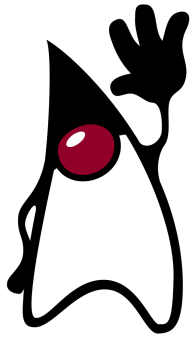
Classes for the Masses:

Inheritance, polymorphism, containment, and recursion - oh my!

CO7005 Software Development Techniques



Dr Stuart Cunningham
s.cunningham@chester.ac.uk

Inheritance

noun

I. The action or fact of inheriting.

I.2 *transferred* and *figurative.*

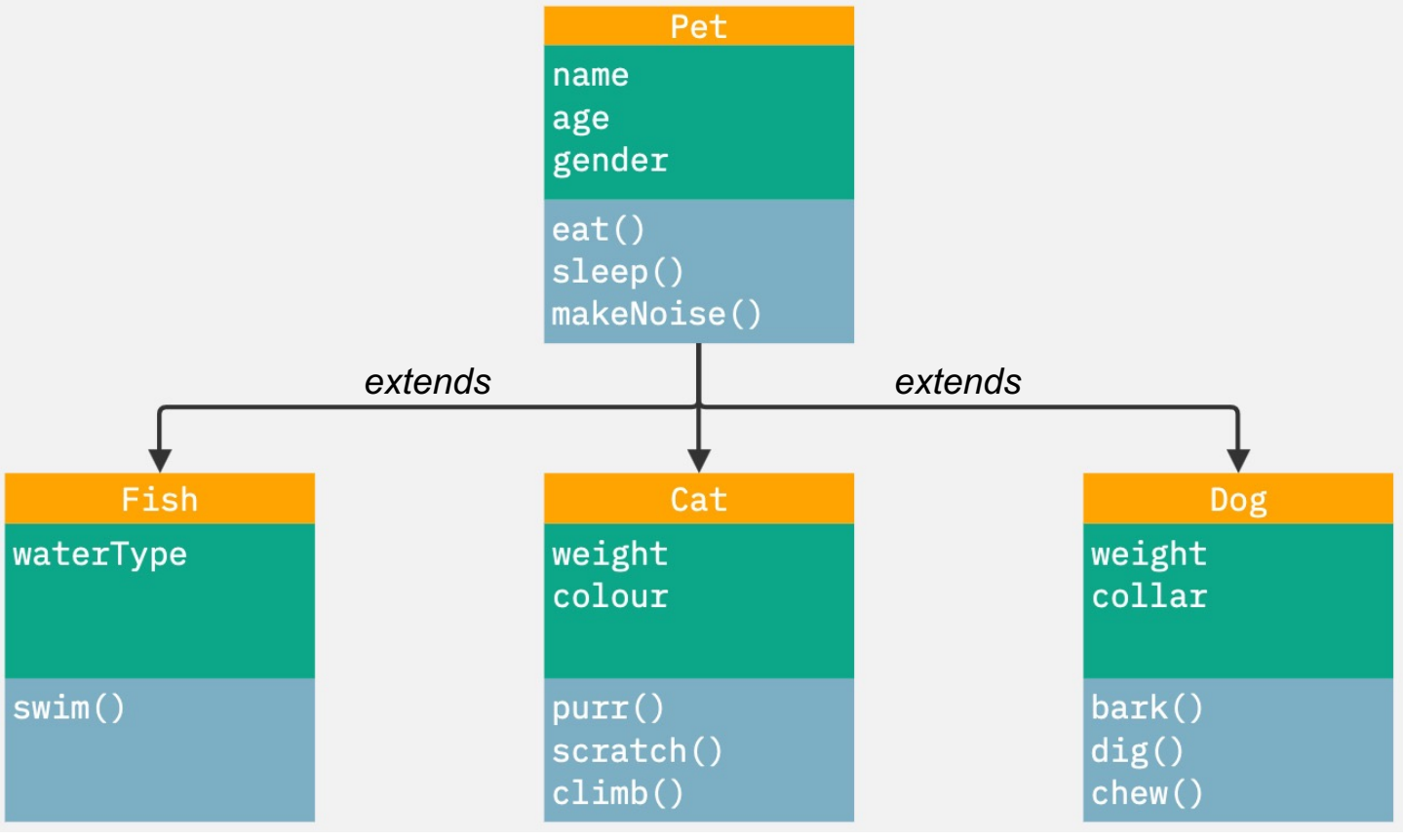
b. Natural derivation of qualities or characters from parents or ancestry.

Oxford English Dictionary, s.v. “inheritance, n., sense I.2.a”, September 2023. https://doi.org/10.1093/OED/7881937317

Inheritance

- The passing of characteristics from *parent* to *child*
- Or in Java, a *super-class* to a *sub-class*
  - *Variables* and *methods* are passed from super-class to sub-class
  - N.B.: *private* variables and methods do not
- Supports a *hierarchy* of objects to be formed
- Objects become *specialised* as sub-classes *extend*
- In Java, all objects created are sub-classes of *Object*

Inheritance



Inheritance and Constructors

- Constructors are responsible for instance variables in their super-class and sub-class respectively in two ways
  1. A sub-class constructor can set its own instance variables and use setters to configure super-class instance variables
  2. A sub-class contains the method `super(params)` in first line of its own constructor
- The method `super()` always refers to the class *immediately above* the sub-class calling it

```
class Pet {
   private String name;
   private int age;
   private char gender;

   Pet (String name, int age, char gender) {
      this.name = name;
      this.age = age;
      this.gender = gender;
   }
```

```
class Cat extends Pet {
   float weight;
   String colour;

   Cat (String name, int age, char gender, float weight, String colour) {
      super(name, age, gender);
      this.weight = weight;
      this.colour = colour;
   }
```

```
class MyPet {
   public static void main(String[] args) {
      Cat petCat = new Cat("Burbank", 5, 'F', 4.78f, "Grey");
      System.out.println("My cat's name is "+petCat.getName());
      petCat.purr();
   }
}
```

Polymorphism

By Red Dwarf Bodysnatcher DVD, Starbug disc, "Polymorph", https://en.wikipedia.org/w/index.php?curid=15844123

- *Adapting* inherited methods
  - Tailoring sub-class functionality
  - *“One interface, multiple methods”*
- Achieved via *method overriding*
  - Duplicate super-class method names with different functionality
  - Method calls in sub-classes always use the overridden method
  - Super-class method still accessed using `super.method()`

Polymorphism and Method Overriding

```
class Pet {

   public String makeNoise () {
      return "Mmmmmmph";
   }
}
```

```
class Cat extends Pet {
   // overridden from Pet superclass
  public String makeNoise () {
    return super.makeNoise()+" - Meow!";
   }
}
```

```
public class PolyPet {
  public static void main(String[] args) {
  Cat myCat = new Cat("Lucky", 10, 'M', 5.43f, "White");
  Pet myPet = new Pet("Juliette", 11, 'F');

  // use getName and makeNoise method for all pets
  // notice overridden method for myCat
  System.out.println(myCat.getName()+" says \""+myCat.makeNoise()+"\"");
  System.out.println(myPet.getName()+" says \""+myPet.makeNoise()+"\"");
  }
}
```

```
java PolyPet
Lucky says "Mmmmmmph - Meow!"
Juliette says "Mmmmmmph"
```

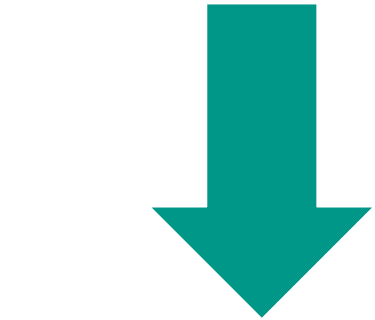
Abstract Classes and Methods

- Abstract *classes* are super-classes never instantiated
  - Outlines attributes and methods, but never used in reality
  - Rather, provides a blueprint or framework for sub-classes
  - Such as Pet class: we instantiate sub-classes: Cat, Dog, Fish, etc.
- Abstract *methods* work in similar ways
  - Only permitted in abstract *classes*
  - *Must* be empty in abstract class
    - `abstract public void myMethod();`
  - Sub-classes *must* implement (override) abstract methods
- Provide further *abstraction* and security in code

```java
abstract public class Character {
  private String name;/
  private int health;

  Character(String n, int h) {
     name=n;
     health = h;
  }

  String getName() {
     return name;
  }
  int getHealth() {
     return health;
  }

  abstract public void move(int x, int y);
}
```

```java
public class MyHero {/
 public static void main(String[] args) {
  Hero hulk = new Hero("Hulk",100,"Mutant");
  hulk.move(5, 2);
  }
}
```

```java
public class Hero extends Character {
  String type;

  Hero (String n, int h, String t) {
     super(n, h);
     type = t;
  }

  public void move(int x, int y) {
     System.out.println("x="+x+" y="+y);
  }
}
```



```
java MyHero
x=5 y=2
```

Containment / Composition

- Including one (or more) object instances *inside* another
- The containing class *has / holds* objects inside it
- We can create useful designs and build our own *syntax*
- As well as class objects containing *multiple* data types
- Further supports *encapsulation*
- For example, using a custom class inside another (e.g. instance variable)

```java
public class Drink {
  String n;
  int v;

  Drink(String name, int volume) {
    n = name;
    v = volume;
  }
}
```

```java
public class Multipack {
  private int qty; //quantity
  private double p; // price
  private Drink d; //drink in pack

  Multipack(int quantity, double price, String name, int volume) {
    qty = quantity;
    p = price;
    d = new Drink(name,volume);
  }

  public int getQuantity() {
    return qty;
  }
  public double getPrice() {
    return p;
  }

  // getters for Drink
  public String getName() {
    return d.n;
  }

  public int getVolume() {
    return d.v;
  }
```

```java
public class DrinkStore {
  public static void main(String[] args) {
    //create instances of Drink Multipacks
    Multipack[] contents = new Multipack[3];
    contents[0] = new Multipack(12, 8.65, "Pepsi", 330);
    contents[1] = new Multipack(10, 12.95, "Monster", 440);
    contents[2] = new Multipack(24, 18.99, "Irn-Bru", 330);

    // output the contents of the DrinkStore
    // all of the multipacks created
    for(int i=0; i<contents.length; i++) {
      System.out.println("Mulipack "+i+" contents");
      System.out.println(contents[i].getName());
      System.out.println(contents[i].getQuantity()+" x "+contents[i].getVolume()+"ml cans");
      System.out.println("\u00A3"+contents[i].getPrice());
      System.out.println();
    }
  }
}
```

Recursion

- A method calling *itself*
  - A circular definition
- Each version ‘updates’ itself
- Including an `if` statement is essential or it may never end
- *Elegant* but not always *efficient*

Recursion (Cutajar 2018)

Recursion is a really useful tool for algorithm designers. It allows you to solve large problems by solving a smaller occurrence of the same problem. Recursive functions usually have a common structure with the following components:

- *One or more stopping conditions : Under certain conditions, it would stop the function from calling itself again*
- *One or more recursive calls : This is when a function (or method) calls itself*

Calculating b^e

```java
static double powerLoop(int b, int e) {
    double ans = 1;
    for (int i=0; i<e; i++) {
        ans = ans*b;
    }
    return ans;
}
```

Iterative solution

```java
static double powerRecursive(int b, int e) {
    double ans;
    if (e==0) {
        ans=1;
        return ans;
    }
    ans = powerRecursive(b, e-1)*b;
    return ans;
}
```

Recursive solution

References

Cutajar, J. (2018). *Beginning Java Data Structures and Algorithms*. Packt Publishing.

Schildt, H. (2022). *Java: a beginner's guide* (9th ed.). McGraw Hill.