Session 2: OOP Basics

Intro to OOP

Object-oriented Programming

- OOP: **O**bject-**o**riented **P**rogramming
- OOP is a paradigm of programming…
- The world is made of objects with relationships and behavior that affect each other and the environment.

  A car **has** an engine.

  A person **drives** a car.

  A person **drives** a car by **using** the steering wheel.

- OOP models the world as objects and their interaction in the source code.
- Java is OOP language

OOP Principles

Abstraction

Hiding the unimportant details.

e.g. A petrol / electric / hybrid car.

Inheritance

Building new objects on top of existing ones.

e.g. an armoured vehicle

Encapsulation

Protecting integral parts of an objects from outside and providing only an interface to perform actions.

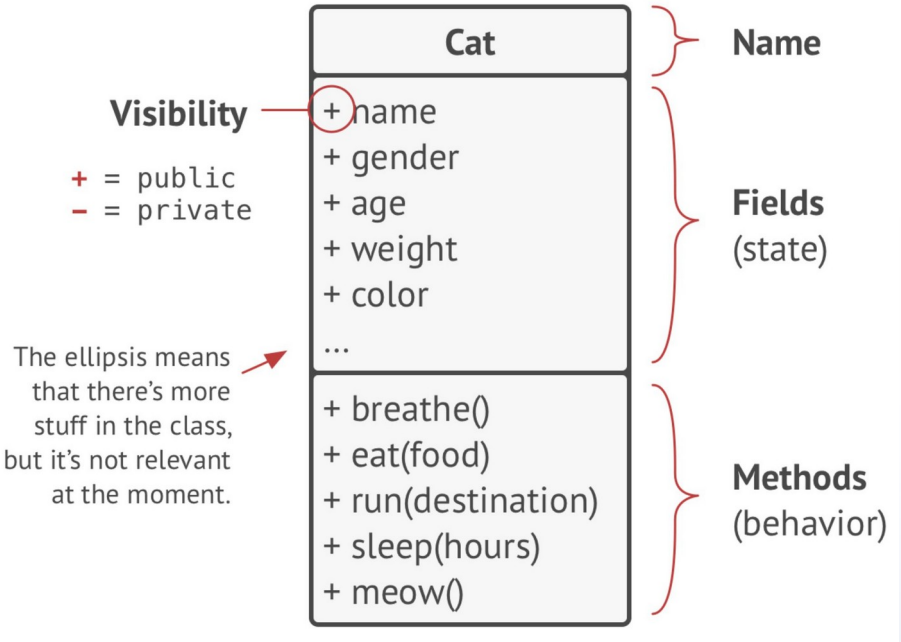
e.g. steering wheel

Polymorphism

Detecting different behavior of objects in the hierarchy in runtime.

e.g. a hybrid vehicle

Example: Cat



- UML (Unified Modeling Language) is a way of demonstrating OOP but in diagrams instead of code
- It is useful to show how we think in objects and what classes we design

Cat class and instances

Oscar: Cat

```
name    = "Oscar"
sex     = "male"
age     =  3
weight  =  7
color   =  brown
texture =  striped
```

Luna: Cat

```
name    = "Luna"
sex     = "female"
age     =  2
weight  =  5
color   =  gray
texture =  plain
```

- A “Cat” is the “class” that defines a name, fields, and methods
- Fields define the state
- Methods define the behavior
- The two cats have same set of fields but different values

- Cat = class
- Oscar, Luna = instances (a.k.a. objects)

Class vs. object

Classes

- A class is a named collection of fields that hold data values and methods that operate on those values.

```
public class Cat {

    // Fields
    public String name;
    public char gender;
    public int age;
    public double weight;
    public String color;

    // Methods
    public void breathe() { }
    public void eat(Food food) { }
    public void run(Destination destination) { }
    public void sleep(int hours) { }
    public void meow() { }
}
```

Instantiation

- Instantiation is creation of object instances from a define class.
- Class = cookie cutter
- Instances = cat cookies
- In Java instantiation is done by the “**new ()**” operator.
- Instantiated objects are stored
- In heap
- Every class has a “constructor”
- Constructor is a special method that is executed when object is instantiated (only once)

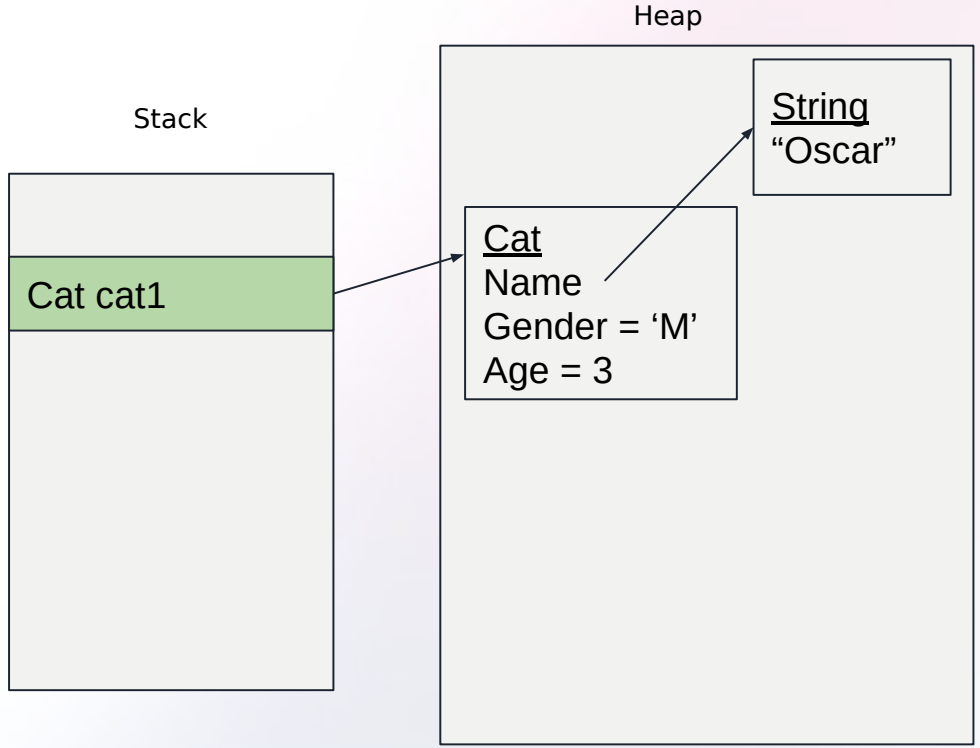
```
public static void main(String[] args) {
    Cat cat1 = new Cat();
    cat1.name = "Oscar";
    cat1.gender = 'M';
    cat1.age = 3;
    cat1.weight = 7;
    cat1.color = "brown";

    Cat cat2 = new Cat();
    cat2.name = "Luna";
}
```

Instantiation

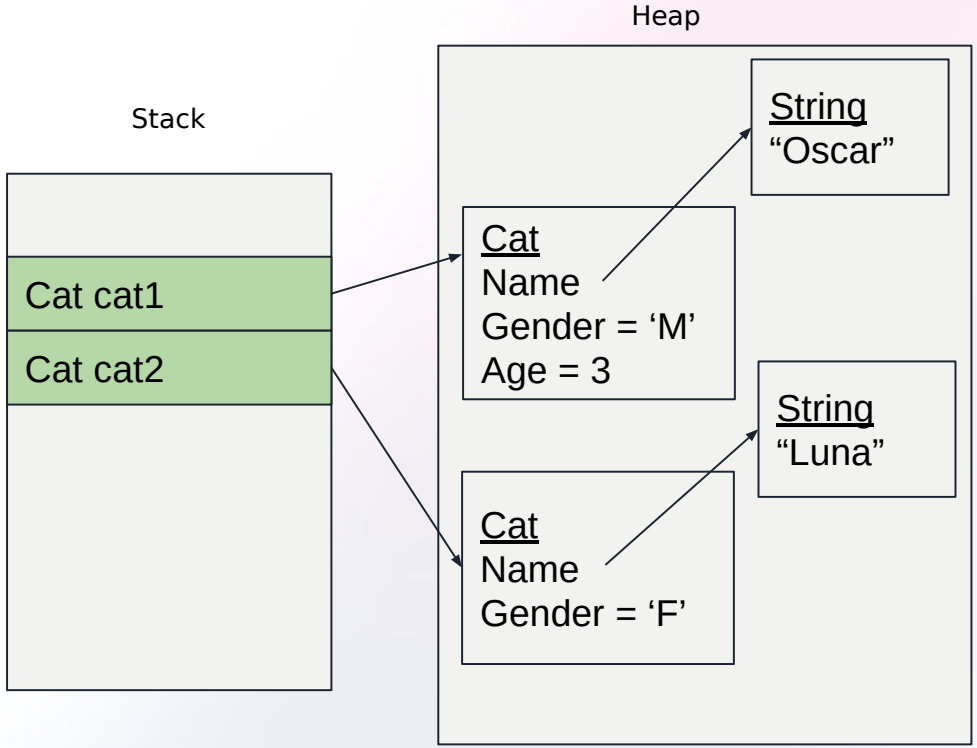
Reference stored in stack

```
Cat cat1 = new Cat();
cat1.name = "Oscar";
cat1.gender = 'M';
cat1.age = 3;
```

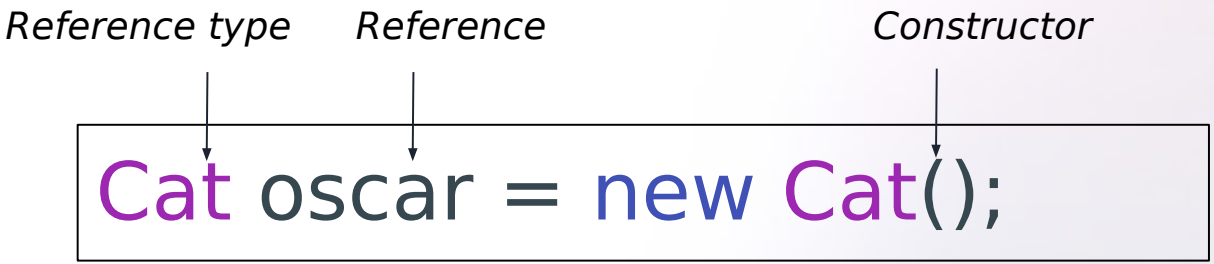


Instantiation

```
Cat cat1 = new Cat();
cat1.name = "Oscar";
cat1.gender = 'M';
cat1.age = 3;
Cat cat2 = new Cat();
cat2.name = "Luna";
cat2.gender = 'F';
```



Reference vs. object



Exercise #1

1. Set default cat age when instantiating a cat

2. Make it possible to choose a name when instantiating a cat

```
public class Cat {

    public Cat() {
        System.out.println("Cat construction is executed");
    }

    // Fields
    public String name;

    ...
}
```

Class and instance members

- The fields and methods defined with
static keyword are called static members.

- Non-static fields and methods are owned by the instance.

- Every instance has the same fields and methods because they **belong to the class**.

```
public class Cat {

    // Class-member field
    public static int count = 0;

    // Class member method
    public static String describe() {
        return "https://en.wikipedia.org/wiki/Cat";
    }

    // Instance-member fields
    private String name;
    public char gender;

    // Instance-member methods
    public void breathe() {
    }
}
```

Instance this reference

- Every class instance has a special variables denoted by **this** keyword
- The variable **this** is a reference to the current instance of the class
- The variable **this** has no meaning in class members (static members)
- The example also shows that a class can have multiple constructors

```
class Cat {
    private String name;
    private char gender;

    // constructors also have access modifiers
    public Cat(String catName) {
        name = catName;        // Inferred this
    }

    public Cat(String name, char gender) {
        this.name = name;      // Explicit this
        this.gender = gender;
    }

    public String getName() {
        return name;           // Inferred this
    }
}
```

Making the cat design more logical...

- Think of a pet cat. Should you change his name after a while?
- Who decides the age of a cat?
- How to fix this?

```
public static void main(String args[]) {
    Cat cat1 = new Cat("Oscar");

    System.out.println(cat1.name);

    cat1.name = "Thomas";
    cat1.age = 5;
}
```

Cat name must be read-only

- We’ve seen **public** keyword a lot. If there is **public**, there should also be **private**, righ
- Let’s change Cat name and age to private.
- We want to be able to set the name of the cat when instantiating it but not be able to change it afterwards.
- We still want to be able to get the name of the cat.
- This is an example of **encapsulation**.

Access Modifiers

Class members (fields and methods) can be:

- **public**
  The field/method can be used from inside and outside the class
- **private**
  The field/method can be used from inside the class only
- **protected**
  The field/method can be used from inside the class and its subclasses, and other classes in the same package.
- **package-private (empty)**
  The field/method can be used from inside same package only.

```
class Cat {
    private String name;
    public char gender;
    protected isPet;

    // constructors also have access modifiers
    public Cat(String catName) {
        name = catName;
    }

    public String getName() {
        return name;
    }

    private void meow() {
    }
}
```

Class Inheritance

- A class can **extend** another class
- Subclass inherits members from superclass
- Subclass can **override** methods of its superclass
- Animal = superclass
- Cat = subclass
- A class can also **implement** an interface
- An interface is like a class but without implementation
- A class must override methods of the interface it implements

```
public class Cat extends Animal implements Walkable {


}

public class Animal {


}

public interface Walkable {
    void walk();
}
```

Class Inheritance

- A class can **extend** another class
- Subclass inherits members from superclass
- Subclass can **override** methods of its superclass

- Animal = superclass
- Cat = subclass

```
public class Animal {

}

public class Cat extends Animal {

}

public class Dog extends Animal {

}

public class PersianCat extends Cat {

}
```

Homework 3

Modify the source code in work folder such that two instances of Cat having the same properties (e.g. name, age, ...) are considered equal. However, a Cat and a Dog should not be considered equal even if they have the same name or other attributes.

P.S. Add some fields in Dog class.