1. Prototypal Inheritance:
    a. Working with prototypes:
       true //because the child object has the property jumps set to true
       null // after deletion from the child, the parent still has the property set to null.
       undefined // after deletion from the parent, there is not attribute at all.

    b. Searching algorithm

```
let head = {
  glasses: 1
};

let table = {
  pen: 3
};
table.__proto__ = head;

let bed = {
  sheet: 1,
  pillow: 2
};

bed.__proto__ = table;

let pockets = {
  money: 2000
};

pockets.__proto__ = bed;

console.log(pockets.pen);
console.log(pockets.glasses);
console.log(head.glasses);
```

       Since we have been defining prototypes for each objects as we created the inheritance chain , the object lookup speed will be almost similar whether looking from the object at the end of the inheritance chain or from the base object.

    c. Where does it write?
       The write will occur in **rabbit** object as the execution of the function *eat()* is done with the reference to the **rabbit** object, the **this** object will infer to the enclosing object first which is the **rabbit**.

    d. Why are both hamsters full?

Since the function *each(food)* puts the food parameter in the stomach object which is referencing through **this** object. This object looks for property stomach to push the values into, when it cannot find the property in speedy object, it looks to its prototype object hamster and pushes the value in food parameter to stomach array.

2. F.prototype
   a. Changing prototype
      i. Alerts true. The line Rabbit.prototype = {} has not impact on existing property and its values.
      ii. Alerts false. Unlike previous case, the value of the property is assigned to a new value. So, the change is reflected.
      iii. Alerts true. The line **delete rabbit.eats** tries to remove the property eats from object rabbit. Since the property eats exists in the parent object and not the child object, the attribute and value on the parent object is not impacted.
      iv. Alerts undefined. Unlike previous condition that tries to remove the property from the child object, the line **delete Rabbit.prototype.eats** removes the property and values from the parent object so the child object doesn't have the property to get the value to alert out.

   b. Create an object with the same constructor
      Yes, we can create an arbitrary object from the constructor function. If the object is created from a constructor function, it succeeds other wise without the object from the construction function, it fails.

3. Native Prototypes
   a. Add method f.defer(ms) to functions

```
Function.prototype.defer = function(timer){
        setTimeout(this, timer);
}

function f() {
  alert("Hello!");
}

f.defer(5000);
```

   b. Add the decorating defer() to functions

```
Function.prototype.defer = function(timer) {
  let func = this;
  return function(x, y) {
    setTimeout(func.bind(this, x, y), timer);
  }
}

function f(a, b) {
  alert(a + b);
}

console.log(f.defer(1000));
f.defer(1000)(5, 6);
```

4. Prototype Methods
    a. The difference between calls

```
function Rabbit(name) {
  this.name = name;
}
Rabbit.prototype.sayHi = function() {
  alert(this.name);
};

let rabbit = new Rabbit("Rabbit");
```

rabbit.sayHi();   // shows "Rabbit" because rabbit object has name, "Rabbit" and the function *sayHi()* reads from the object and prints out the content of **name** property.
Rabbit.prototype.sayHi(); // shows **undefined**. This calls the function *sayHi()* but since there aren't any value to the property name, it alerts out undefined.
Object.getPrototypeOf(rabbit).sayHi();   // shows **undefined**. It is the another way calling the function *sayHi()* but with method getPrototypeOf() method of Object class but the logic is same as the above execution.
rabbit.__proto__.sayHi(); // shows **undefined**. This is yet another way of calling the function *sayHi()* but the logic that is prints out undefined is similar to the last two.