JÖNKÖPING UNIVERSITY

School of Engineering

ADVANCED JAVASCRIPT

Web Development with JavaScript and DOM

TWJK14 Spring 2017

Peter Larsson-Green

EXTRA LAB SESSION

Rikard Olsson, Tuesday 17:00-19:00 in E2432

REGULAR EXPRESSIONS

Used in formal language theory to describe a regular language.

Example

The language described by the regular expression `ab{1,3}` contains the words `ab`, `abb` and `abbb`.

Often used by programmers to:

- Validate input from the user.
- Search for something in a string.
- Replace something in a string.

Are often hard to write.

Are often very hard to read.

Programmers use RegExp, not regular expressions!

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Matching a character:

- `/x/` (a character)
  "The string must contain the character `x`."
  - Does it match `"xyz"`?
  - Does it match `"zyx"`?
  - Does it match `"abc"`?
  - Does it match `""`?
  - Does it match `"xxx"`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Matching a character:

- `/x/` (a character)
  "The string must contain the character `x`."
- `/[a7x]/` (square brackets)
  "The string must contain one of the characters between [and]."
  - Does it match `"xxx"`?
  - Does it match `"yxz7"`?
  - Does it match `"abc"`?
  - Does it match `""`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Matching a character:

- `/x/` (a character)
  "The string must contain the character `x`."
- `/[a7x]/` (square brackets)
  "The string must contain one of the characters between [and]."
- `/[^a7x]/` (square brackets with ^)
  "The string must contain one of the characters not between [and]."
  - Does it match `"xxx"`?
  - Does it match `"yxz"`?
  - Does it match `""`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Matching a character:

- `/[a-z]/` (ranges in square brackets)
  "The string must contain a character between `a` and `z`."
- `/./` (the dot symbol)
  "The string must contain one character."
- `/\d/` (an escaped d)
  "The string must contain a digit character" (same as `/[0-9]/`).
- `/\D/` (an escaped D)
  "The string must contain a non digit character" (same as `/[^0-9]/`).

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Repeated matching:

- `/x{4}/` (curly brackets with one number)
  "The string must contain 4 `x` in sequence."
- `/x{4,6}/` (curly brackets with two numbers)
  "The string must contain 4-6 `x` in sequence."
- `/x{4,}/` (curly brackets with one number and a comma)
  "The string must contain 4-∞ `x` in sequence."
- `/x*/` (the star symbol)
  "The string must contain 0-∞ `x` in sequence" (same as `/x{0,}/`).
- `/x+/` (the plus symbol)
  "The string must contain 1-∞ `x` in sequence" (same as `/x{1,}/`).

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Repeated matching:

- `/x?/` (the question mark symbol)
  "The string must contain 0-1 `x` in sequence" (same as `/x{0,1}/`).

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Compound matching:

- `/xy/` "The string must contain an `x` followed by a `y`."
  - Does it match `"x y"`?
  - Does it match `"axyz"`?
  - Does it match `"xy"`?
  - Does it match `"yxyx"`?
  - Does it match `"yx"`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Compound matching:

- `/xy/` "The string must contain an `x` followed by a `y`."
- `/x{1,3}y/` "The string must contain 1-3 `x` followed by a `y`."
  - Does it match `"xxxxxxyyy"`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Compound matching:

- `/xy/` "The string must contain an `x` followed by a `y`."
- `/x{1,3}y/` "The string must contain 1-3 `x` followed by a `y`."
- `/x?y?[ab]{2}/` "..."
    - Does it match `"a"`?
    - Does it match `"aax"`?
    - Does it match `"yxaa"`?
    - Does it match `"ba"`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Matching start and end:

- `/^x/` (the ^ symbol)
  "The string must start with `x`."
- `/x$/` (the `$` symbol)
  "The string must end with `x`."

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Alternative matching:

- `/(x|y{2,3})/` (parenthesis with |)
  "The string must contain at least one `x` or 2-3 `y` in a sequence."
  - Does it match `"y"`?
  - Does it match `"xxx"`?
  - Does it match `"xxxyy"`?
  - Does it match `"yyxx"`?
  - Does it match "yxyx"?
  - Does it match `""`?

REGEXP IN JAVASCRIPT

RegExp are written between two `//` in JavaScript.

Modifiers can be added at the end:

- `/x/i` (case-insensitive)
  "The string must contain `x` or `X`."
- `/x/g` (match all)
  "The string must contain x (we find all matches)."

REGEXP IN JAVASCRIPT

A RegExp to match a date on the format YYYY-MM-DD.

- `/^[0-9]{4}-[0-9]{2}-[0-9]{2}$/`
  - Matches `"0000-99-99"` ☹
- `/^[12][0-9]{3}-[01][0-9]-[0-3][0-9]$/`
  - Matches `"1000-00-00"` ☹
- `/^[12][0-9]{3}-(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])$/`
  - Matches `"1000-02-31"` ☹

Keep RegExp as simple as possible!

- Only check the YYYY-MM-DD format.
- Validate year month and date with ordinary JavaScript.

REGEXP IN JAVASCRIPT

```
var input = "2016-05-18"
var dateRegExp = /^[0-9]{4}-[0-9]{2}-[0-9]{2}$/
if(dateRegExp.test(input)){
  var parts = input.split("-")
  var year = parseInt(parts[0], 10)
  var month = parseInt(parts[1], 10)
  var date = parseInt(parts[2], 10)
  // Validate the actual date here...
}else{
  alert("Enter the date on the format YYYY-MM-DD.")
}
```

REGEXP IN JAVASCRIPT

```
var input = "I was born 2016-05-18, which was a sunny day."
var dateRegExp = /[0-9]{4}-[0-9]{2}-[0-9]{2}/
var match = dateRegExp.exec(input)
var date = match[0] // "2016-05-18"
```

```
var input = "I was born 2016-05-18, which was a sunny day."
var dateRegExp = /([0-9]{4})-([0-9]{2})-([0-9]{2})/
var match = dateRegExp.exec(input)
var date = match[0] // "2016-05-18"
var year = match[1] // "2016"
var month = match[2] // "05"
var date = match[3] // "18"
```

REGEXP IN JAVASCRIPT

```
var input = "2016-05-18 was a better day than 2014-11-27."
var dateRegExp = /[0-9]{4}-[0-9]{2}-[0-9]{2}/g
var match = input.match(dateRegExp)
var date0 = match[0] // "2016-05-18"
var date1 = match[1] // "2014-11-27"
```

REGEXP IN JAVASCRIPT

```
var input = '<html>...<img src="...">...</html>'
// We want to remove all images!
var imgRegExp = /<img.*>/g
input = input.replace(imgRegExp, "")
// input = '<html>...'
```

* is greedy!

```
var input = '<html>...<img src="...">...</html>'
// We want to remove all images!
var imgRegExp = /<img.*?>/g
input = input.replace(imgRegExp, "")
// input = '<html>......</html>'
```

? makes * ungreedy!

ONLINE REGEXP TESTER

- Test RegExp online:
  - https://regex101.com

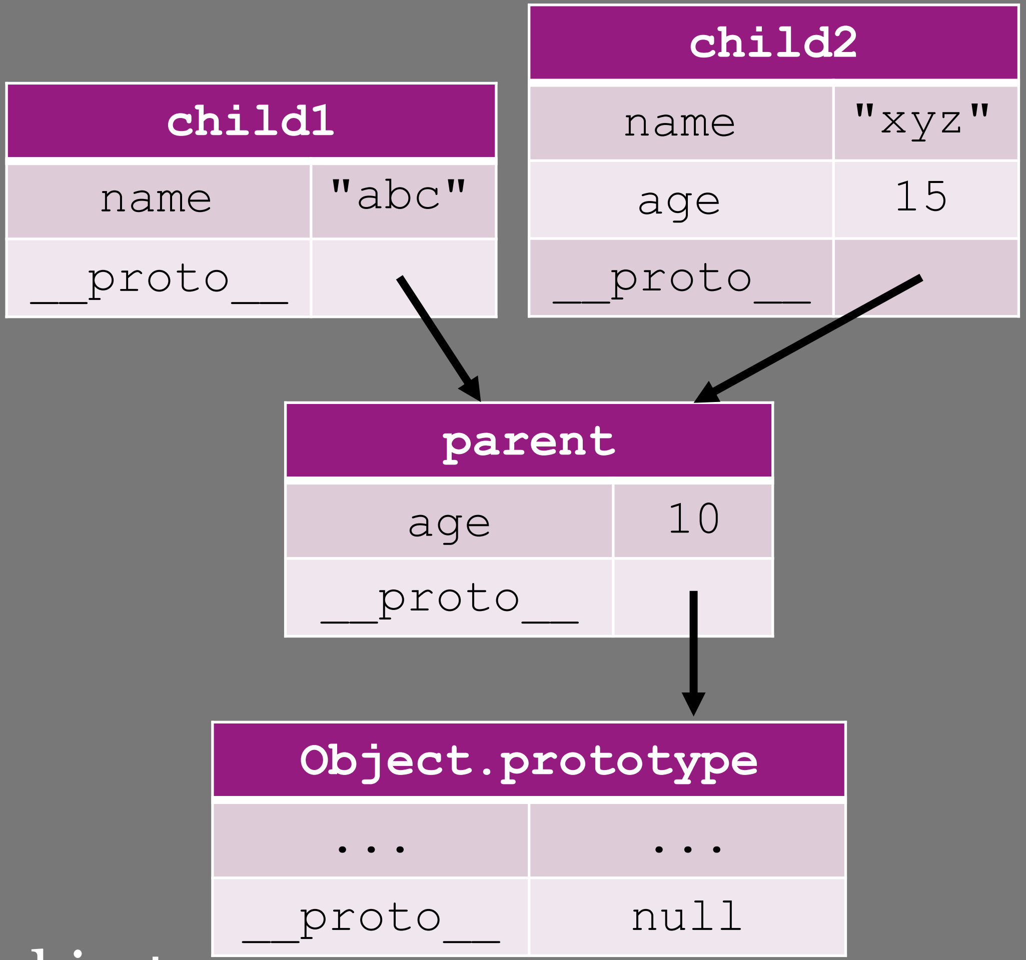
PROTOTYPAL INHERITANCE

Objects inherit from other objects.

- Each object has a "hidden" property named `__proto__`.
  - Contains a reference to the object the object inherits from.
  - Should never be used directly by you!
- Objects created with `{ ... }` inherits from `Object.prototype`.
- `Object.prototype` does no inherit from any object.
  - `Object.prototype.__proto__` is `null`.
- Use `Object.create(thePrototype)` to create an object that inherits from an object of your choice.

PROTOTYPAL INHERITANCE

```
var parent = {
  age: 10
}
var child1 = Object.create(parent)
child1.name = "abc"
var child2 = Object.create(parent)
child2.name = "xyz"
child2.age = 15
```

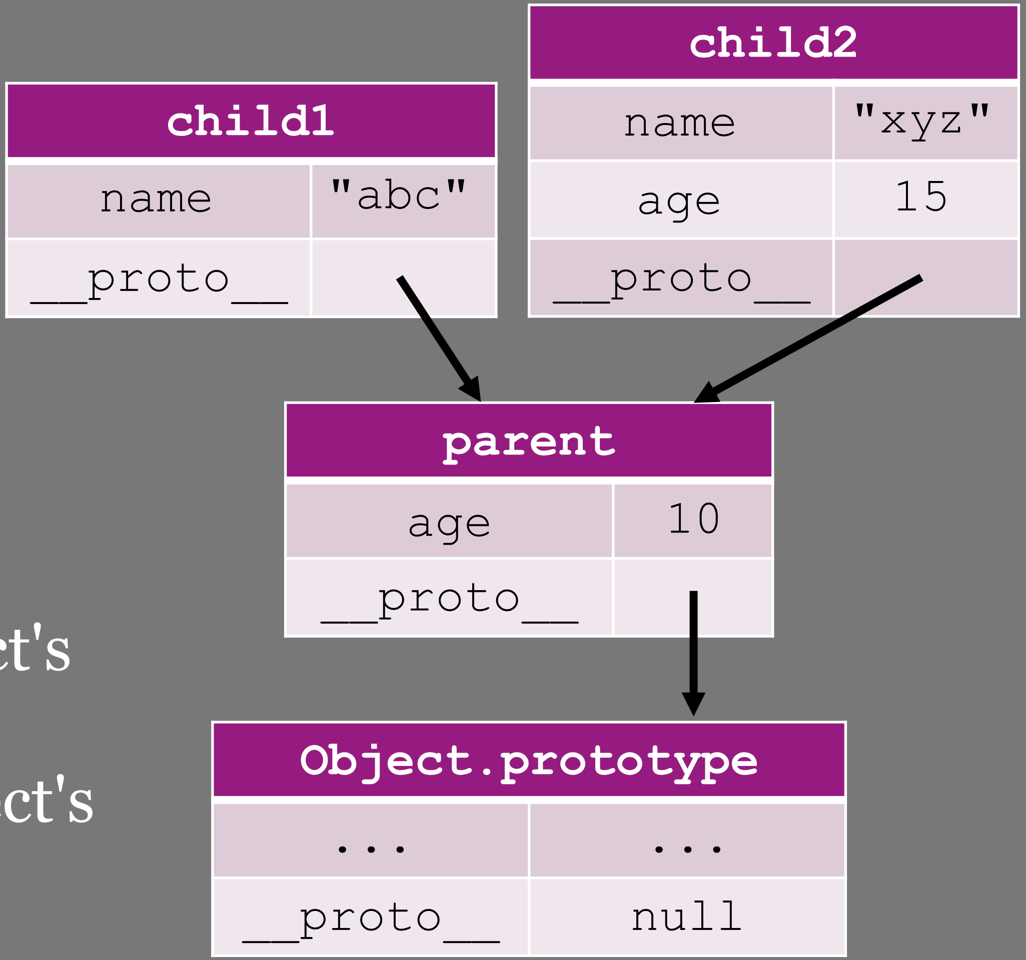


Procedure for writing

1. Always write the value directly to the object.

PROTOTYPAL INHERITANCE

```
var ten = child1.age
var fifteen = child2.age
var undefinedValue = child1.aaa
```



Procedure for reading

1. Look for the key in the object itself.
2. If it's not there, look for it in the object's prototype.
3. If it's not there, look for it in that object's prototype.
4. And so on...

CONSTRUCTORS

A more convenient way to create objects with a custom prototype.

- A constructor is a function called with the `new` operator.

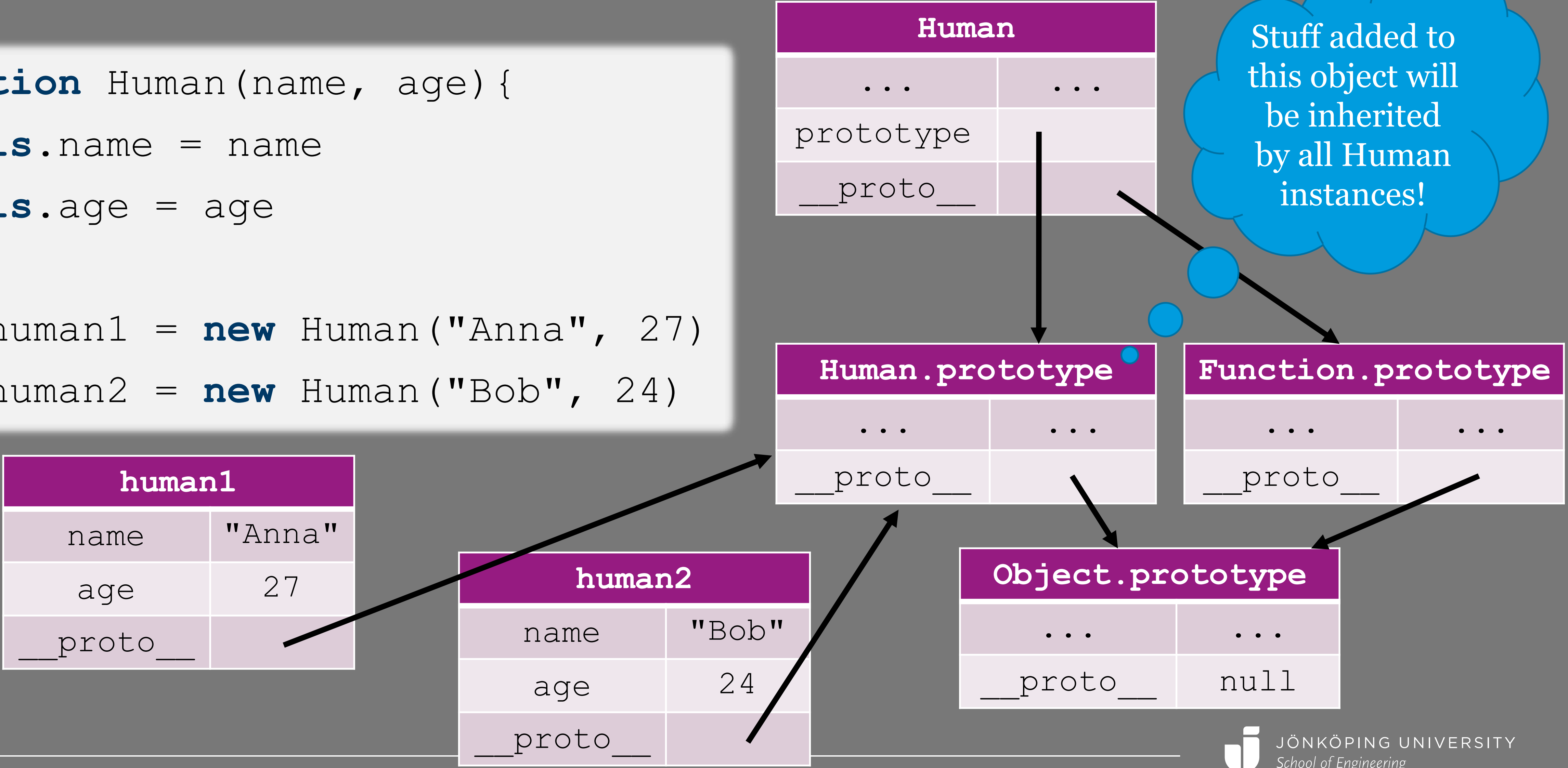
```
function Human(name, age){
  this.name = name
  this.age = age
}
var human = new Human("Anna", 27)
```

`this` refers to the new object.

- Functions are object → Functions have properties.
  - The `prototype` property will be used as the prototype for new instances.

CONSTRUCTORS

```
function Human(name, age){
  this.name = name
  this.age = age
}
var human1 = new Human("Anna", 27)
var human2 = new Human("Bob", 24)
```

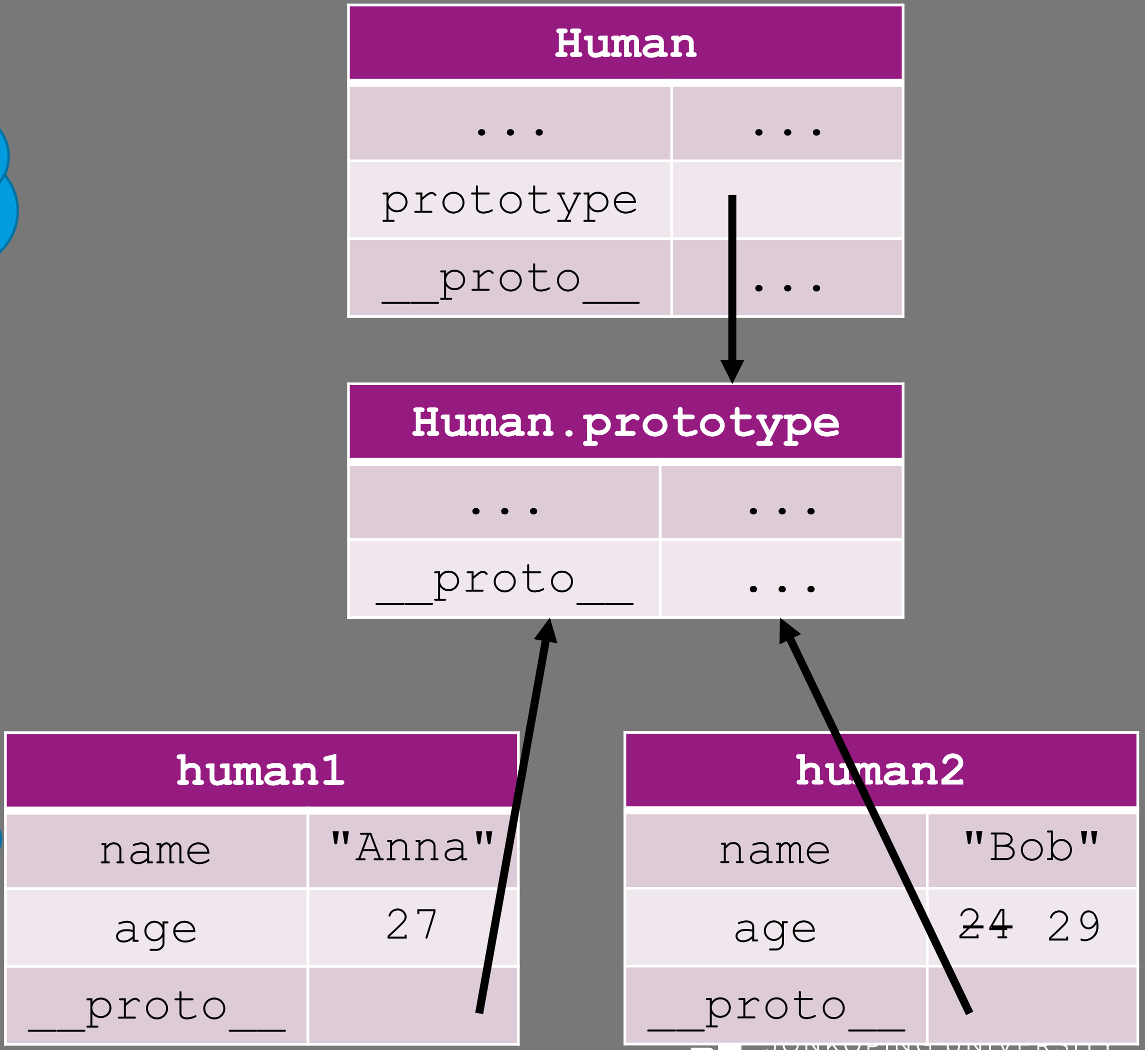


CONSTRUCTORS

```
function Human(name, age){
  this.name = name
  this.age = age
}
var human1 = new Human("Anna", 27)
var human2 = new Human("Bob", 24)
Human.prototype.incAge =
               function(amount){
  this.age += amount
}
human2.incAge(5)
```

Store data directly in the object.

Store methods in the prototype.



INHERITANCE

```
function Human(age){
  this.age = age
}
Human.prototype.incAge =
                function(amount){
  this.age += amount
}
```

```
var superman = new Superhero(10)
superman.incAge(5)
superman.fly()
```

```
// Superhero inherits from Human.
function Superhero(age){
  Human.call(this, age)
  this.isFlying = false
}
Superhero.prototype =
   Object.create(Human.prototype)
Superhero.prototype.fly =
                      function(){
  this.isFlying = true
}
```

CLASS SYNTAX IN ES6

```
class Human{
  constructor(age){
    this.age = age
  }
  incAge(amount){
    this.age += amount
  }
}
```

```
class Superhero extends Human{
  constructor(age){
    super(age)
    this.isFlying = false
  }
  fly(){
    this.isFlying = true
  }
}
```

```
var superman = new Superhero(10)
superman.incAge(5)
superman.fly()
```

ARRAYS

Can be created using:

- The `Array` constructor:
  - `new Array(numberOfElements)`
  - `new Array(element0, element1, ...)`
- The array literal expression:
  - `[element0, element1, ...]`

Arrays inherit from `Array.prototype`.

Avoid these 2!

- A stupid programmer might create the local function `Array`.
- Changging
  `new Array(3, 2)`
  to
  `new Array(3)`
  is none intuitive.

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.

```
var arr = ["I", "am", "red"]
arr.forEach(
  function(element, i){
    alert(element)
  }
)
```

```
var arr = ["I", "am", "red"]
for(var i=0; i<arr.length; i++){
  var element = arr[i]
  alert(element)
}
```

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.

```
var arr = ["I", "am", "red"]
arr.forEach(
  function(element, i){
    alert(element)
  }
)
```

```
Array.prototype.forEach = function(callback){
  for(var i=0; i<this.length; i++){
    callback(this[i], i)
  }
}
```

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.

```
var arr = ["I", "am", "red"]
var newArr = arr.map(
  function(element, i){
    return element + element
  }
)
```

```
var arr = ["I", "am", "red"]
var newArr = []
for(var i=0; i<arr.length; i++){
  var element = arr[i]
  newArr.push(element + element)
}
```

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.

```
var arr = ["I", "am", "red"]
var newArr = arr.map(
  function(element, i){
    return element + element
  }
)
```

```
Array.prototype.map = function(callback){
  var newArr = []
  for(var i=0; i<this.length; i++){
    newArr.push(callback(this[i], i))
  }
  return newArr
}
```

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.
- `filter` to produce a new array.

```
var arr = ["I", "am", "red"]
var newArr = arr.filter(
  function(element, i){
    return element.length>1
  }
)
```

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.
- `filter` to produce a new array.
- `some` to check if any element fulfills a condition.

```
var arr = ["I", "am", "red"]
var containsAm = arr.some(
  function(element, i){
    return element == "am"
  }
)
```

ARRAYS

```
var arr = ["I", "am", "red"]
var areAllEmpty = arr.every(
  function(element, i){
    return element == ""
  }
)
```

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.
- `filter` to produce a new array.
- `some` to check if any element fulfills a condition.
- `every` to check if all elements fulfills a condition.

ARRAYS

Arrays have many useful methods.

- `forEach` to iterate over the array.
- `map` to produce a new array.
- `filter` to produce a new array.
- `some` to check if any element fulfills a condition.
- `every` to check if all elements fulfills a condition.
- `reduce` to compute a value based on all elements.
- `reduceRight` to compute a value based on all element (right to left).

```
var arr = [6, 2, 9]
var sum = arr.reduce(
  function(tot, element, i){
    return tot + element
  },
  0
)
```

VARIABLES IN ES6

Variables can now be:

- More local than before.
    - Use the `let` statement.

```
var iAmAGlobalVariable = 1
function test(){
  var iAmALocalVariable = 2
  if(false){
    var iAmLocalToo = 3
  }
  // Can use iAmLocalToo.
}
```

```
var iAmAGlobalVariable = 1
function test(){
  var iAmALocalVariable = 2
  if(false){
    let iAmEvenMoreLocal = 3
  }
  // Can't use iAmEvenMoreLocal.
}
```

VARIABLES IN ES6

Variables can now be:

- More local than before.
  - Use the `let` statement.

```
var callbacks = []
for(var i=0; i<5; i++){
  callbacks.push(function(){
    return i // i will be 5!
  })
}
```

```
var callbacks = []
for(let i=0; i<5; i++){
  callbacks.push(function(){
    return i
  })
}
```

VARIABLES IN ES6

Variables can now be:

- More local than before.
  - Use the `let` statement.
- More constant than before.
  - Use the `const` statement.

```
const object = {a: 1}
object = 3 // Not OK.
object.b = 12 // OK.
```

FUNCTIONS IN ES6

Two new ways to create functions:

```
var sum = function(x, y){
  return x + y
}
```

```
var sum = (x, y) => x + y
```

```
var average = function(x, y){
  var sum = x + y
  return sum / 2
}
```

```
var average = (x, y) => {
  var sum = x + y
  return sum / 2
}
```

FUNCTIONS IN ES6

The arrow function keeps the value of `this`.

```
var human = {
  name: "Alice",
  delayedAlert: function(){
    var self = this
    setTimeout(function(){
      alert(self.name)
    }, 1000)
  }
}
```

```
var human = {
  name: "Alice",
  delayedAlert: function(){

    setTimeout(() => {
      alert(this.name)
    }, 1000)
  }
}
```

CAN WE USE ES6 NOW?

- Modern browsers supports almost all of it.
  - https://kangax.github.io/compat-table/es6/
  - http://caniuse.com/#search=es6
- Old browsers supports almost none of it.

But we can start to use it any way!

- Most ES6 features can be expressed with ES5 features.

```
const variable = 3
```

→

```
var variable = 3
```

  - Use https://babeljs.io to "transpile" it.

GETTERS

A getter is a function that returns a computed value in an object.

```
// 09:32:38 = 9*60*60 + 32*60 + 38 = 34358 seconds since 00:00:00.
var time = {
  totalSeconds: 34358,
  getSecond: function(){ return this.totalSeconds % 60 },
  getMinute: function(){ return Math.floor(this.totalSeconds/60) % 60 },
  getHour: function(){ return Math.floor(this.totalSeconds/(60*60)) }
}
var second = time.getSecond() // 38
var minute = time.getMinute() // 32
var hour = time.getHour() // 9
```

GETTERS

A getter is a function that returns a computed value in an object.

```
// 09:32:38 = 9*60*60 + 32*60 + 38 = 34358 seconds since 00:00:00.
var time = {
  totalSeconds: 34358,
  get second(){ return this.totalSeconds % 60 },
  get minute(){ return Math.floor(this.totalSeconds/60) % 60 },
  get hour(){ return Math.floor(this.totalSeconds/(60*60)) }
}
var second = time.second // 38
var minute = time.minute // 32
var hour = time.hour // 9
```

SETTERS

A setter is a function that computes and sets a value in an object.

```
// 09:32:38 = 9*60*60 + 32*60 + 38 = 34358 seconds since 00:00:00.
var time = {
  totalSeconds: 34358,
  getSecond: function(){ return this.totalSeconds % 60 },
  setSecond: function(newSecond){
    this.totalSeconds -= this.getSecond()
    this.totalSeconds += newSecond
  }
}
```

```
var s = time.getSecond() // 38
time.setSecond(12)
var s2 = time.getSecond() // 12
```

SETTERS

A setter is a function that computes and sets a value in an object.

```
// 09:32:38 = 9*60*60 + 32*60 + 38 = 34358 seconds since 00:00:00.
var time = {
  totalSeconds: 34358,
  getSecond: function(){ return this.totalSeconds % 60 },
  set second(newSecond){
    this.totalSeconds -= this.getSecond()
    this.totalSeconds += newSecond
  }
}
```

```
var s = time.getSecond() // 38
time.second = 12
var s2 = time.getSecond() // 12
```

THE FOR IN LOOP

```
var object = {a: 1, b: 2}
for(var key in object){
  var value = object[key]
  console.log(key, value)
}
```

We iterate over all enumerable properties (including inherited).

```
var arr = ["a", "b", "c"]
for(var i=0; i<arr.length; i++){
  var element = arr[i]
  console.log(i, element)
}
```

```
var arr = ["a", "b", "c"]
for(var i in arr){
  var element = arr[i]
  console.log(i, element)
}
```

CUSTOMIZING OBJECTS

Properties are enumerable by default.

```
Array.prototype.getSum = function(){
  return this.reduce(function(total, element){
    return total + element
  }, 0)
}
var arr = ["a", "b", "c"]
for(var i in arr){
  var element = arr[i]
  console.log(i, element)
}
```

Is enumerable.

`i` will also be `"getSum"`.

CUSTOMIZING OBJECTS

Properties are enumerable by default.

- Can be changed using the `Object.defineProperty` function.

```
var myObject = {}
Object.defineProperty(myObject, "theKey", {
  configurable: false,
  enumerable: false,
  value: undefined,
  writable: false
})
```

CALLBACK HELL

```
getUserByUsername("Lisa", function(user, error){
  if(user != null){
    const newPassword = generatePassword()
    storePassword(user, newPassword, function(user, error){
      if(error == null){
        emailPassword(user, newPassword, function(user, error){
          if(error == null){
            // Password change complete.
          }else{
            // Handle error.
          }
        })
      }else{
        // Handle error.
      }
    })
  }else{
    // Handle error.
  }
})
```

PROMISES

```
getUserByUsername("Lisa").then(function(user){
  const newPassword = generatePassword()
  return storePassword(user, newPassword)
}).then(function(user){
  return emailPassword(user.email, newPassword)
}).then(function(){
  // Password change complete.
}).catch(function(error){
  // Handle error.
})
```

PROMISES WITH ASYNC/AWAIT

```
try{
  const user = await getUserByUsername("Lisa")
  const newPassword = generatePassword()
  await storePassword(user, newPassword)
  await emailPassword(user.email, newPassword)
  // Password change complete.
}catch(error){
  // Handle error.
}
```

PROMISES

A Promise consists of:

- An executor:
  - Is a function doing the asynchronous work.
  - Two possible outcomes:
    - The work was successfully carried out → The promise is resolved.
    - The work was not successfully carried out → The promise is rejected.

```
const thePromise = new Promise(function(resolve, reject){
  setTimeout(function(){
    if(/* Worked carried out successfully */){
      resolve("The result of the work.")
    }else{
      reject("The error message.")
    }
  }, 1000)
})
```

PROMISE EXAMPLE

```
function sendGetRequest(uri){
  return new Promise(function(resolve, reject){
    const request = new XMLHttpRequest()
    request.open("GET", uri)
    request.addEventListener('load', function(){
      resolve(request.responseText)
    })
    request.addEventListener('error', function(){
      reject("error")
    })
    request.send()
  })
}
```

PROMISES

A Promise consists of:

- Listeners for when the promised is fulfilled.
  - Will be called when the promise is fulfilled.

```
thePromise.then(function(resolvedValue){
  // I'm called when the promise has been resolved.
})
```

- Listeners for when the promise is rejected.
  - Will be called when the promise is rejected.

```
thePromise.catch(function(rejectedValue){
  // I'm called when the promise has been rejected.
})
```

PROMISE EXAMPLE

```
sendGetRequest("http://ju.se").then(function(responseBody){
  // Do something.
}).catch(function(errorMessage){
  // Do something.
})
```

```
try{
  const responseBody = await sendGetRequest("http://ju.se")
  // Do something.
}catch(errorMessage){
  // Do something.
}
```

ES6 MODULES

Module = a JS files exporting values.

- Exported values can be imported by other JS files.

```
function sum(x, y){
  return x + y
}
export { sum }
```

math.js

```
import { sum } from './math'

alert(sum(3, 4)) // 7
```

main.js

Does not work in web browsers yet.

- Module bundlers to the rescue! E.g. webpack, browserify, ...

NEXT LECTURE

- Is not mandatory.
- Introduction to webpack (45 minutes?).
- Feedback on your work (45 minutes?).
  - If you want, email your work to Peter.Larsson-Green@ju.se.

RECOMMENDED READING

You Don't Know JS:

- this & Object Prototypes:
  - https://github.com/getify/You-Dont-Know-JS/blob/master/this%20&%20object%20prototypes/README.md
- Async & Performance:
  - https://github.com/getify/You-Dont-Know-JS/blob/master/async%20&%20performance/README.md
- ES6 & Beyond:
  - https://github.com/getify/You-Dont-Know-JS/blob/master/es6%20&%20beyond/README.md

ECMAScript 7.0 Specification

- http://www.ecma-international.org/ecma-262/7.0