



Middlebury

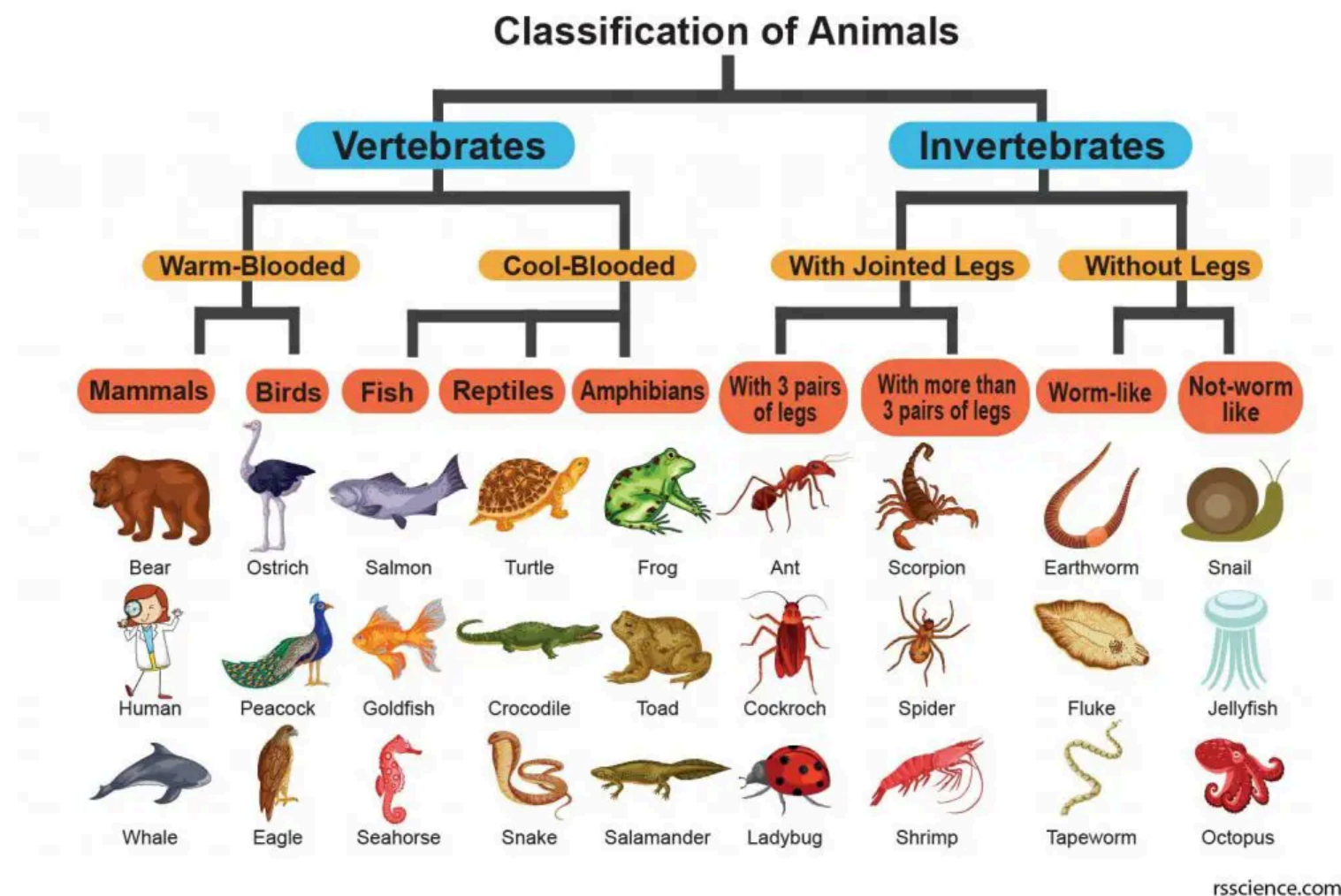
CSCI 201: Data Structures

Spring 2025

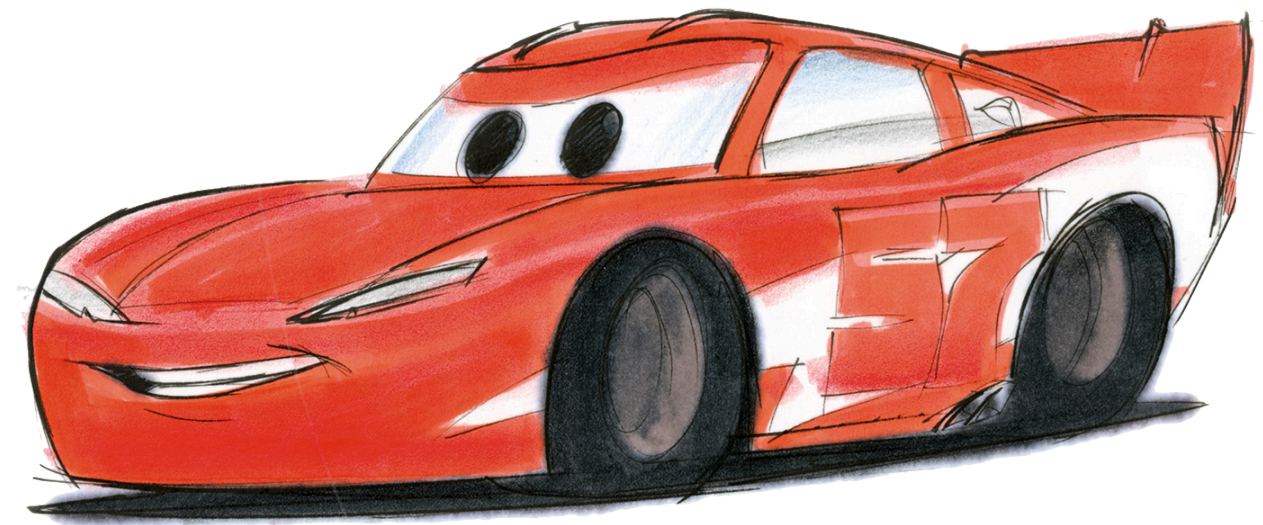
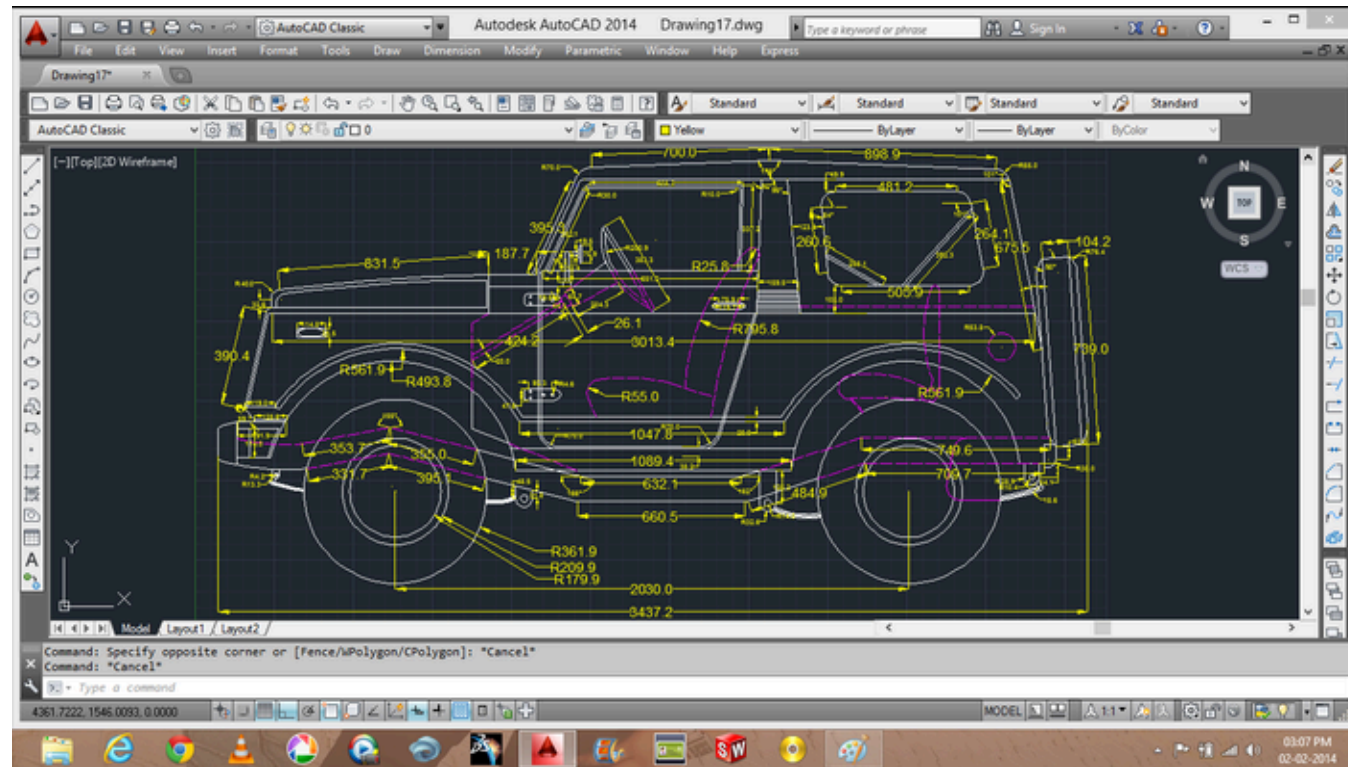
Lecture 3M: Polymorphism

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 **packages**.
- Use **generics** to define parametrized classes.



Last week we created a class called **Car**.

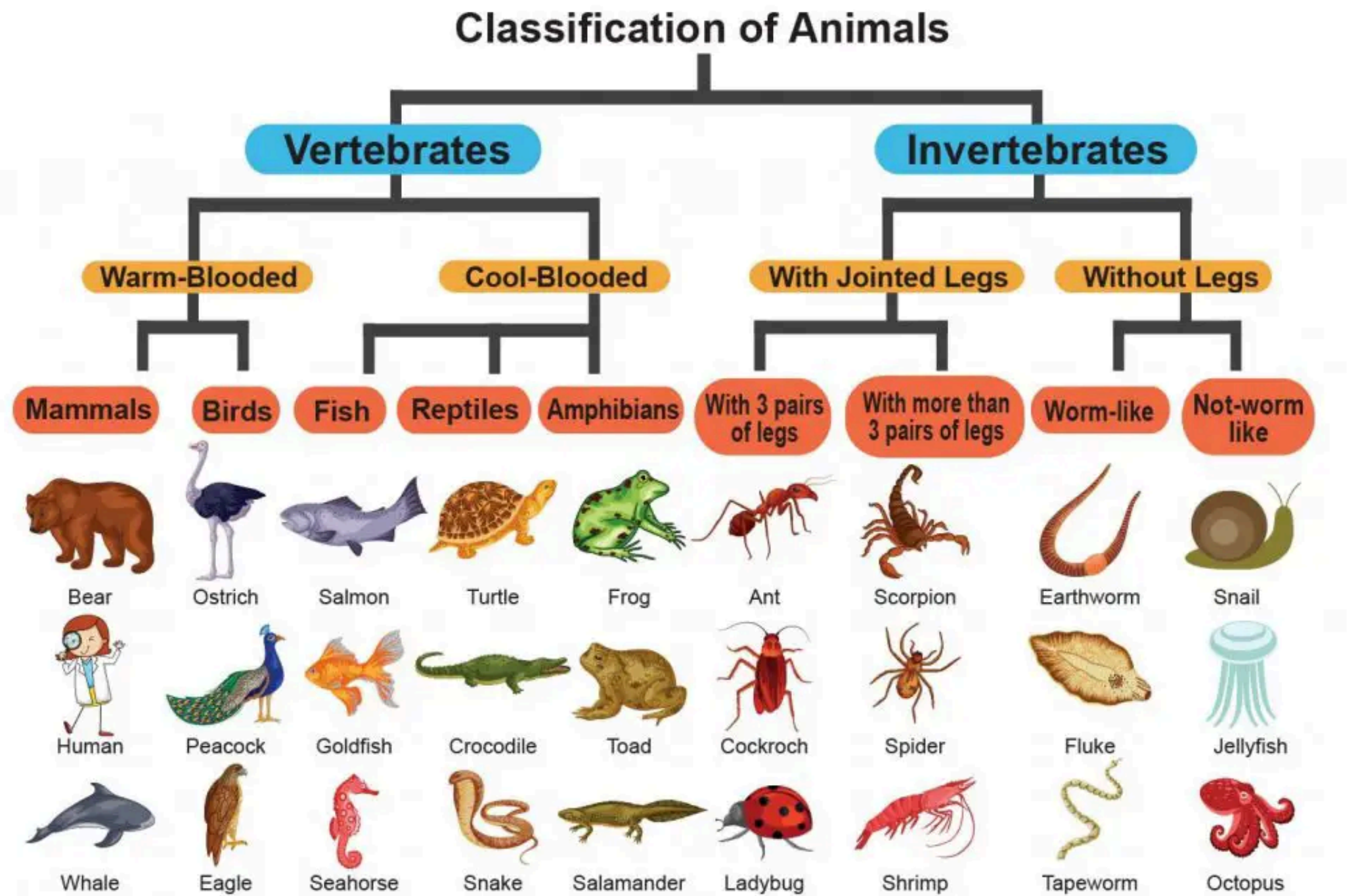


```
1 class Car {  
2     ...  
3 }
```

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

What kinds of cars are there?

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



rsscience.com

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*.

```
1 package animals;
2
3 public class Animal {
4     protected int numLegs;
5
6     // constructor
7     public Animal(int n) {
8         numLegs = n;
9     }
10
11    public void speak() {
12        System.out.println("Hello World");
13        System.out.println("# legs = " + numLegs);
14    }
15 }
```

```
1 package animals;
2
3 public class Dog extends Animal {
4     public Dog(int n) {
5         super(4); // call the Animal constructor
6         // calling super is not necessary if the
7         // superclass has a constructor with no
8         // arguments
9     }
10
11    // @Override is optional, but good practice
12    @Override
13    public void speak() {
14        System.out.println("Woof World");
15        System.out.println("# legs = " + numLegs);
16    }
17 }
```

- **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 **overrides**.
- 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*.

```
1 // import all classes from the animals subfolder
2 import animals.*;
3
4 public class InheritanceExamples {
5
6     public static void main(String[] args) {
7
8         Animal animal = new Animal(0);
9         animal.speak();
10
11         int numAnimals = 5;
12         Animal[] animals = new Animal[numAnimals];
13         animals[0] = new Dog();
14         animals[1] = new Cat();
15         animals[2] = new Sheep();
16         animals[3] = new Dog();
17         animals[4] = new Penguin();
18         // ^ these are all references to objects with
19         // the type SUPERCLASS
20
21         for (Animal a : animals) {
22             // speak is overridden in the SUBCLASS
23             a.speak();
24         }
25     }
26 }
```

```
1 Hello World, I have 0 legs
2 Woof World, I have 4 legs
3 Meow World, I have 4 legs
4 Baaa world, I have 4 legs
5 Woof World, I have 4 legs
6 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!

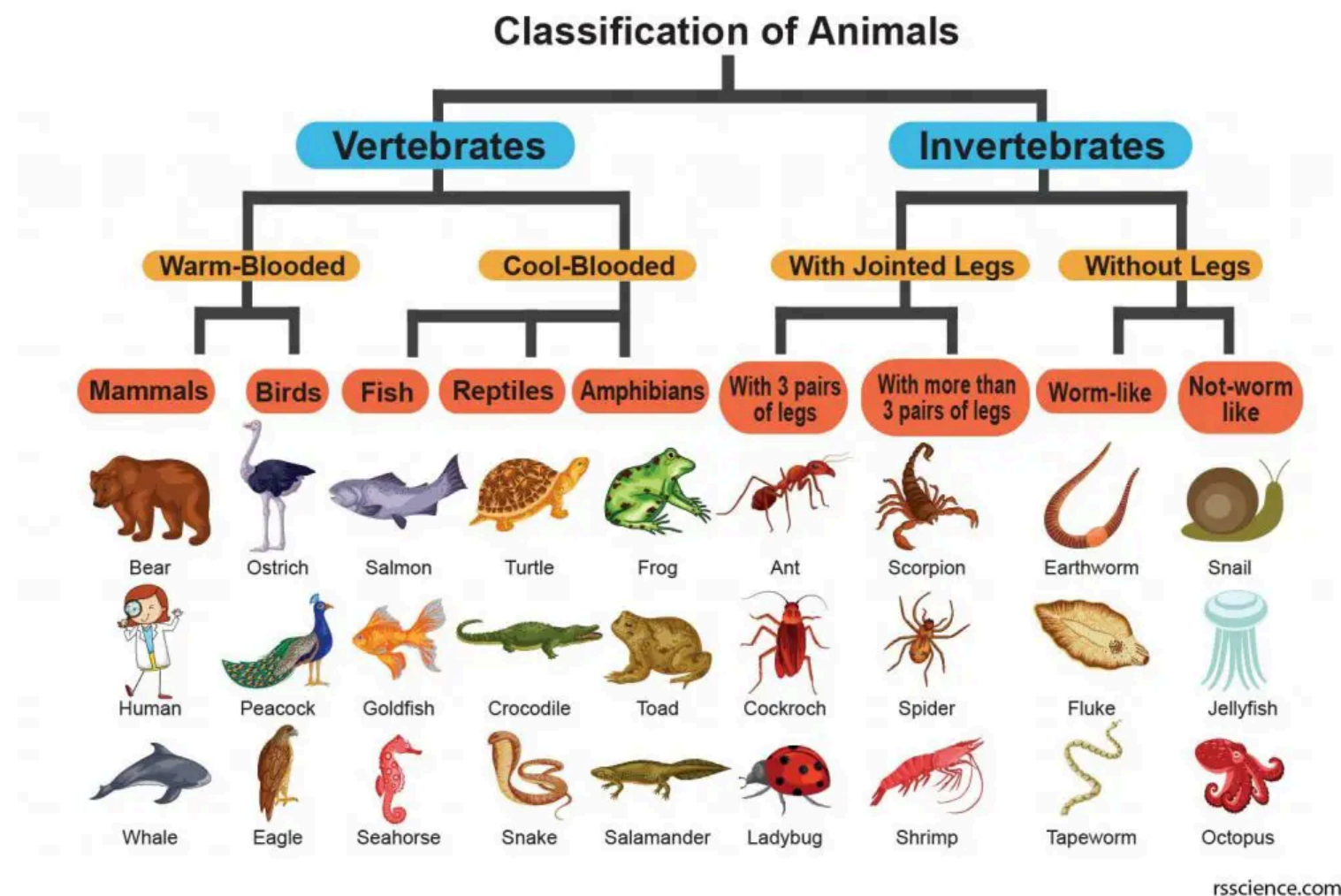
- Can you access the `numLegs` field in the `InheritanceExamples` PSVM?
- What happens if we make the `Animal` constructor `protected`?
- What happens if you don't override the `speak` method?
- What happens if `numLegs` was declared `private` in the `Animal` superclass?
What would you need to do to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

```
1 package animals;
2
3 public class Animal {
4     private int numLegs;
5
6     public Animal(int n) {
7         numLegs = n;
8     }
9
10    protected int numLegs() {
11        return numLegs;
12    }
13 }
```

```
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");
11        System.out.println("# legs = " + numLegs());
12    }
13 }
```

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 **packages**.
- 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 overridden
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.

```
1 class Box<T> {
2     T item;
3 }
4
5 public class BoxExample {
6     public static void main(String args[]) {
7         // newer versions of Java allow us to do this
8         // saves us a bit of typing
9         Box<FrootLoops> frootLoops = new Box<>(new FrootLoops());
10    }
11 }
```

- **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:

```
1 class Cereal {
2     private String name;
3     public String[] ingredients; // look this up
4     public int sugarPerServing; // in grams
5     public double servingSize; // in cups
6     Cereal(String name) {
7         this.name = name;
8     }
9
10    public String getName() {
11        return name;
12    }
13 }
14
15 class FrootLoops extends Cereal {
16     FrootLoops() {
17         super("Froot Loops");
18         // TODO save Cereal fields here
19     }
20 }
```

```
1 class Box<T extends Cereal> {
2     T cereal;
3
4     String toString() {
5         // TODO print label with name,
6         // ingredients,
7         // sugarPerServing and servingSize
8     }
9 }
10
11 public class GenericsExample {
12     public static void main(String args[]) {
13         Box<FrootLoops> frootLoops =
14             new Box<>(new FrootLoops());
15         System.out.println(frootLoops);
16     }
17 }
```

See you on Wednesday!

- We'll introduce **Collections** which use generics.
- Work on **Homework 2**, due 2/27 at 11:59pm.
- Make use of **CS Help** in 75 Shannon 203:
 - Sunday 6 - 8 pm (Yahya)
 - Sunday 8 - 10 pm (Yahya)
 - Monday 4 - 6 pm (Frannie)
 - Monday 8 - 10 pm (AJ)
 - Tuesday 4 - 6 pm (Yahya)
 - Wednesday 4 - 6 pm (Frannie)
 - Wednesday 8 - 10 pm (AJ)
 - Thursday 6 - 8 pm (AJ)