what is the nature of a scientific result in computer science?
what is the vocabulary?
what sorts of concepts are expressed?

science? science? we don’t neeeeed no ...
some 200-level computer science concepts

computation (at least, to computer scientists)
  - process versus processor
    - von Neumann architecture and other models
    - a language and its interpreter
    - interpretation as process
  - representations
    - numbers, objects, and relationships
    - algebraic and logical expressions
    - algorithm and flow of control
    - sequentiality and parallelism
algorithm **complexity** and the order of growth
big "O" notation, e.g. $O(n)$, $O(n^2)$
recursion (linear, tree, ...)
sequentiality, parallelism, search, heuristics, hard problems
**simulation, models, and abstractions**
abstracting processes, interactions between objects
modeling natural systems
200-level concepts, con’t.

**encapsulation** (of data, methods)
what do we mean by “data” versus "methods"?
is that a hard and fast dichotomy?
why is encapsulation useful?

**instance / object / class**
what sorts of concepts are these?
how do they relate to their real-world counterparts?

*computer science makes abstraction concrete*

**representations**
of abstract data structures
(e.g., arrays, stacks and queues, trees, ...)
what sorts of things are they?
of domain-specific things, relationships
211 concepts

Object Oriented Design
  objects and encapsulation
  division of labor (responsibilities of objects)
  abstraction barriers
abstract engines
  state machines (finite state automata)
interpreters
designing for extensibility (and other ity’s)
capitalizing on similarities
  solve once and reuse, ...
design patterns
  state, visitor, observer, decorator, ...
containers and enumeration
  encapsulating iteration
211 concepts, con’t.

Object Oriented Programming

- classes, methods and instance variables
- class hierarchies and simple inheritance
- controlling access through visibility modifiers
- shadowing of variables, methods
- abstract classes and interfaces
- exceptions
- graphics and event listeners
- ‘best practices’ in software development (testing, delivery, ...)
applying the concepts

a ‘staged delivery’ term project
a game or simulation of your own design and implementation
conceptualizing your own problem domain
what active agents, inactive objects, actions, locales, events?
classes and methods that capture essence of that domain
using Unified Modeling Language (UML) to describe the entities and their relationships comprising your domain
for you to explore and refine your design
to document and explain your design
progressive refinement, elaboration of your OOD as you develop an implementation using OOP techniques
basic programming practices
successively incorporating design patterns
more than just Oh Oh

creating fresh new programs from scratch, using the very finest of modern ingredients:

the basic OO skill set of classes, interfaces, encapsulation, shadowing, exceptions, etc.

some design patterns:

- Strategy Pattern
- Observer Pattern
- Decorator Pattern
- Factory Pattern
- Visitor Pattern
- State Pattern
- Command Pattern

plus careful reflection on the design as it evolves

rapid prototyping and

refactoring
Hierarchies

Making some hierarchies first:

**Animate objects**
entities that are decidedly active, not passive, can move between **Locations**, engage in **actions** with **Inanimate objects** and each other.

**Inanimate objects**
things that are acted upon, have **Location**, are passive (or reactive) more than active.

**Location objects**
where things **Actions** happen that involve **Animates** and **Inanimates** (or other **Animates**).

**Action objects**
something that happens involving **Animates** and **Inanimates** (or other **Animates**) at some **Location**.
having the hierarchies implement interfaces

casting subclass to superclass, vice versa

using interfaces and classes interchangeably

shadowing methods (and what happens in casting)

stuff like that...

but then we might want subclasses to differ in their "behavior" (Strategy Pattern)

for an object to be "aware" of another (Observer Pattern)

for variations on some kind of object to be given different attributes (Decorator Pattern)... and so forth
creating hierarchies is good, but runs into limitations
good for creating a simple collection of subclasses, each with different behaviors and attributes, with much shared behavior in the superclasses,
but if the subclasses have divergent behaviors associated with similar named methods, ...
the text "Head First Design Patterns" will show how to create far more elegant and extensible code than might come to mind just using standard OOP practices.
relationships between classes

a basic relationship: membership (informal usage suggested by English phrases "x is a kind of ...", or simply "x is a ...")

- a Human is a MetabolizingThing
- a Human is a AnimateThing
- a Human is a ThingWithMass
- a Human is a BiodegradableThing
- a Human is a LargerThanBreadboxThing

in each case the class Human shares properties with other members of that class

some such "is a" relationships naturally nest:

an A is a B (which is a C (which is a D ...
"is a" relationships among classes

some "is a" relationships naturally nest:
  an A is a kind of B (which is a kind of C (which is a D ... i.e., if x is an instance of an A, then x also has the properties of being a B, ...
  a Human is a Primate is a Mammal is a Vertebrate is a Animal ...

"has the properties of" = inheritance
inheritance of properties

Human inherits
its grasping hands from Primate
its fur from Mammal
its backbone from Vertebrate
and its metabolism from Animal.
In general, an A inherits most of (but not all) the properties of also being a B, etc.
[I'm foreshadowing shadowing]
but in every day life; some classes have properties that derive partly:
from being members of class A (within B ... and simultaneously from being in class X (within Y ...)
where A and X are orthogonal (independent) hierarchies
[we'll get to that complexity later; we'll stay with simple hierarchies for now]
Java supports hierarchies of classes

Human inherits its metabolism from Animal,
its backbone from Vertebrate,
its fur from Mammal,
grasping hands from Primate

class Animal { <basal functions here> }
class Vertebrate extends Animal { <more> }
class Mammal extends Vertebrate { <more> }
class Primate extends Mammal { <more> }
class Human extends Primate { <more> }
class Cat extends Mammal { <more> }
class Fish extends Vertebrate { <more> }
Java supports hierarchies of classes

class Primate extends Mammal { <additions> }
class Human   extends Primate { <additions> }
class Cat     extends Mammal  { <additions> }
class Dog     extends Mammal  { <additions> }

Mammal m;
Cat   c = new Cat();
Dog   d = new Dog();

wouldn’t it be useful to write:
m = c;
Java supports hierarchies of classes

```java
Cat    c1 = new Cat("Anne Boots");
Dog    d1 = new Dog("Gromit");
Mammal m  = c1;
m = new Cat("Fred");
```

it seems m is essentially acting as a pronoun, referring to any instance of Mammal.
and c1 can refer to any instance of a Cat.

btw, what happened to Anne Boots?
(c1 still references it, even if m no longer does)
and to those who know about "pointers" (eeek!)
think of the equivalence of pointers, pronouns, and physically pointing at an object
(so CS explicitly deals with ideas of reference and type)
inheritance and hierarchies

If one has a class C, and wanted another class D which is like C but has some additional properties or capabilities ...

Instead of copying code of C and renaming it as D, do this:

```java
class C { ... }
class D extends C { <extensions go here> }
```

And now every D is also a C:
```
C c1 = new C();
D d1 = new D();
(d1 instanceof D) == true
(d1 instanceof C) == true
```
Mammal m;
Cat c;
Dog d;

m = new Cat(); // legal (implicit cast)
c = m;         // not allowed by compiler!
c = (Cat)m;    // legal but dangerous!
m = new Dog(); // legal, but then
// now try to cast a Dog into a Cat:
c = (Cat)m;    // runtime error

if (m instanceof Cat)
    c = (Cat)m;
else if (m instanceof Dog)
    ...

an example

class Animal {
}
class Vertebrate extends Animal {
}
class Reptile extends Vertebrate {
}
class Mammal extends Vertebrate {
}
class Crocodile extends Reptile {
}
class Dog extends Mammal {
}

class Test1 {
    public static void main (String[] args) {
        Dog       d = new Dog();
        Crocodile c = new Crocodile();
        Mammal    m = d;
        System.out.println(m instanceof Dog);
        System.out.println(d instanceof Mammal);
        System.out.println(c instanceof Dog);
    }
}

Test1.java:19: Impossible for Crocodile to be instance of Dog.
System.out.println(c instanceof Dog);
distributing specializations

every animal has a scientific name.

the non-OOP way to implement a method to return an animal's scientific name would be:

class Animal {
    public String getName() {
        if (this instanceof Dog)
            return "Canis familaris";
        else if (this instanceof Cat)
            return "Felix dakat";
        < and so forth >
        return "unknown";
    }
}

Dog d = new Dog();
System.out.println(d.getName());
>> Canis familaris
critique of getName

seems a good idea to put all names in one place
... every Dog, Cat, etc. is an Animal, so put the method in Animal, right?

That allows:
Animal a = new Dog();
System.out.println(a.getName());
>> Canis familiaris

but it is a little strange when you ask that of an Animal instance:
Animal a = new Animal();
System.out.println(a.getName());
>> unknown

since every sort of animal is some specific kind, why allow making an instance of a (generic, non-specific) Animal
abstract classes

every specific kind of animal has a scientific name, but an animal in the abstract does not.

any instance of an animal is some specific type of animal ... so

abstract class Animal {
    abstract public String getName();
}
class Dog extends Animal {
    public String getName() { return "Canis familiaris"; }
}
Animal a = new Animal(); // won't compile
Dog d = new Dog();
the OO approach to getName

```java
abstract class Animal {
    abstract public String getName();
}

class Dog extends Animal {
    public String getName() {
        return "Canis familiaris";
    }
}

class Cat extends Animal {
    public String getName() { return "Felix dakat"; }
}

Note: I abbreviated the hierarchy for Dog and Cat to extend Animal (as opposed to Mammal, etc.)
graphically representing inheritance

Unified Modeling Language, UML
adding methods

let's flesh out the Animal class a bit:

every animal weight (regardless of specific type) and the weight may vary across/within individuals

it would be useful to set/get an animal's weight

abstract class Animal {
    private int weight;
    abstract public String getName();
    public void setWeight(int w) { weight = w; }
    public int getWeight() { return weight; }
}

so we've added:

public accessor methods: setWeight and getWeight
private (hidden) instance variable weight
incidentally, why not simply allow public access to weight?
distributing methods within a hierarchy

notions of:

- **superclass** (Animal, wrt Mammal)
- **subclass** (Mammal, wrt Animal)
- Mammal "is derived from" Animal
- Mammal extends Animal

in general:

- methods common to all subclasses are put in the highest common superclass
  - `getWeight()`
  - `setWeight(int w)`

- methods specific to subclass are placed in subclass, and abstract in superclass
  - `getName()`
initializing via constructors

abstract class Animal {
    private int weight;
    abstract String setName();
    public void setWeight(int w) { weight = w; }
    public int getWeight() { return weight; }
}

who calls setWeight?
what if it were never called?
how to insure any Dog instance has a valid weight?
    answer: via the constructor

class Dog extends Mammal {
    public Dog() { this(50); }
    public Dog(int w) { setWeight(w); }

    public String getName() { return "Canis familiaris"; }
}
more regarding design

previous version had the "magic constant" 50
here is a new version:

class Dog extends Mammal {
    final int DEFAULT_WEIGHT = 50;
    final int MIN_WEIGHT     = 1;
    final int MAX_WEIGHT     = 200;

    public Dog() { setWeight(DEFAULT_WEIGHT); }  
    public Dog(int w) {
        if (w < MAX_WEIGHT && w > MIN_WEIGHT)
            setWeight(w);
        else
            setWeight(DEFAULT_WEIGHT);
    }

    public String getName() { return "Canis familiaris"; }
}
design principles (continued)

this new version:
- **encapsulates** some parameters specific to Dog
- deals with **default**
- has **backward compatibility**
- can be specialized for different **subclasses**