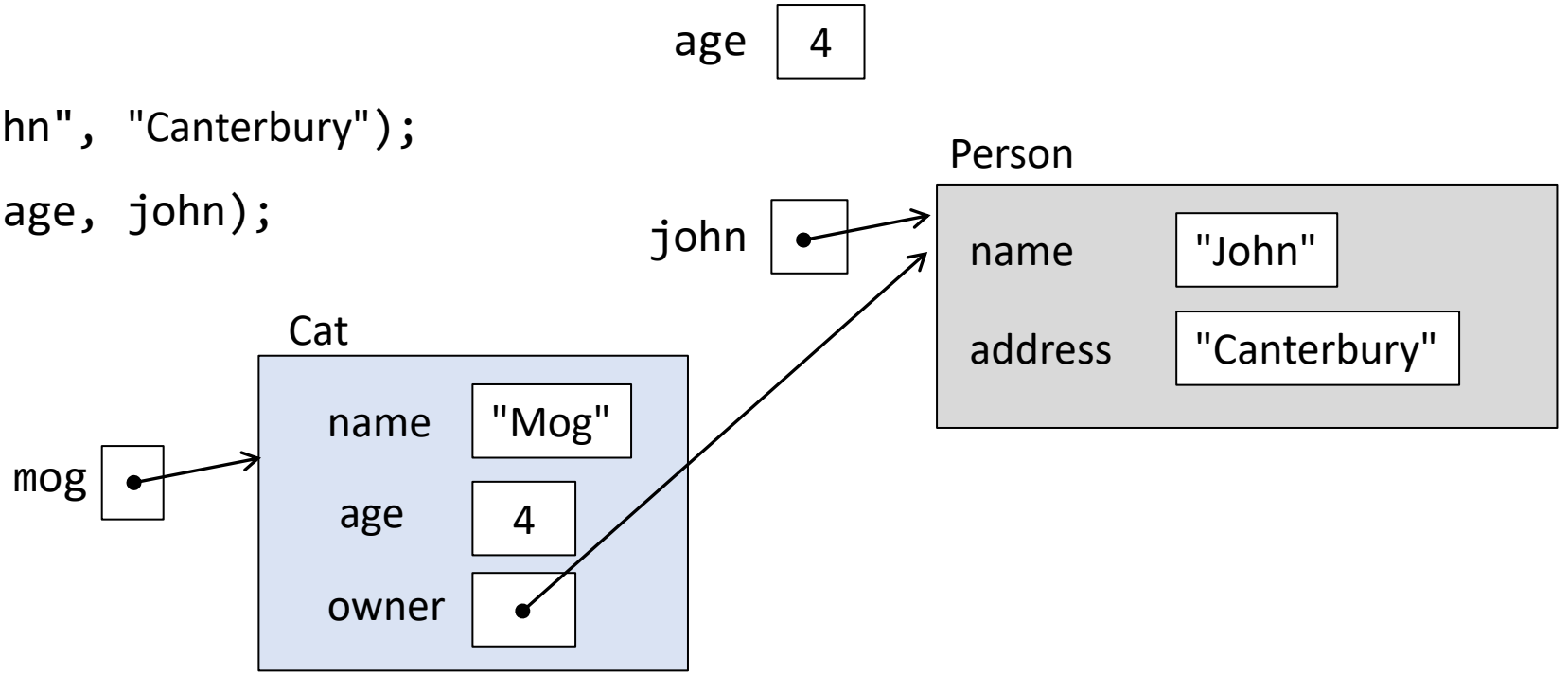
- Variable myOffice is declared as Room type; it stores a reference to a Room object (or to a subclass of Room)
- To create an object, we invoke a class constructor that allocates and initializes the object, and returns a reference to the new object
  - As an object is an instance of a class, object creation is also known as instantiation
  - Constructors typically fill in the data members of newly minted objects

Parameter passing

- Primitive types
  - A copy of the contents (value) of an int, say, is used in the method
  - Modifying it will not affect the value of the original argument
- Reference types
  - A copy of the contents (value) of a reference (like a pointer) to an object is passed to the method
  - If the ref contains the address of an (instantiated) object, then any change that the method makes to the parameter is made to the object
  - The effect is similar to pass-by-reference

Parameter passing example (1)

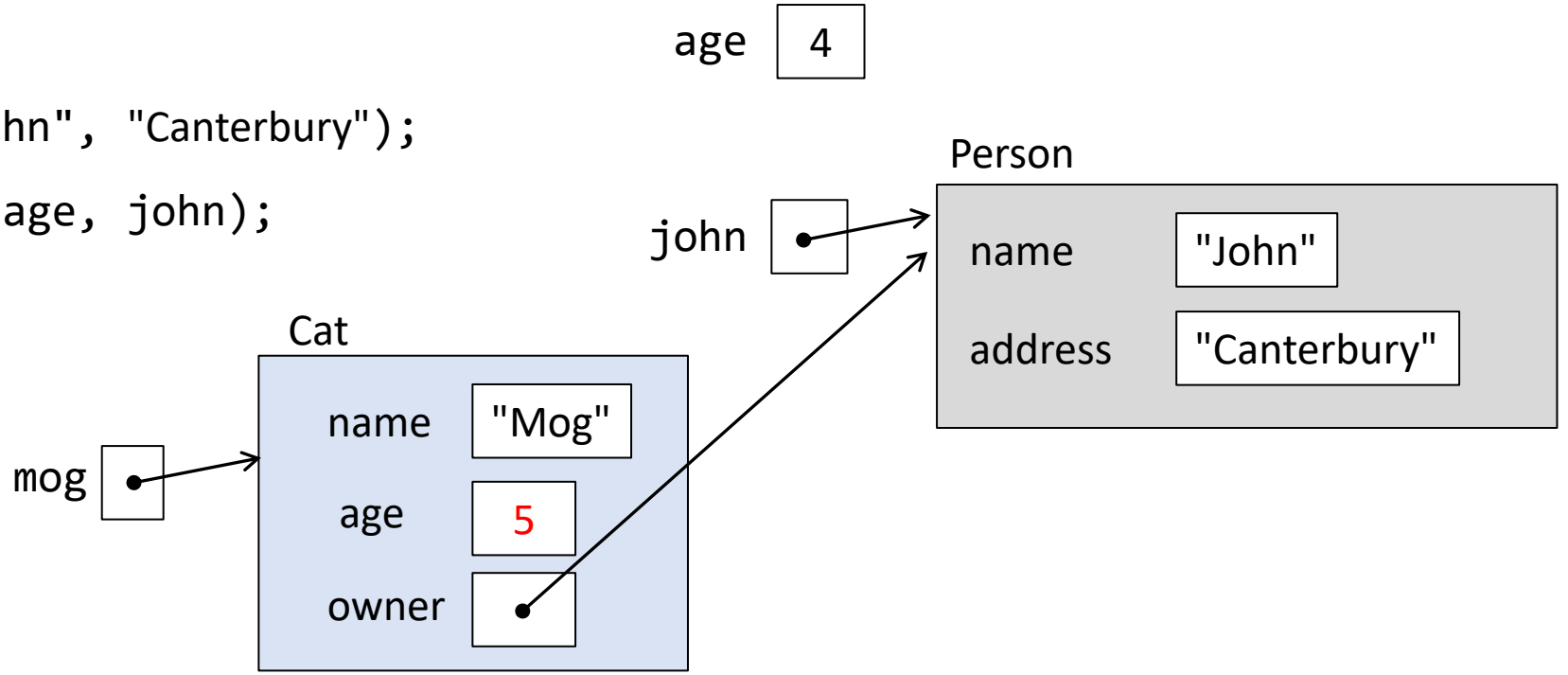
```
var age = 4;
var john = new Person("John", "Canterbury");
var mog = new Cat("Mog", age, john);
```



- The Person class constructor:
  public Person(String name, String address)
- The Cat class constructor:
  public Cat(String name, int age, Person owner)

Parameter passing example (2)

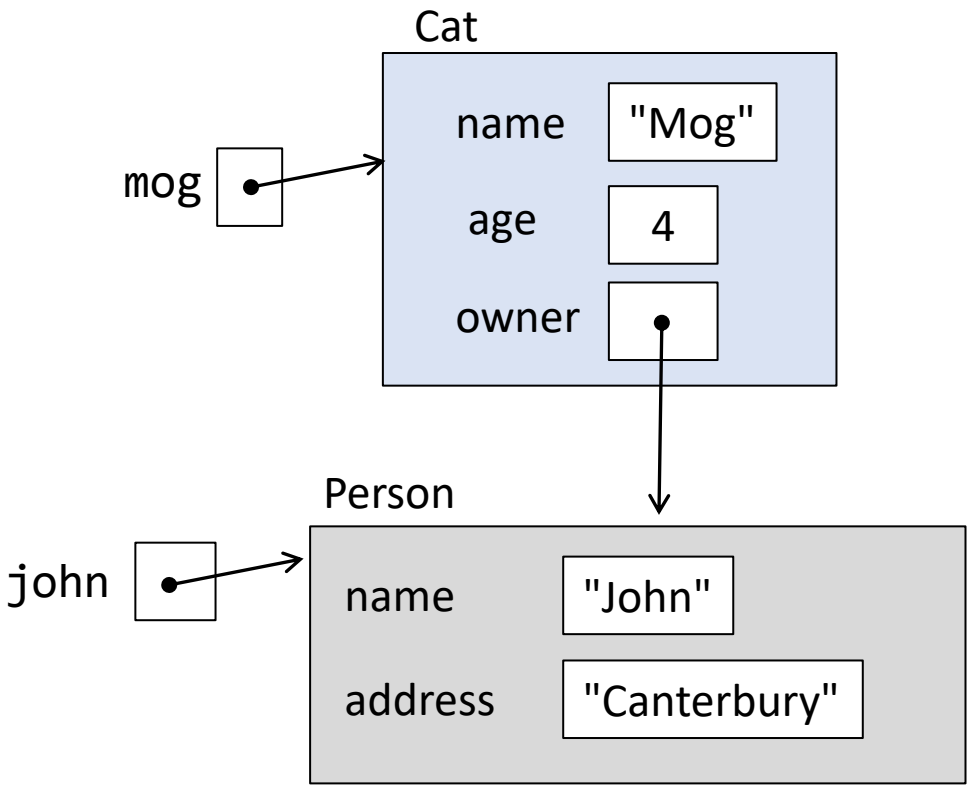
```
var age = 4;
var john = new Person("John", "Canterbury");
var mog = new Cat("Mog", age, john);
mog.setAge(5);
```



- The Person class constructor:
  public Person(String name, String address)
- The Cat class constructor:
  public Cat(String name, int age, Person owner)

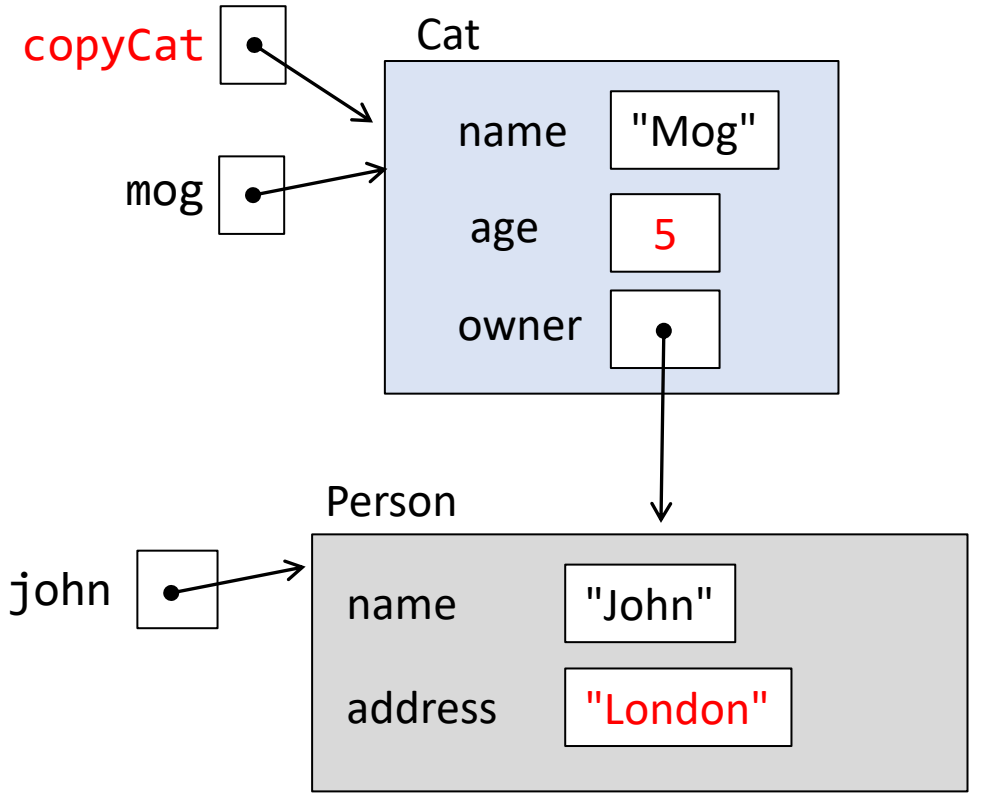
Quiz – What would be printed out?

```
var age = 4;
var john = new Person("John", "Canterbury");
var mog = new Cat("Mog", age, john);
var copyCat = mog;
mog.setAge(5);
john.setAdress("London");
System.out.println("Age: " + copyCat.getAge());
System.out.println("Owner's address: " +
                   copyCat.getOwner().getAddress());
```



Quiz – What would be printed out?

```
var age = 4;
var john = new Person("John", "Canterbury");
var mog = new Cat("Mog", age, john);
var copyCat = mog;
mog.setAge(5);
john.setAdress("London");
System.out.println("Age: " + copyCat.getAge());
System.out.println("Owner's address: " +
                   copyCat.getOwner().getAddress());
```



Homework

- What does this print? Why? How can we fix it?

```
String letters = "ABC";
char[] numbers = {'1','2','3'};
System.out.println(letters + " as easy as " + numbers);
```

Record classes

Immutable data (1)

- Some data are immutable
  - e.g. student records retrieved from a database
- We can create a data class with the following:
  - All fields are private and final
  - accessor method for each field
  - public constructor with arguments for all fields
  - equals method that returns true for when all fields match
  - hashCode method that returns the same value when all fields match
  - toString method that includes the names and values of all fields

Immutable data (2)

- E.g.
  Book class

But there are lots of boiler-plate code!

```
public final class Book {
    private final String title;
    private final String author;
    public Book(String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle() { return title; }
    public String getAuthor() { return author; }
    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null || obj.getClass() != this.getClass()) return false;
        var that = (Book) obj;
        return this.title.equals(that.title) && this.author.equals(that.author);
    }
    public int hashCode() {
        return Objects.hash(title, author);
    }
    public String toString() {
        return "title=" + title + ", " + "author=" + author;
    }
}
```

Record class (1)

- Java 14 introduced record class to reduce boiler-plate code
- A record class declaration includes a name, parameters and a body
- E.g.

```
public record Book(String title, String author) {}
```

- Java compiler *automatically* generates the private final fields, a public constructor, assessors for all fields, equals, hashCode and toString methods

Record class (2)

- For immutable data we can simply use record instead of class
- E.g.

```
public record Book(String title, String author) {}

var book = new Book("Great Expectations", "Charles Dickens");
String title = book.title();                                    Accessor method
System.out.println(book.toString());     //or simply book

Output:  Book[title=Great Expectations, author=Charles Dickens]
```

Canonical constructor

- The default record class constructor is called canonical constructor
- We can override the canonical constructor
- E.g. checking if the given age is valid

```
public record Cat(String name, int age) {

    public Cat(String name, int age) {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }

        this.name = name;
        this.age = age;
    }
}
```

Compact constructor

- Compact constructor is a concise way of overriding the canonical constructor

```
public record Cat(String name, int age) {

    public Cat {
        if (age < 0) {
            throw new IllegalArgumentException("Age cannot be negative");
        }
    }
}
```

Find out more about record classes at:
https://docs.oracle.com/en/java/javase/17/language/records.html