Middlebury

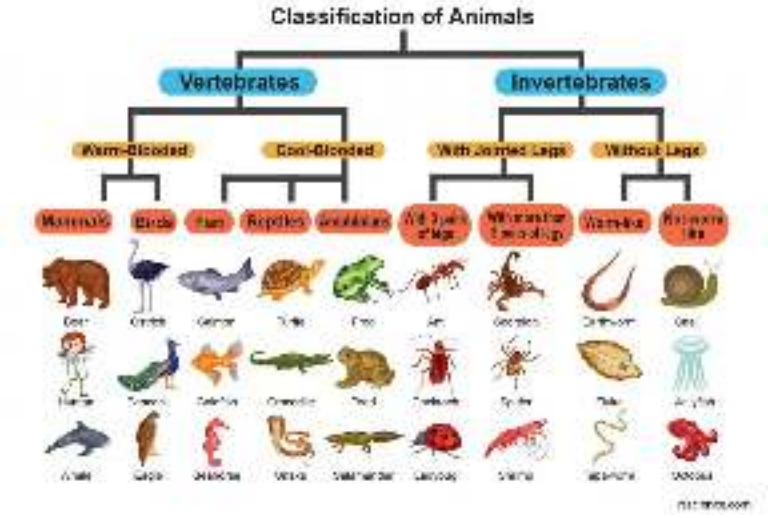
CSCI 201: Data Structures

Fall 2024

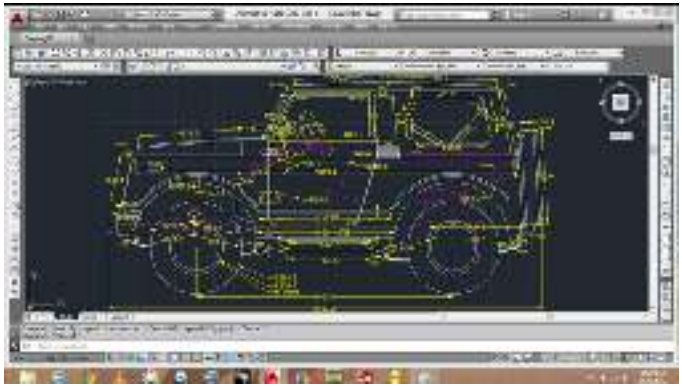
Lecture 3T: Polymorphism

Goals for today:

- **Derive** (inherit) child/subclasses from a **parent/base/superclass** using `extends`.
- Save references to a base class in an array.
- Use the `protected` access modifier to limit access to fields/methods.
- Call the `super` class constructor to initialize the base object.
- Introduce `package`s.
- Use **generics** to define **parametrized** classes.



Last week we created a class called `Car`.

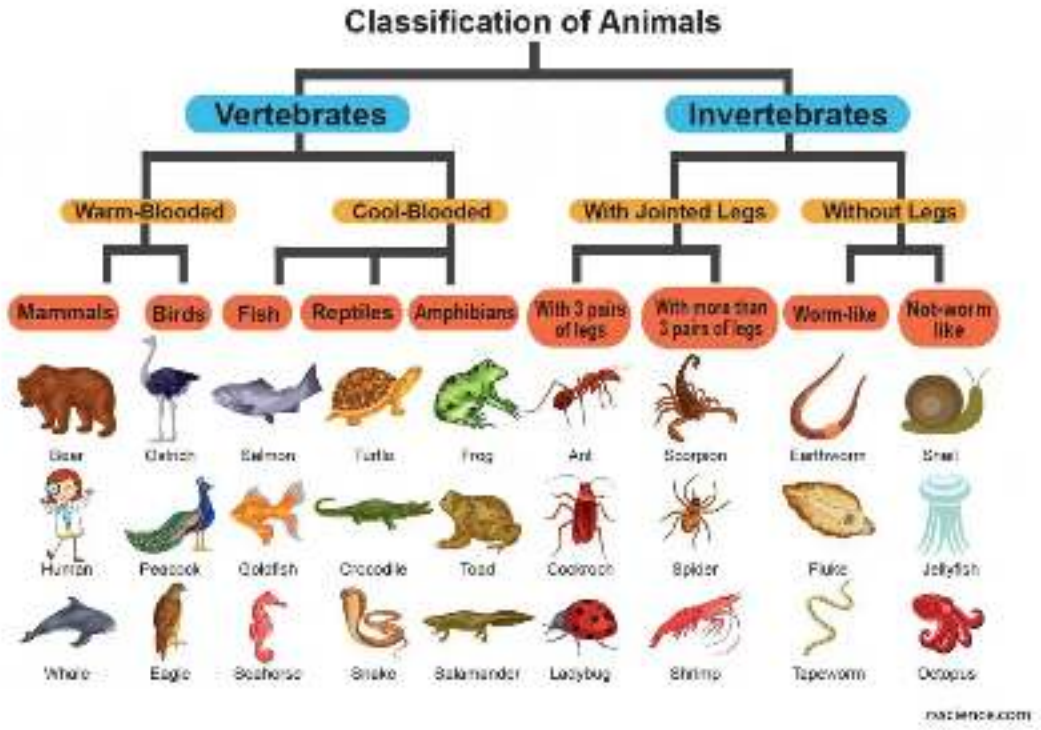


```
1 class Car {
2   ...
3 }
```

```
1 Car car = new Car("Subaru", 2019);
```

What kinds of cars are there?

What if we were designing a class called `Animal`?



This implies that some applications would benefit from *specializing* our class definitions.

- Some methods are shared between all car/animal types.
- Some fields/methods are special for different car/animal types.

How can we achieve these?

Polymorphism (use of a single interface to create many types).

Two types of polymorphism to consider:

- **Run-time polymorphism:** types are decided at run-time (running `java`).
- **Compile-time polymorphism:** types are decided at compile-time (with `javac`).

Run-time polymorphism using *inheritance*.

```java
package animals;

public class Animal {
  protected int numLegs;

  // constructor
  public Animal(int n) {
    numLegs = n;
  }

  public void speak() {
    System.out.println("Hello World);
    System.out.println("# legs = " + numLegs);
  }
}
```

```java
package animals;

public class Dog extends Animal {
  public Dog(int n) {
    super(4); // call the Animal constructor
    // calling super is not necessary if the
    // superclass has a constructor with no
    // arguments
  }

  // @Override is optional, but good practice
  @Override
  public void speak() {
    System.out.println("Woof World");
    System.out.println("# legs = " + numLegs);
  }
}
```

- `Animal` is a **superclass** (base/parent class).
- `Dog` is a **subclass** (derived/child class).
- Subclass can access any `public`/`protected` fields/methods of superclass.
- Methods defined in subclass with the same signature as superclass will be **overriden**.
- Superclass needs to be constructed when subclass is constructed.
- `Java` allows you to inherit from **one class** (otherwise, you inherit from `Object`).
- **Note:** a `package` is used here to keep all animals in one subfolder.

Note that we can save an array of references to the *base* type.

```java
// import all classes from the animals subfolder
import animals.*;

public class InheritanceExamples {

  public static void main(String[] args) {

    Animal animal = new Animal(0);
    animal.speak();

    int numAnimals = 5;
    Animal[] animals = new Animal[numAnimals];
    animals[0] = new Dog();
    animals[1] = new Cat();
    animals[2] = new Sheep();
    animals[3] = new Dog();
    animals[4] = new Penguin();
    // ^ these are all references to objects with
    // the type SUPERCLASS

    for (Animal a : animals) {
      // speak is overriden in the SUBCLASS
      a.speak();
    }
  }
}
```

```
Hello World, I have 0 legs
Woof World, I have 4 legs
Meow World, I have 4 legs
Baaa world, I have 4 legs
Woof World, I have 4 legs
Chirp World, I have 2 legs
```

Exercise: define your own animal that inherits from the `Animal` class.

```
1  package animals;
2
3  public class Cow extends Animal {
4    public Cow(int n) {
5      super(4);
6    }
7
8    @Override
9    public void speak() {
10     System.out.println("Moo World, I have " + numLegs + " legs");
11   }
12 }
```

Investigate!

- Can you access the `numLegs` field in the `InheritanceExamples` PSVM?
- What happens if we make the `Animal` constructor `protected`?
- What happens if you don't override the `speak` method?
- What happens if `numLegs` was declared `private` in the `Animal` superclass? What would you need to do to access `numLegs` in the `speak` method of the subclasses in this case?

```java
package animals;

public class Animal {
  private int numLegs;

  public Animal(int n) {
    numLegs = n;
  }

  protected int numLegs() {
    return numLegs;
  }
}
```

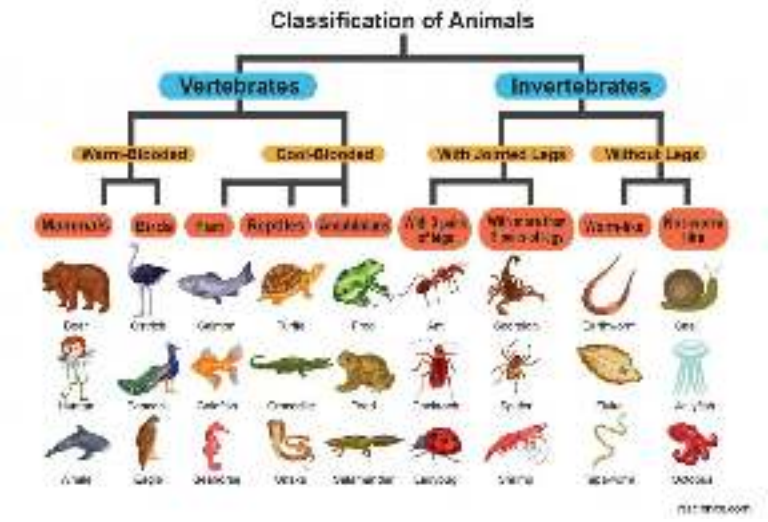
```java
package animals;

public class Cow extends Animal {
  public Cow(int n) {
    super(4);
  }

  @Override
  public void speak() {
    System.out.println("Moo World");
    System.out.println("# legs = " + numLegs());
  }
}
```

Where are we in our goals for today?

- **Derive** (inherit) child/subclasses from a **parent/base/superclass** using `extends`.
- Save references to a base class in an array.
- Use the `protected` access modifier to limit access to fields/methods.
- Call the `super` class constructor to initialize the base object.
- Introduce `package`s.
- Use **generics** to define **parametrized** classes.



What if we want to design some kind of container, but hold *anything* in that container?

Parametric polymorphism using *generics* (checked at compile-time).

Motivation: imagine we want to create a `Box` class.
Boxes can hold anything.

```
1  class Box {
2    public boolean empty() {
3      return false; // to be overriden
4    }
5  }
6
7  class FrootLoops {}
8  class FrootLoopsBox extends Box {
9    FrootLoops cereal;
10   FrootLoopsBox(FrootLoops cereal) {
11     this.cereal = cereal;
12   }
13
14   public boolean empty() {
15     return cereal == null;
16   }
17 }
```

- What if we had another cereal `HoneyNutCheerios`? Create another `HoneyNutCheeriosBox` that inherits from `Box` again?
- The class design on the left makes this process cumbersome.

Parametric polymorphism using *generics* (checked at compile-time).

```
1  class Box<T> {
2    T cereal;
3    Box(T cereal) {
4      this.cereal = cereal;
5    }
6    public boolean empty() {
7      return cereal == null;
8    }
9  }
10
11 class FrootLoops {}
12 class HoneyNutCheerios {}
13
14 public class GenericsExample {
15   public static void main(String[] args) {
16     Box<FrootLoops> loops =
17       new Box<FrootLoops>(new FrootLoops());
18     Box<HoneyNutCheerios> cheerios =
19       new Box<HoneyNutCheerios>(new HoneyNutCheerios());
20     ...
21   }
22 }
```

- Instead, we can use *generics*, which allows us to parametrize our classes in terms of some *type*.
- Allows us to define one interface to be instantiated with specialized types.
- Useful for things like containers (next class).

Compiler will check if we're using the types correctly. For example:

```
Box<HoneyNutCheerios> cheerios = new Box<FrootLoops>(new FrootLoops());
```

will not compile!

Other notes and conventions with generics.

```java
class Box<T> {
  T item;
}

public class BoxExample {
  public static void main(String args[]) {
    // newer versions of Java allow us to do this
    // saves us a bit of typing
    Box<FrootLoops> frootLoops = new Box<>(new FrootLoops());
  }
}
```

- `T` for a **type**.
- `E` for an **element**.
- `K` for a **key**.
- `V` for a **value**.
- Generics are useful at compile-time, but type information is thrown away and not available at run-time (called *type erasure*).
- We can also make sure the type `T` is a subclass of some type (next slide).

Exercise: add a `toString()` method for the `Box` class to print out cereal label information.

Get started with this:

```java
class Cereal {
  private String name;
  public String[] ingredients; // look this up
  public int sugarPerServing; // in grams
  public double servingSize; // in cups
  Cereal(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}

class FrootLoops extends Cereal {
  FrootLoops() {
    super("Froot Loops");
    // TODO save Cereal fields here
  }
}
```

```java
class Box<T extends Cereal> {
  T cereal;

  String toString() {
    // TODO print label with name,
    // ingredients,
    // sugarPerServing and servingSize
  }
}

public class GenericsExample {
  public static void main(String args[]) {
    Box<FrootLoops> frootLoops =
              new Box<>(new FrootLoops());
    System.out.println(frootLoops);
  }
}
```

See you on Thursday!

- We'll introduce `Collections` which use generics.
- Get started on Homework 2: due 9/26 at 11:59pm.
- Office hours posted:
  - Monday 9:30 - 10:30am
  - Tuesday: 11 - 11:30am
  - Thursday: 1:30 - 3:30pm
  - Friday: 2 - 2:30pm
- Submit exit ticket 3T today.