Object Oriented Programming with Java

Inheritance

What problem are we trying to solve?

National Zoo is building an Object Oriented program to manage its animal inventory

- about 1800 animals

- of 300 different species

How can we model this?

image source: www.schoolspecialty.com

How to model the 300 species of the National Zoo?

Approach 1: create a general class for all species

```
class Animal {
   int id;
   String name;
   float weight;
   int legs;
   boolean carnivore;
   boolean amphibian;
   ...

   void walk() {...}
   void run() {...}
   void fly() {...}
   void crawl() {...}
   void swim() {...}
   ...
}
```

- ✗ Redundant attributes and methods in order to cover all possible species
- ✗ Requires too many conditions for validating and using instances
- ✗ Extremely difficult to maintain the code; every change affects many other classes
- ✗ Violates the OO philosophy; it's practically a procedural approach

How to model the 300 species of the National Zoo?

Approach 2: create a separate class for each species

```
class Dog {
    String name;
    boolean sniffer;
    …
    void eat() {…}
    void bark() {…}
    …
}
```

```
class Cat {
    String name;
    int mouse_counter;
    …
    void eat() {…}
    void meow() {…}
    …
}
```

… …

```
class Cow {
    String name;
    float milk_production;
    …
    void eat() {…}
    void moo() {…}
    …
}
```

- ✗ Code repetition; doesn't take advantage of the overlap in attributes and methods
- ✗ Extremely difficult to maintain the code; every minor change modifies all classes

and most important...

How to model the 300 species of the National Zoo?

Approach 2: create a separate class for each species

```
class Dog {...}
class Cat {...}
class Cow {...}
...

class Analytics {
   float average_age(Dog[] v) {...}
   float average_age(Cat[] v) {...}
   float average_age(Cow[] v) {...}
   ...
}
```

✗ Requires a tedious overloading of functions

If we build a separate class for each of the 300 species in the National Zoo, we would need 300 implementations of the method `average_age()` in this example

How to model the 300 species of the National Zoo?

Both approaches are inefficient and cause many side effects
It would be nice if we had a structure that mimics taxonomy

- ✔ Tree structure
- ✔ Child node has parent features
- ✔ Child expands on parent features
- ✔ Siblings share common features
- ✔ Exploit *child-like-parent* relationship

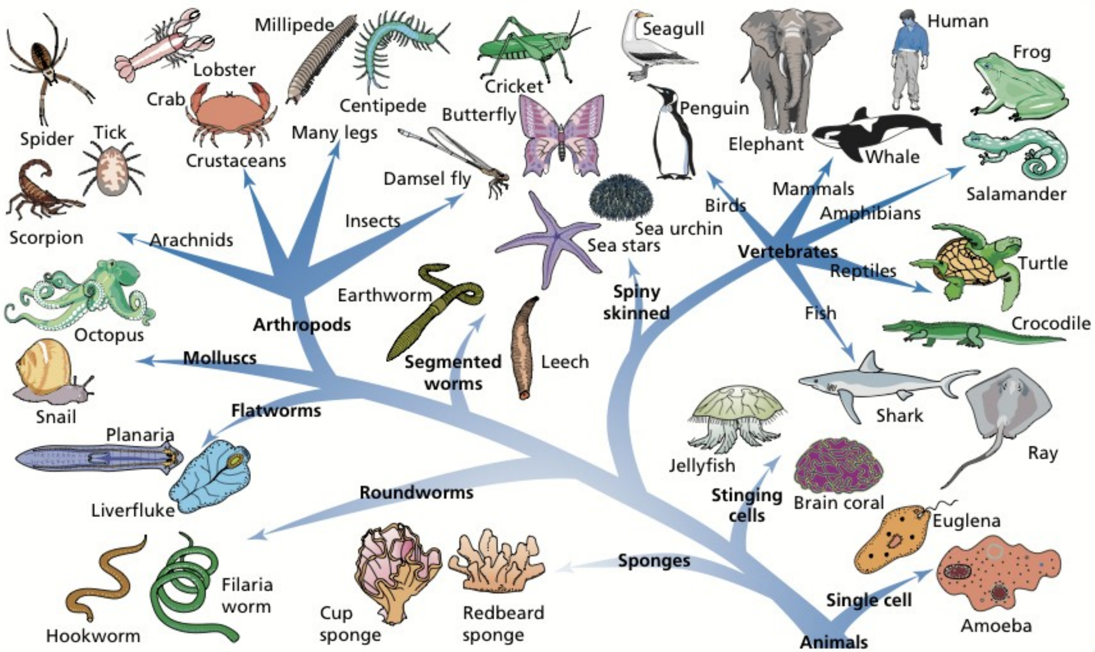

image source: cdn.thinglink.me

How to model the 300 species of the National Zoo?

Approach 3: organize classes in a tree structure

```
class Animal {
   String name;
   …
   void eat() {…}
   …
}
```

Ideally we would like to have a means to define such a linkage between classes:

All animals have a **name** and **eat()**, but each species has features added to the inherited ones

```
class Dog {
   boolean sniffer;
   …
   void bark() {…}
   …
}
```

```
class Cat {
   int mouse_counter;
   …
   void meow() {…}
   …
}
```

… …

```
class Cow {
   float milk_production;
   …
   void moo() {…}
   …
}
```

How to model the 300 species of the National Zoo?

Approach 3: object oriented inheritance

```
class Animal {
   String name;
   …
   void eat() {…}
   …
}
```

Java offers inheritance with keyword **extends**

Simply append statement **extends <class_name>** to the definition of the derived class

```
class Dog extends Animal {
   boolean sniffer;
   …
   void bark() {…}
   …
}
```

```
class Cat extends Animal {
   int mouse_counter;
   …
   void meow() {…}
   …
}
```

… …

```
class Cow extends Animal {
   float milk_production;
   …
   void moo() {…}
   …
}
```

Inheritance benefits

- **Clarity** – code is easier to read, debug and maintain in general
- **Reuse** – it's easier to make updates and/or scale the application without repeating implementation and testing
- **Efficiency** – it eliminates tedious redundancies in the code and reduces the memory footprint
- **Versatility** – code can be plugged into different functions by taking advantage of the parent-child relationships
- **Subtyping** – it offers a mechanism for type polymorphism

Types of inheritance supported in Java

Single

```java
class A {
    int a;
    void f1()
}
```

```java
class B extends A {
    int b;
    void f2()
}
```

Multi-level

```java
class A {
    int a;
    void f1()
}
```

```java
class B extends A {
    int b;
    void f2()
}
```

```java
class C extends B {
    int c;
    void f3()
}
```

Hierarchical

```java
class A {
    int a;
    void f1()
}
```

```java
class B extends A {
    int b;
    void f2()
}
```

```java
class C extends A {
    int c;
    void f3()
}
```

```java
class D extends B {
    int d;
    void f4()
}
```

```java
class E extends B {
    int e;
    void f5()
}
```

Types of inheritance NOT allowed in Java

Multiple

```
class A {
  int a;
  void f1()
}
```

```
class B {
  int b;
  void f2()
}
```

```
class C extends A, B {
  int c;
  void f3()
}
```

Circular

```
class A extends C {
  int a;
  void f1()
}
```

```
class C extends B {
  int c;
  void f3()
}
```

```
class B extends A {
  int b;
  void f2()
}
```

- A class cannot extend more than **one class** (*interfaces* offer a workaround)
- There can be **no cycles** in the graph (in any language, not just Java)

Inheritance basics

```java
class Base {
   int a;
   void func1() {…}
}
```

```java
class Derived extends Base {
   int b;
   void func2() {…}
}
```

Terminology: Base / Parent / Super class
vs
Derived / Child / Sub class

All members of Base class (except the constructor) are automatically becoming members of Derived class without explicitly declaring them again, e.g.

```java
Derived obj = new Derived();
System.out.println(obj.a+obj.b);
obj.func1();
```

Child class can modify the inherited members but cannot remove any

Constructors are **not** inherited but can be invoked

Java Class Hierarchy

- The **Object** class is the root of all classes. It defines and implements behavior common to all classes, including the ones that you write
- Many Java classes derive directly from *Object*, other classes derive from some of those classes, and so on, forming a hierarchy of classes
- If a class does not explicitly have a parent class, it will implicitly derive from *Object*

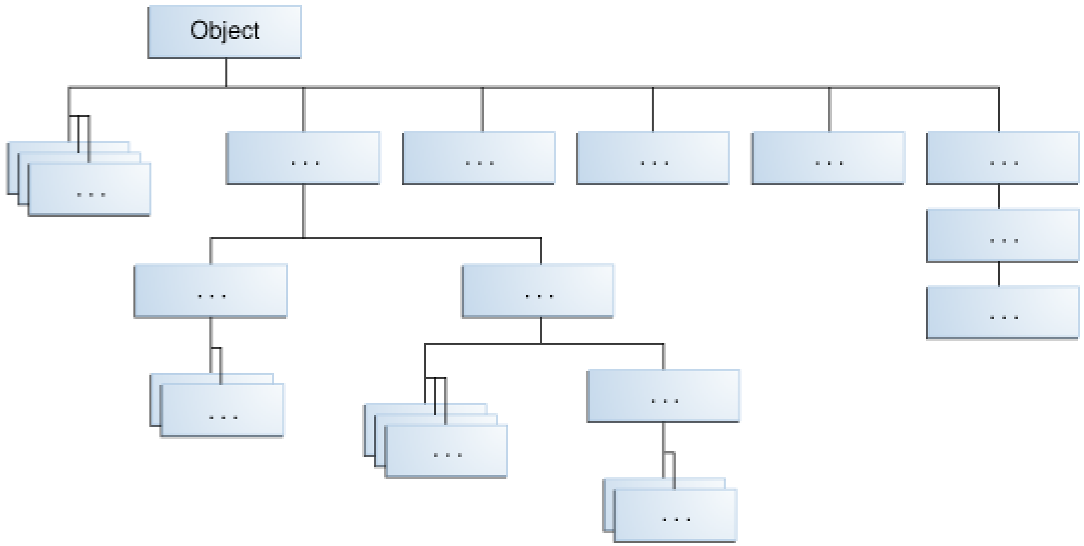

image source: docs.oracle.com

Constructor

```
class Parent
{
   Parent()
   {
      System.out. println(“Parent constructor”);
   }
}

class Child extends Parent
{
   Child()
   {
      super();
      System.out. println(“Child constructor”);
   }
}
```

- First statement of the Child constructor must be a call to Parent constructor by using keyword **super**
- *super* can call both parametric and non parametric constructors
- If the Child constructor does not explicitly invoke the Parent constructor, the Java compiler automatically inserts a call to a non parametric constructor of the Parent
- If Parent doesn’t have a non parametric constructor and we omit **super** in the Child constructor, we will get a compile-time error

Constructor chaining
The Child constructor invokes, either explicitly or implicitly, the Parent constructor which invokes its own superclass constructor, and the process repeats all the way back to the constructor of *Object*

Constructor examples

1

```java
class Parent {
  Parent() {…}
}

class Child extends Parent {
  Child(int num) {
    super();
  }
}
```

2

```java
class Parent {
  Parent() {…}
}

class Child extends Parent {
  Child(int num) {
  }
}
```

3

```java
class Parent {
  Parent(int var) {…}
}

class Child extends Parent {
  Child(int num) {
    super(num);
  }
}
```

4

```java
class Parent {
  Parent(int var) {…}
}

class Child extends Parent {
  Child(int num, int id) {
    super(id);
  }
}
```

5

```java
class Parent {
  Parent() {…}
}

class Child extends Parent {
  Child(int num) {
    int a = 5;
    super();
  }
}
```

6

```java
class Parent {
  Parent(int var) {…}
}

class Child extends Parent {
  Child(int num) {
  }
}
```

7

```java
class Parent {
  Parent(float var) {…}
}

class Child extends Parent {
  Child(int num) {
    super(num);
  }
}
```

8

```java
class Parent {
  Parent(int var) {…}
}

class Child extends Parent {
  Child(int num, int id) {
    super(num, id);
  }
}
```

Is-a relationship

```
class Animal {
   String name;
   void eat() {…}
}

class Mammal extends Animal {
   boolean gender;
   void nursing() {…}
}

class Dog extends Mammal {
   boolean sniffer;
   void bark() {…}
}
```

Inheritance is **unidirectional** and creates an ***is-a*** relationship

Mammal extends Animal → Mammal ***is-a*** Animal

Inheritance is also **transitive**

Dog extends Mammal extends Animal → Dog ***is-a*** Animal

Inheritance does **not** work the opposite direction though

~~Animal ***is-a*** Mammal~~
~~Mammal ***is-a*** Dog~~
~~Animal ***is-a*** Dog~~

How do we take advantage of ***is-a*** relationship in practice?

Is-a relationship in practice

Let's go back to our National Zoo problem...

```
class Dog {…}
class Cat {…}
class Cow {…}
…

class Analytics {
   float average_age(Dog[] v) {…}
   float average_age(Cat[] v) {…}
   float average_age(Cow[] v) {…}
   …
}
```

```
class Animal {
   int age;
}
class Dog extends Animal {…}
class Cat extends Animal {…}
class Cow extends Animal {…}
…

class Analytics {
   float average_age(Animal[] v) {…}
}
```

```
public static void main(String[] args) {
   Animal data[] = {new Dog(), new Cat(), new Cow()};
   Analytics an = new Analytics();
   System.out.println(an.average_age(data));
}
```

Visibility of classes

- modifiers are optional keywords that are prepended to the definition of classes 1, attributes 2 and methods 3 in order to control their visibility across the program
- the outer class modifier can be either the keyword **public** or no modifier
- class member modifiers can be one of the following keywords: **public**, **private**, **protected** or no modifier
- classes with no modifier are visible everywhere inside the package (a.k.a. ***package-private***)
- *public* classes are visible everywhere in the program

```
1 class Dog {

  2 int id;

  3 void walk() {…}

}
```

examples →

```
class Dog {

  protected int id;

  void walk() {…}

}
```

```
public class Dog {

  private int id;

  public void walk() {…}

}
```

Visibility of class members

- class members with no modifier are visible everywhere inside the package (a.k.a. ***package-private***)
- ***public*** class members are visible everywhere in the program
- ***private*** class members are not visible anywhere outside the class
- ***protected*** class members are visible everywhere in the same package as well as inside subclasses in other packages (different from C++)

	no modifier	**private**	**protected**	**public**
same class	yes	yes	yes	yes
subclass in same package	yes	no	yes	yes
non-subclass in same package	yes	no	yes*	yes
subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

Special cases of visibility

A subclass does not inherit the private members of its superclass. But if the parent has public or protected methods for accessing the private variables, the child can use them to indirectly access the private fields

```
class A {
   private int a;
   public A(int a) {
      this.a = a;
   }
   protected int getA() {
      return a;
   }
}

class B extends A {
   public B(int b) {
      super(b);
   }
   public int getB() {
      return getA();
   }
}
```

```
class OuterClass {
   private int a=10;
   class NestedClass {
      int getPrivate() {return a;}
   }
}

class Sub extends OuterClass {
   int getA(){
      return new NestedClass().getPrivate();
   }

   public static void main(String[] args) {
      Sub s = new Sub();
      System.out.println(s.getA());
   }
}
```

A non-static nested class has access to all the private members of its enclosing class. Therefore, a public or protected non-static nested class inherited by a subclass has indirect access to all of the private members of the superclass.

Tips on using access modifiers

- Use the most restrictive access level that makes sense for a particular member
- In most cases you should **use private** unless you have a good reason to do otherwise
- **Avoid public fields** except for constants. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code

Method overriding

What's method overriding?
When both the subclass and the superclass have a method with exactly the same signature but a different body

Why overriding?
Since the subclass is a specialization of the superclass we usually want to customize some of the methods and create more tailored versions of them

Are there any conditions?
The only requirement in order for a subclass to override a method is that the method has not been declared as **final** in the superclass. Similarly to what happens to fields, the *final* modifier doesn't allow any modifications to a method

```
public final double get_Avogadro_constant() {
  return 6.02214076*Math.pow(10,23) ;
}
```

Method overriding example

Which version of eat() will be invoked?

It depends on the type of the object instantiation, not the type of the variable declaration

```
class Animal {
   String name;
   void eat() {
      System.out.println("Eating like animal");
   }
}

class Dog extends Animal {
   boolean sniffer;
   void eat() {
      System.out.println("Eating like a dog");
   }
}
```

output

```
Animal a = new Animal();
Dog d = new Dog();
a.eat();
d.eat();
```

```
Eating like animal
Eating like a dog
```

```
Animal a = new Animal();
Animal d = new Dog();
a.eat();
d.eat();
```

?

```
Dog a = new Animal();
Dog d = new Dog();
a.eat();
d.eat();
```

?

Dynamic dispatch

In cases of polymorphism (e.g. **`obj`**) combined with method overriding (e.g. **`func`**), the compiler cannot resolve the method call **`obj.func()`** statically at compile time. Dynamic dispatch is the mechanism that resolves the method invocation at <u>run time</u>.

```
class Parent {
     void func(){}
}

class Child extends Parent {
     void func(){}
}

class App {
     public static void main(String[] args) {
          Parent obj;
          if (scanner.nextInt() == 0)
               obj = new Parent();
          else
               obj = new Child();
          obj.func();
     }
}
```

Use of super in overriding

```
class Animal {
  int age=0;
  void eat() {
    System.out.println("Eating like animal");
  }
}

class Dog extends Animal {
  int age=10;
  void eat() {
    System.out.println("Eating like a dog");
  }

  void display() {
    eat();
    super.eat();
    System.out.println(age);
    System.out.println(super.age);
  }

  public static void main(String[] args) {
    Dog d = new Dog();
    d.display();
  }
}
```

- If a method of a subclass wants to explicitly invoke an overridden method of the superclass instead of its own overriding method, the keyword **`super`** can help resolve the ambiguity
- To override fields instead of methods is called **shadowing** and is generally <u>not</u> recommended. In case you do, though, keyword **`super`** can help resolve this kind of ambiguity too

output

```
Eating like a dog
Eating like animal
10
0
```

Use of super in practice

```java
class Animal
{
    String eat()
    {
        return("I eat like an animal");
    }
}

class Dog extends Animal
{
    String eat()
    {
        return(super.eat()+" and I bark like a dog");
    }
}
```

Usually, this is how a method is using its parent's overridden method

It **incorporates** and **expands** the overridden method instead of ~~rewriting any code~~

Abstract class

What's an Abstract class?
An abstract class is a partially defined class that cannot be instantiated but can be inherited. It usually defines some abstract (empty) methods that the inheriting subclass must implement.

Why is this useful?
The purpose of an abstract class is to define some common behavior that will be inherited and implemented by subclasses. It's like a blueprint for the derived classes in order for all of them to follow the same protocol.

How to define it?
Simply add keyword **abstract** in the definition of the class

```
public abstract class WalkingAnimal {…}
```

Abstract class example

```java
public abstract class WalkingAnimal {
  private int num_of_legs;

  abstract public void walk();

  final public int getNumLegs() {
    return num_of_legs;
  }

  public void eat() {
    System.out.println("Eating like animal");
  }
}

public class Dog extends WalkingAnimal {
  Dog() {
    eat();
  }
  public void walk() {
    System.out.println("I can walk!");
  }
}
```

- Abstract classes may contain abstract methods, e.g. walk(). Note that the definition of an abstract method doesn't have a body because it <u>must</u> be implemented by the subclass.
- Concrete classes <u>cannot</u> contain abstract methods because object instantiation would fail
- Abstract classes may contain regular methods, e.g. eat(), that a subclass can directly use or override.
- A *final* or *static* method cannot be *abstract* because a final cannot be modified in a subclass, and a static doesn't belong to any instance
- A subclass may opt to not implement an abstract method and leave it for a descendant. But in this case the subclass cannot be instantiated either.

final keyword revisited

Final variable → create constant variables

You must initialize a final variable. If the final variable is a reference, this means it cannot be re-bound to reference another object

Final method → prevent method overriding

Final class → prevent inheritance / make class immutable

How to design inheritance in practice

Designing the hierarchy of classes for a certain application is not an easy task. There is not a unique organization of the attributes and the behaviors and it's hard to prove which one is more efficient; let alone that efficiency depends on the data you model as well as the application's requirements that may evolve over time too.

eat prey jump leg fin

neck

feed trunk

tail

climb

walk fly swim antenna

fur

crawl hand

run

nurse

How to design inheritance in practice

Rule of thumb

- Examine all *species*, identify common attributes and behaviors, wrap them in a class that is put as high in the hierarchy as possible
- Take the remaining attributes/behaviors, identify maximal sets of common attributes/behaviors, and create subclasses and/or abstract classes
- Reuse code and override methods as much as possible. Avoid shadowing
- Make use of visibility modifiers. Do not abuse the *public* keyword
- Always think one step ahead… what modification the next update will require?