JavaScript for ABAP Programmers

Referential Inheritance

Chris Whealy / The RIG

ABAP

Strongly typed
Syntax similar to COBOL
Block Scope
No equivalent concept
OO using class based inheritance
Imperative programming

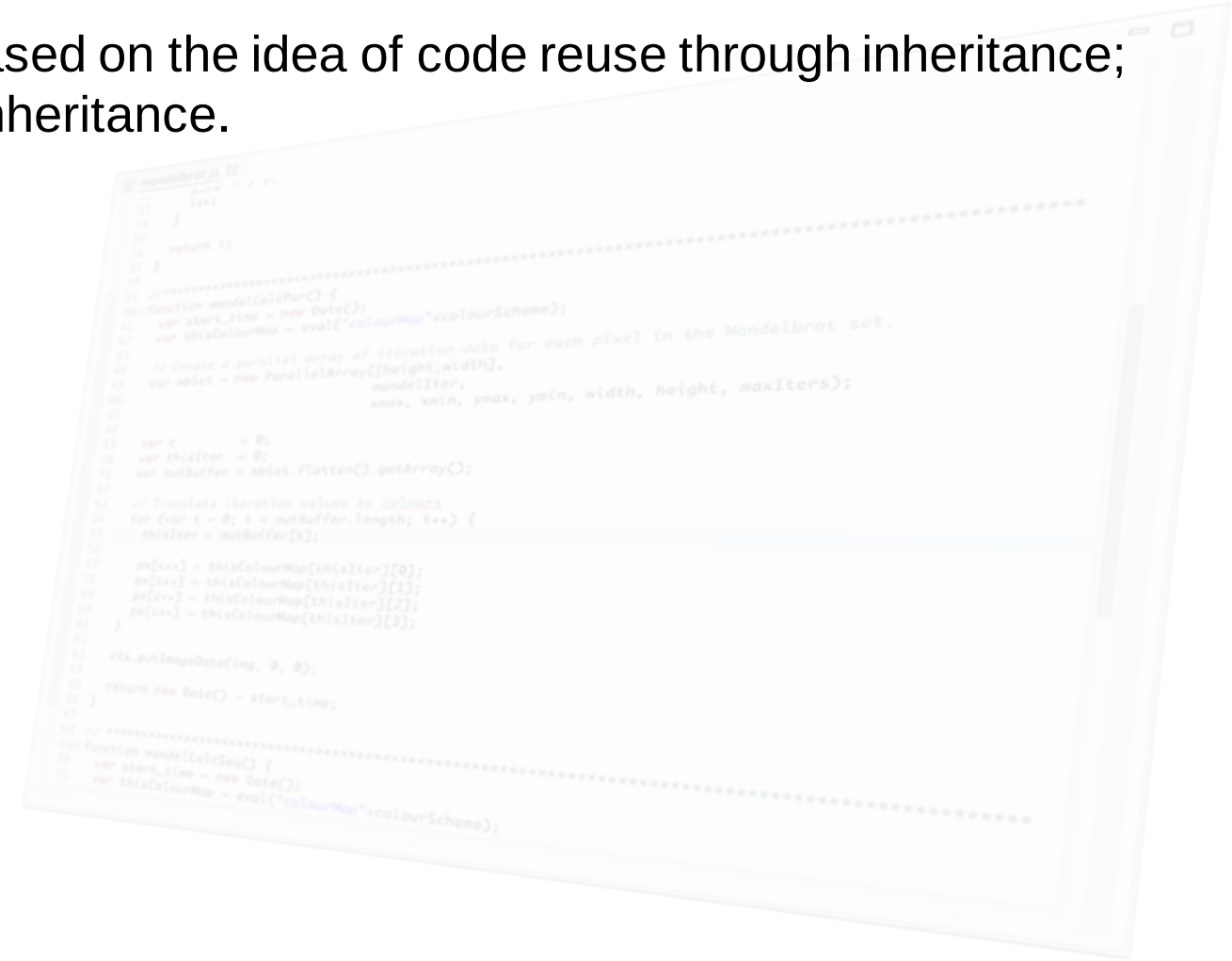
JavaScript

Weakly typed
Syntax derived from Java
Lexical Scope
Functions are 1st class citizens
OO using referential inheritance
Imperative or Functional programming

Object Orientation and Inheritance

Object Orientation and Inheritance

Object orientation is a programming concept based on the idea of code reuse through inheritance; however, there are two ways of implementing inheritance.



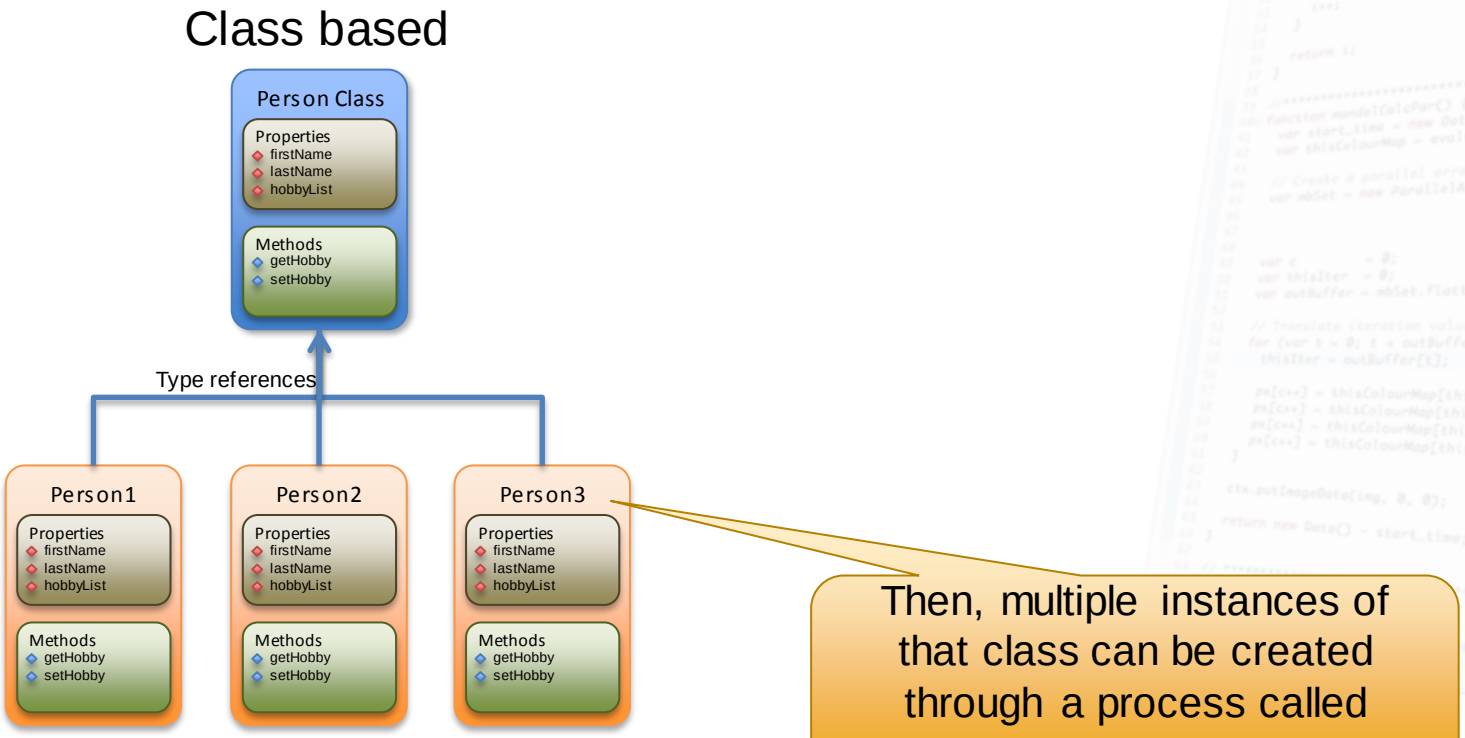
Object Orientation and Inheritance

Object orientation is a programming concept based on the idea of code reuse through inheritance; however, there are two ways of implementing inheritance.

Class based

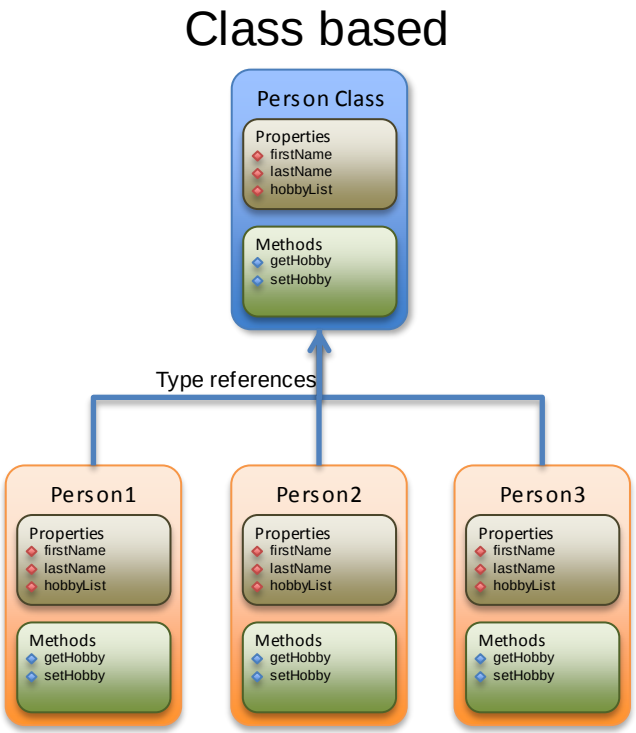
Object Orientation and Inheritance

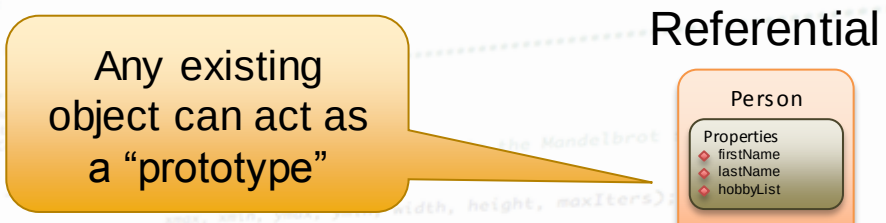
Object orientation is a programming concept based on the idea of code reuse through inheritance; however, there are two ways of implementing inheritance.



Object Orientation and Inheritance

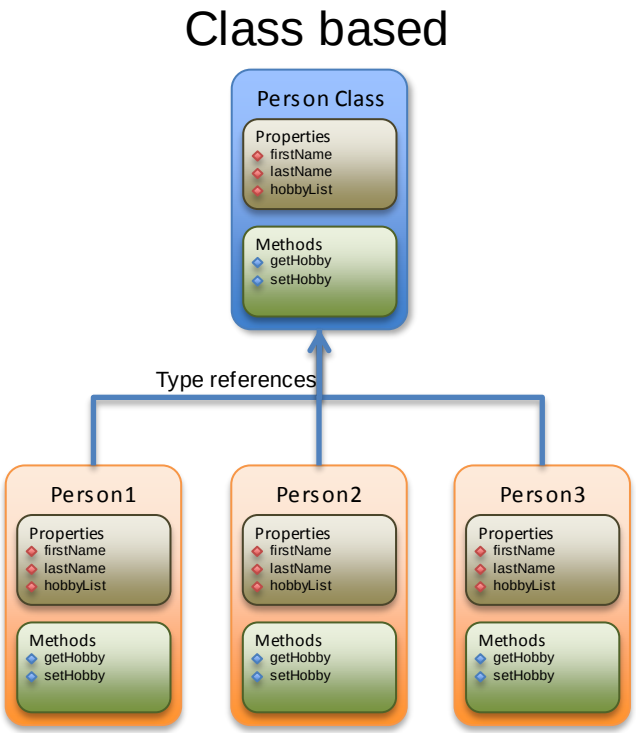
Object orientation is a programming concept based on the idea of code reuse through inheritance; however, there are two ways of implementing inheritance.

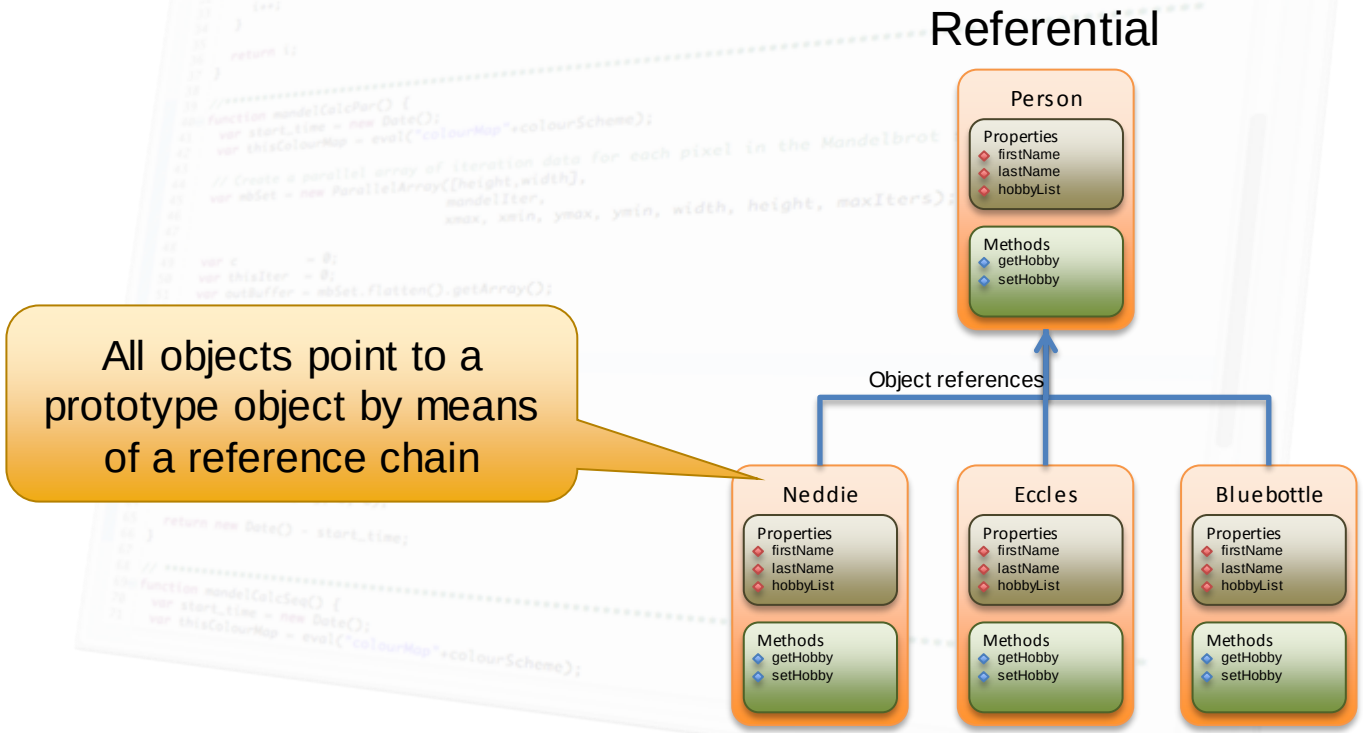




Object Orientation and Inheritance

Object orientation is a programming concept based on the idea of code reuse through inheritance; however, there are two ways of implementing inheritance.

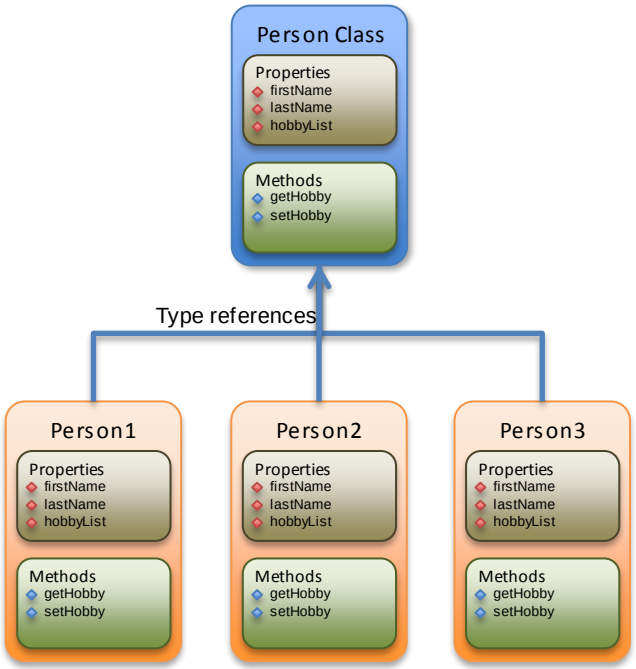




Class-based Inheritance vs. Referential Inheritance 1/2

There is a key conceptual difference between class-based inheritance and referential (or prototypical) inheritance.

Class based

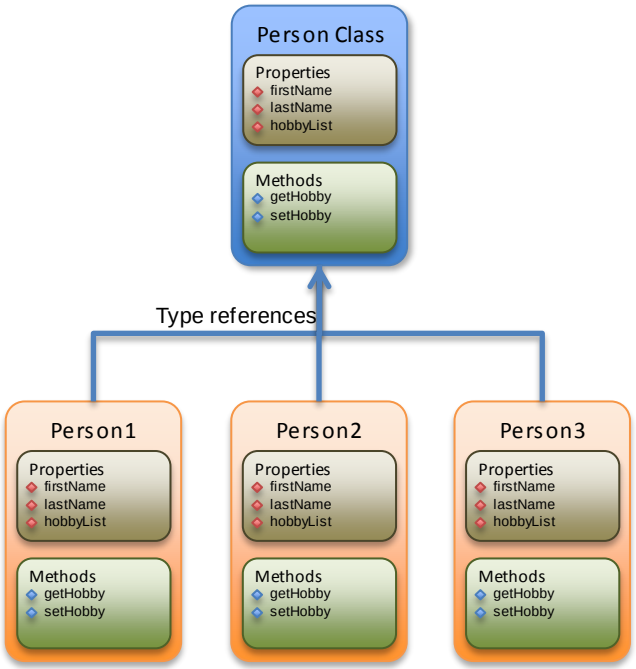


Class-based inheritance (used in Java or ABAP) specifies that unless a class is defined as static, all object instances must be created through a process called instantiation. During instantiation, the class acts as a template by which the object instance is defined.

Class-based Inheritance vs. Referential Inheritance 1/2

There is a key conceptual difference between class-based inheritance and referential (or prototypical) inheritance.

Class based



Class-based inheritance (used in Java or ABAP) specifies that unless a class is defined as static, all object instances must be created through a process called instantiation. During instantiation, the class acts as a template by which the object instance is defined.

Key feature of class-based inheritance

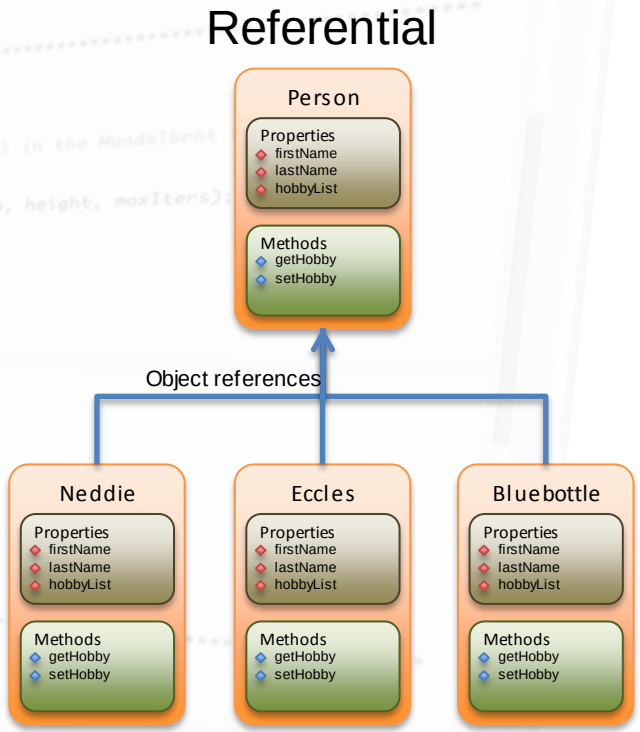
Once an object instance has been created, ***that instance exists as an independent entity from its parent class***.

Any changes made to the parent class have no effect on existing child instances.

Class-based Inheritance vs. Referential Inheritance 2/2

There is a key conceptual difference between class-based inheritance and referential (or prototypical) inheritance.

Referential (or prototypical) inheritance has no concept of creating an object instance from a class. New objects are simply created with a reference to some other object that acts as a “prototype”.



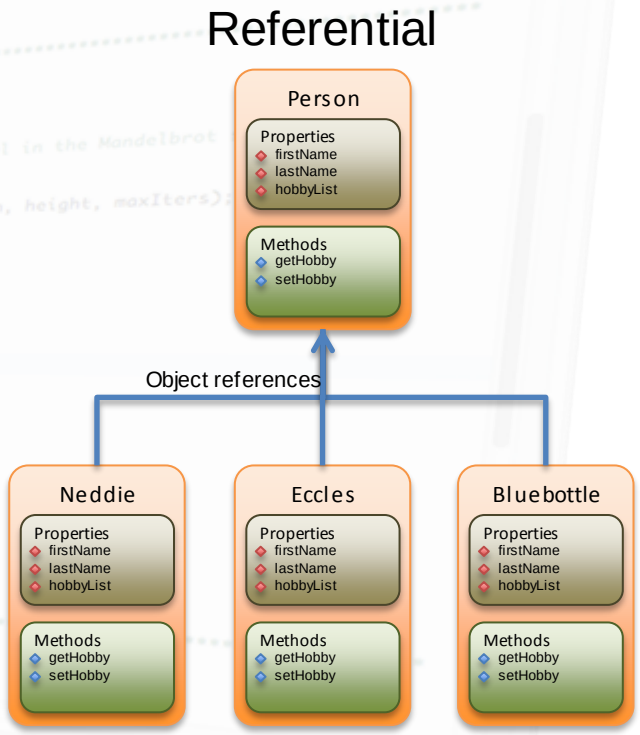
Class-based Inheritance vs. Referential Inheritance 2/2

There is a key conceptual difference between class-based inheritance and referential (or prototypical) inheritance.

Referential (or prototypical) inheritance has no concept of creating an object instance from a class. New objects are simply created with a reference to some other object that acts as a “prototype”.

The link from an object to its prototype is known as a “prototype chain”.

A prototype chain is a chain of objects used to implement both code reuse through inheritance and shared properties.



Class-based Inheritance vs. Referential Inheritance 2/2

There is a key conceptual difference between class-based inheritance and referential (or prototypical) inheritance.

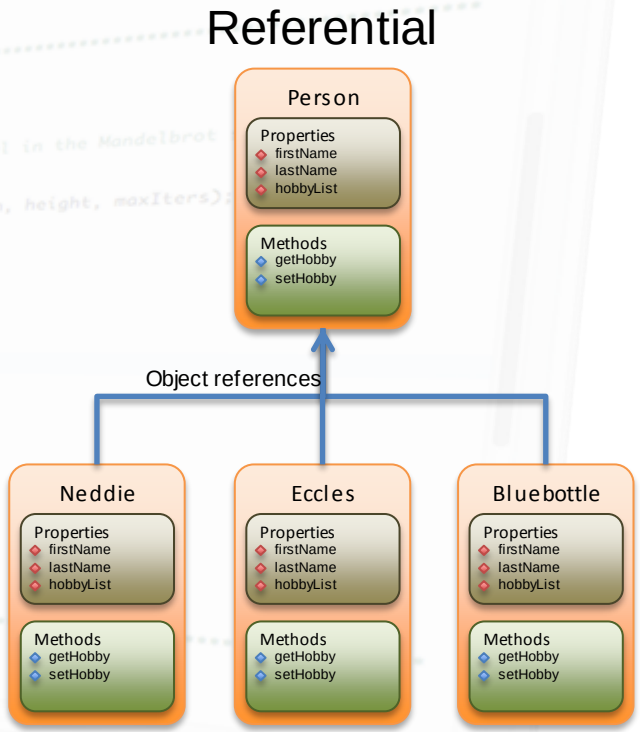
Referential (or prototypical) inheritance has no concept of creating an object instance from a class. New objects are simply created with a reference to some other object that acts as a “prototype”.

The link from an object to its prototype is known as a “prototype chain”.

A prototype chain is a chain of objects used to implement both code reuse through inheritance and shared properties.

Key feature of referential inheritance

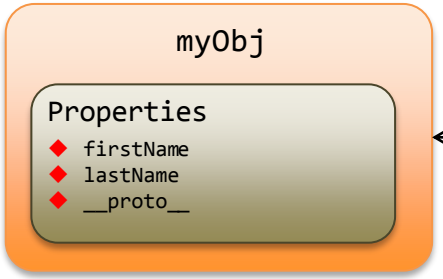
Any changes made to the prototype object ***are immediately visible to all referencing objects***.



Understanding the Prototype Chain

Understanding the Prototype Chain 1/2

All JavaScript objects have a default property called `__proto__` that is used to define the next object in the prototype chain.*



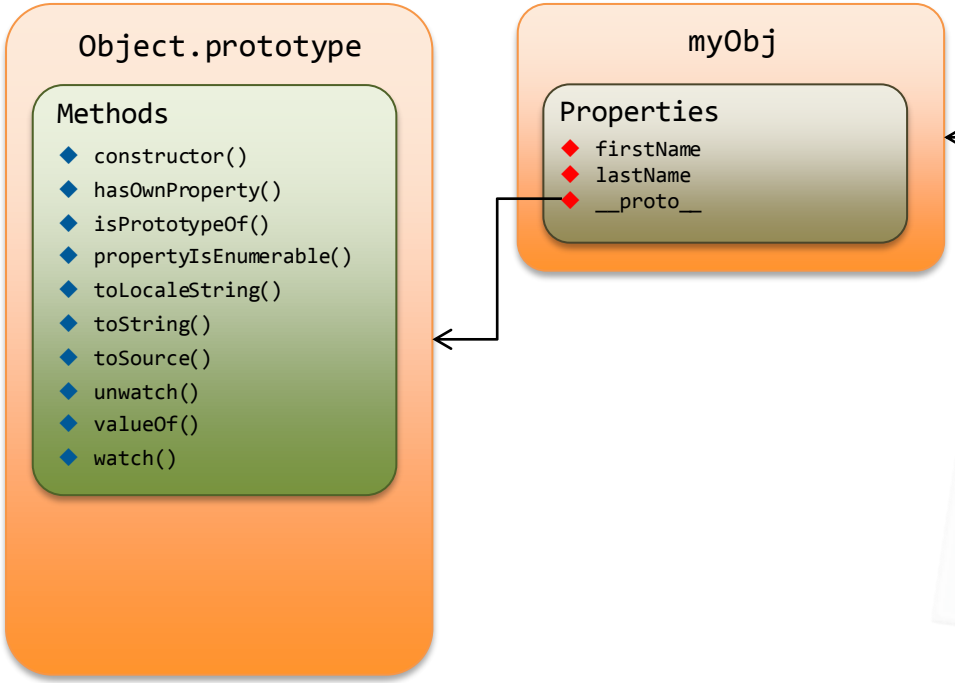

```
// Create an object
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk'
}
```

* `__proto__` is not an official property name defined in the ECMAScript specification, but apart from Internet Explorer, all modern browsers implement this property.

Understanding the Prototype Chain 1/2

All JavaScript objects have a default property called `__proto__` that is used to define the next object in the prototype chain.*

Whenever you create a JavaScript object, by default `__proto__` will point to `Object.prototype`.



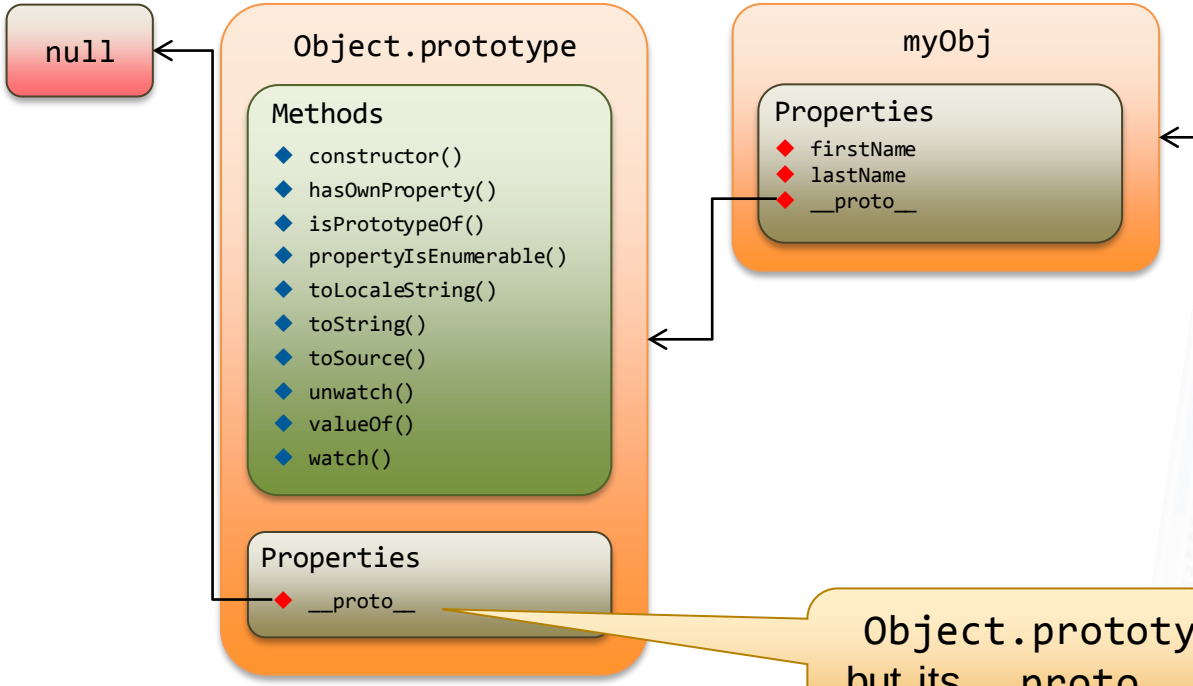

```
// Create an object
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk'
}

myObj.__proto__;   // → Object.prototype

// All the methods of Object.prototype are now
// available to myObj
myObj.valueOf();
```

* `__proto__` is not an official property name defined in the ECMAScript specification, but apart from Internet Explorer, all modern browsers implement this property.

Understanding the Prototype Chain 1/2

All JavaScript objects have a default property called `__proto__` that is used to define the next object in the prototype chain.*

Whenever you create a JavaScript object, by default `__proto__` will point to `Object.prototype`.



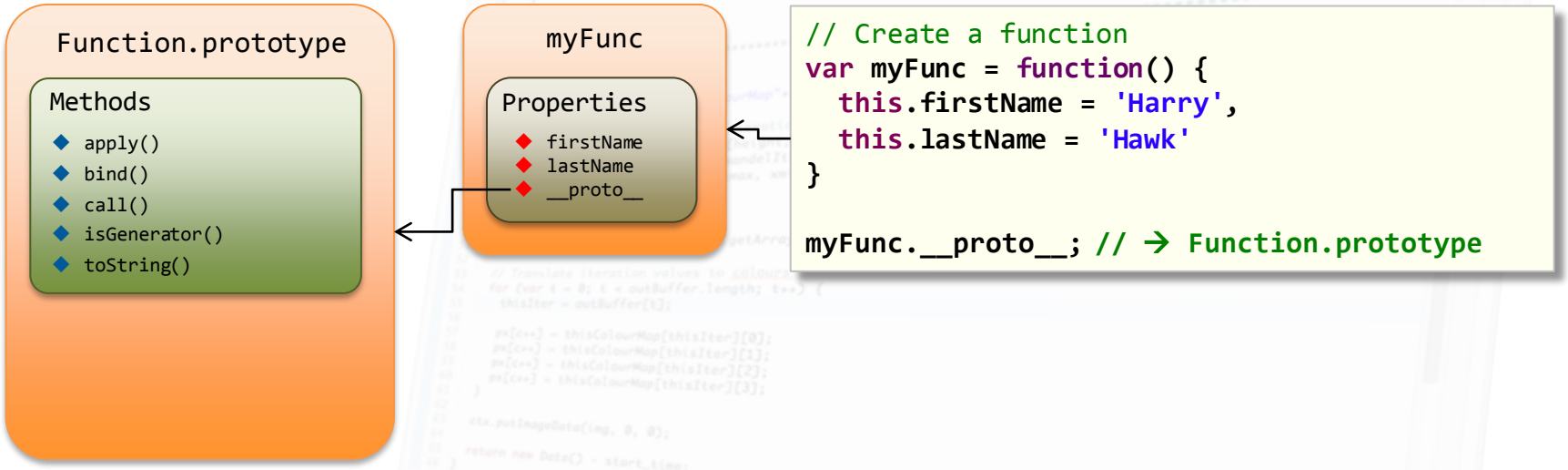

```
// Create an object
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk'
}

myObj.__proto__;   // → Object.prototype

// All the methods of Object.prototype are now
// available to myObj
myObj.valueOf();
```

`Object.prototype` is itself a JavaScript object, but its `__proto__` property is always set to null to indicate the end of the prototype chain

* `__proto__` is not an official property name defined in the ECMAScript specification, but apart from Internet Explorer, all modern browsers implement this property.

Understanding the Prototype Chain 2/2

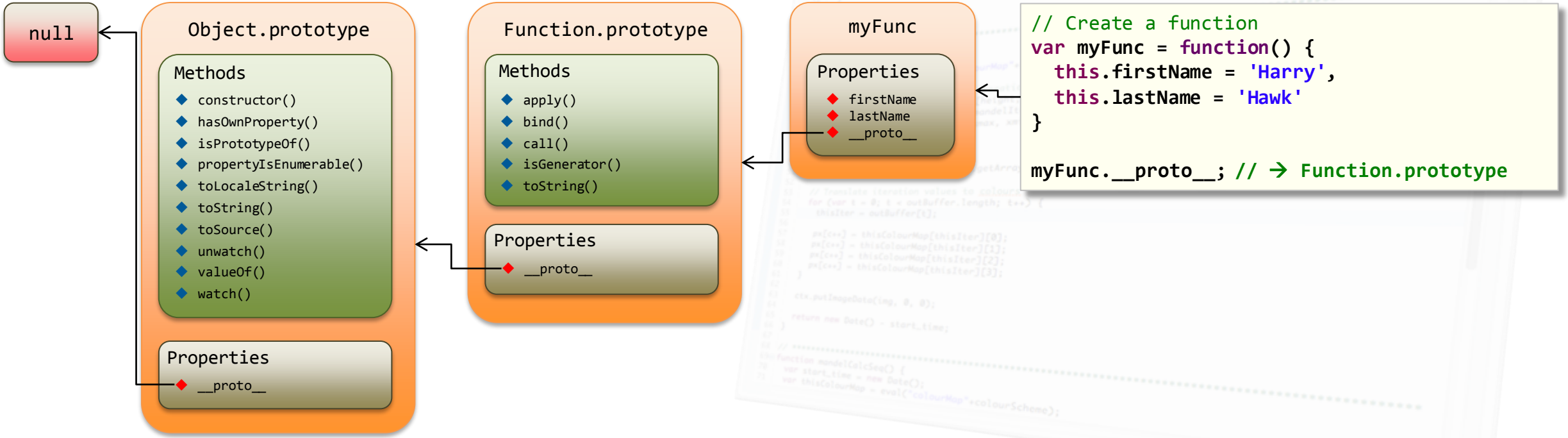
Similarly, all `Function` objects inherit their basic properties from `Function.prototype`.



Understanding the Prototype Chain 2/2

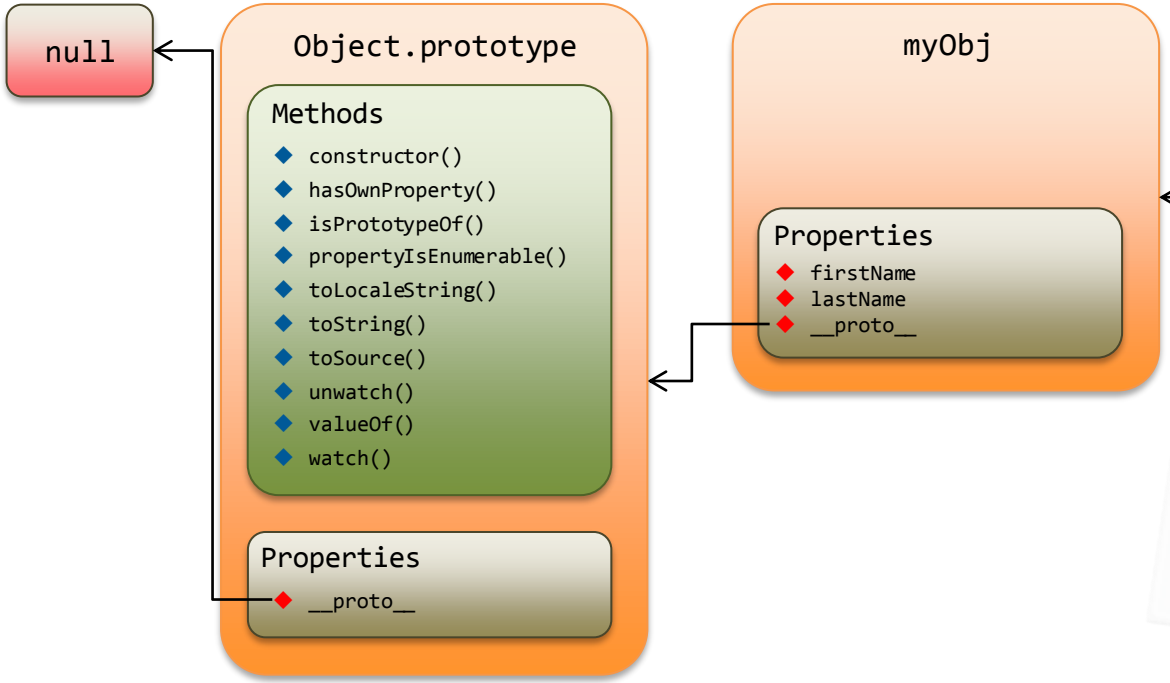
Similarly, all `Function` objects inherit their basic properties from `Function.prototype`.

Since `Function.prototype` is also an object, it inherits from `Object.prototype`



Understanding the Prototype Chain: Property Shadowing

A child object can override any property inherited from its prototype simply by redefining it. This is known as “property shadowing”.

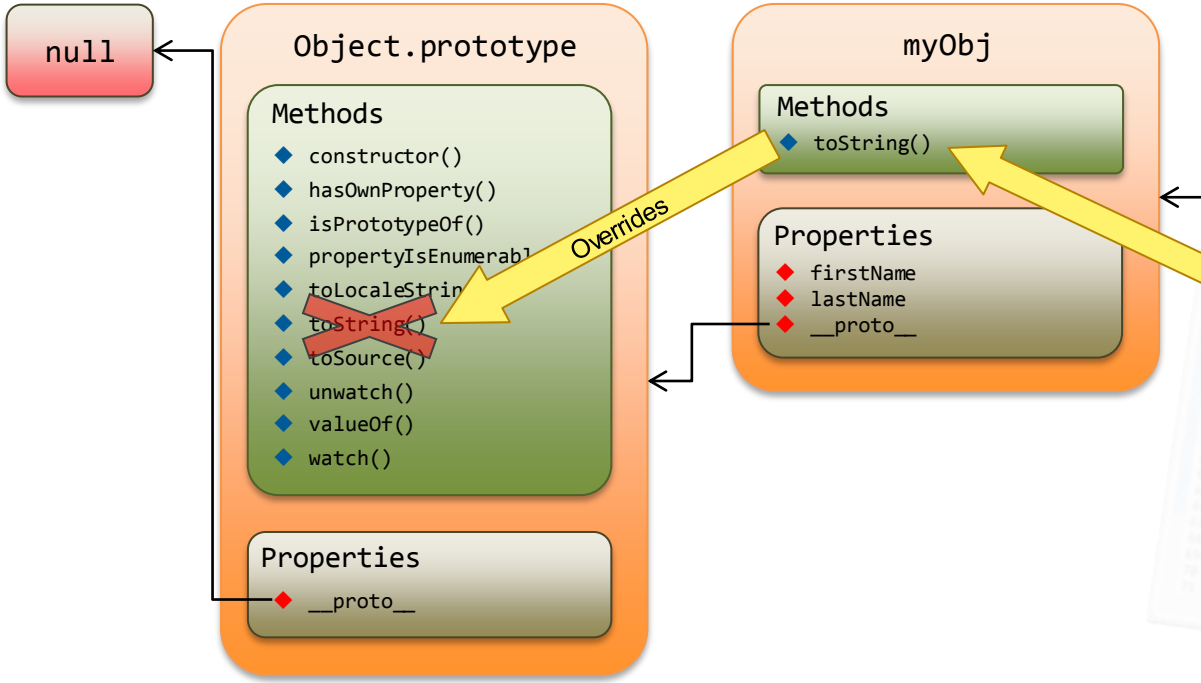



```
// Create an object that overrides a
// standard method in Object.prototype
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk',




}
```

Understanding the Prototype Chain: Property Shadowing

A child object can override any property inherited from its prototype simply by redefining it. This is known as “property shadowing”.

For instance, this technique is often used to customise the behaviour of the `toString()` method.

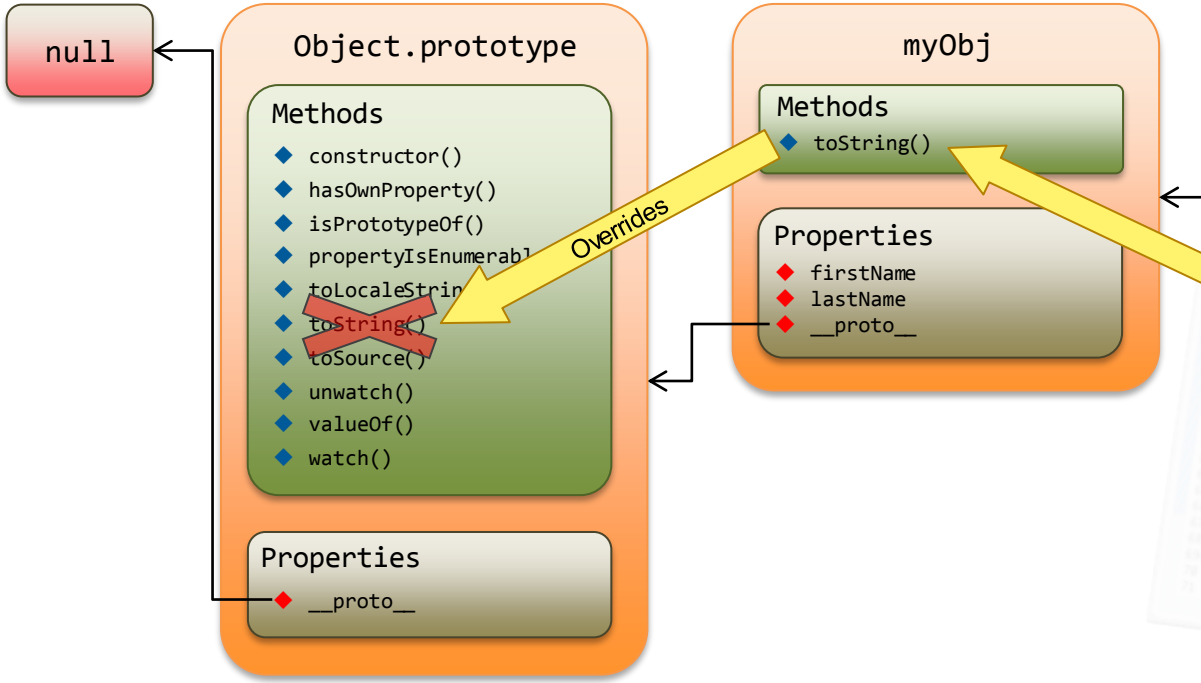



```
// Create an object that overrides a
// standard method in Object.prototype
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk',

  toString : function() {
    return "My name is " +
           this.firstName + " " +
           this.lastName;
  }
}
```

Understanding the Prototype Chain: Property Shadowing

A child object can override any property inherited from its prototype simply by redefining it. This is known as “property shadowing”.

For instance, this technique is often used to customise the behaviour of the `toString()` method.

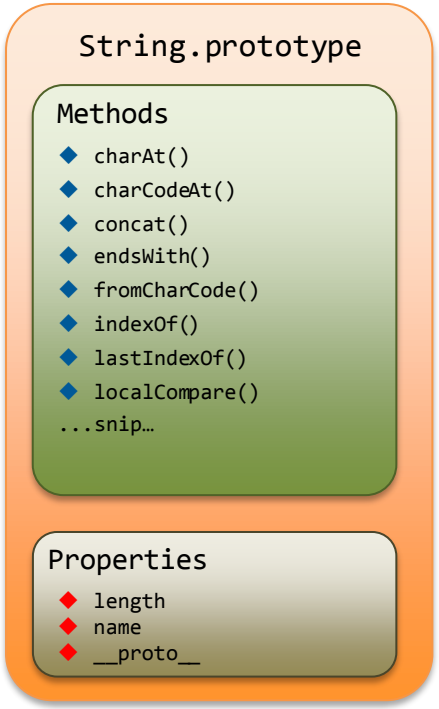



```
// Create an object that overrides a
// standard method in Object.prototype
var myObj = {
  firstName : 'Harry',
  lastName : 'Hawk',

  toString : function() {
    return "My name is " +
           this.firstName + " " +
           this.lastName;
  }
}
```

Property shadowing is a temporary, local effect within the referencing object and changes neither the prototype object nor any other object that references the prototype.

Understanding the Prototype Chain: Extending a Prototype

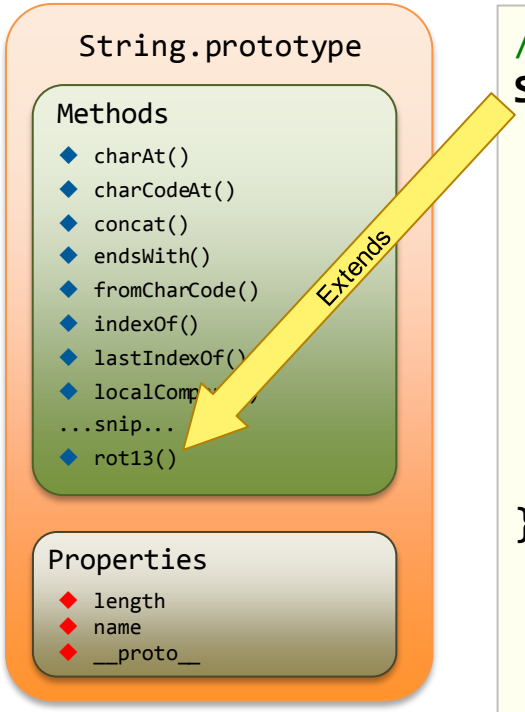
For any object used as a prototype, all referencing objects will immediately inherit any new functionality added to that prototype. E.G. the standard JavaScript String object has no `rot13()` method, but one can easily be added by extending the `String.prototype` object.




```
// Extend the standard String prototype to include a rot13() method
String.prototype.rot13 = function() {
  var from = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  var to   = "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm";

  function rot13Char(c) {
    return (from.indexOf(c) > -1) ? to.charAt(from.indexOf(c)) : c;
  }

  return this.split('').map(rot13Char).join('');
}
```

Understanding the Prototype Chain: Extending a Prototype

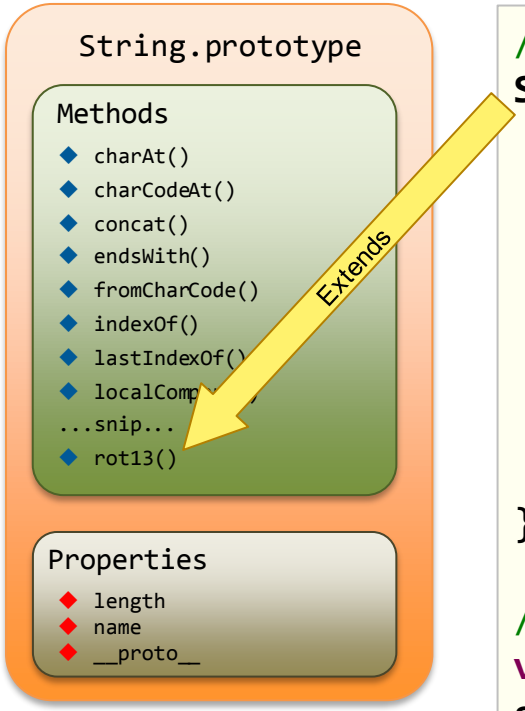
For any object used as a prototype, all referencing objects will immediately inherit any new functionality added to that prototype. E.G. the standard JavaScript String object has no `rot13()` method, but one can easily be added by extending the `String.prototype` object.




```
// Extend the standard String prototype to include a rot13() method
String.prototype.rot13 = function() {
  var from = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  var to   = "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm";

  function rot13Char(c) {
    return (from.indexOf(c) > -1) ? to.charAt(from.indexOf(c)) : c;
  }

  return this.split('').map(rot13Char).join('');
}
```

Understanding the Prototype Chain: Extending a Prototype

For any object used as a prototype, all referencing objects will immediately inherit any new functionality added to that prototype. E.G. the standard JavaScript String object has no `rot13()` method, but one can easily be added by extending the `String.prototype` object.




```
// Extend the standard String prototype to include a rot13() method
String.prototype.rot13 = function() {
  var from = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
  var to   = "NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm";

  function rot13Char(c) {
    return (from.indexOf(c) > -1) ? to.charAt(from.indexOf(c)) : c;
  }

  return this.split('').map(rot13Char).join('');
}

// Now all string objects immediately inherit a rot13() method
var someStr = "Hi there!";
someStr.rot13();    // → "Uv gurer!"
```

Defining Your Own Prototypes

Defining Your Own Prototype Objects

It is often beneficial to define your own prototype objects. For instance, rather than using a function to explicitly define all the properties of a person, we can define a `Person.prototype` object whose properties are inherited every time a person object is created.

```
// This function creates a person object, but the property definitions are duplicated for each instance
function Person(fName, lName, dob) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
};
```

Defining Your Own Prototype Objects

It is often beneficial to define your own prototype objects. For instance, rather than using a function to explicitly define all the properties of a person, we can define a `Person.prototype` object whose properties are inherited every time a person object is created.

```
// This function creates a person object, but the property definitions are duplicated for each instance
function Person(fName, lName, dob) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
};

// Move the definition of the default properties into a Person.prototype object
Person.prototype = {
  this.firstName : "",
  this.lastName  : "",
  this.dateOfBirth : null
}
```

Defining Your Own Prototype Objects

It is often beneficial to define your own prototype objects. For instance, rather than using a function to explicitly define all the properties of a person, we can define a `Person.prototype` object whose properties are inherited every time a person object is created.

```
// This function creates a person object, but the property definitions are duplicated for each instance
function Person(fName, lName, dob) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
};

// Move the definition of the default properties into a Person.prototype object
Person.prototype = {
  this.firstName : "",
  this.lastName  : "",
  this.dateOfBirth : null
}
```

Not only does this improve code reuse, but it reduces memory usage because the definition of the properties common to all person objects is stored once in the prototype, not replicated in every individual person object.

Assigning Prototypes to Objects

Now that we have assigned the common properties of a person to the `Person.prototype` object, we need to ensure that every time an object of type person is created, it inherits from this prototype.

```
// Create a function whose job is to initialise the properties of a new object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}
```

Assigning Prototypes to Objects

Now that we have assigned the common properties of a person to the `Person.prototype` object, we need to ensure that every time an object of type person is created, it inherits from this prototype.

JavaScript gives us a simple way to do this – the **`new`** operator.

```
// Create a function whose job is to initialise the properties of a new object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Any function invoked using the 'new' operator is known as a "constructor function"
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

Assigning Prototypes to Objects

Now that we have assigned the common properties of a person to the `Person.prototype` object, we need to ensure that every time an object of type person is created, it inherits from this prototype.

JavaScript gives us a simple way to do this – the **`new`** operator.

```
// Create a function whose job is to initialise the properties of a new object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Any function invoked using the 'new' operator is known as a "constructor function"
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

Important

Whenever a function is invoked using the **`new`** operator, the function call behaves differently compared to a normal function call.

JavaScript Constructor Functions

There are two important things to notice here:

1. By convention, the name of a constructor function should start with a capital letter

```
// Constructor function to initialise a new person object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Invoke the constructor function using the new operator
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

JavaScript Constructor Functions

There are two important things to notice here:

1. By convention, the name of a constructor function should start with a capital letter
2. Functions invoked using the **new** operator typically do not use the **return** statement

```
// Constructor function to initialise a new person object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Invoke the constructor function using the new operator
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

JavaScript Constructor Functions

There are two important things to notice here:

1. By convention, the name of a constructor function should start with a capital letter
2. Functions invoked using the **new** operator typically do not use the **return** statement

```
// Constructor function to initialise a new person object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Invoke the constructor function using the new operator
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

The **new** keyword causes a constructor function to be invoked differently compared to a normal function, in that:

1. A new Person object is created that automatically inherits from `Person.prototype`

JavaScript Constructor Functions

There are two important things to notice here:

1. By convention, the name of a constructor function should start with a capital letter
2. Functions invoked using the **new** operator typically do not use the **return** statement

```
// Constructor function to initialise a new person object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Invoke the constructor function using the new operator
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

The **new** keyword causes a constructor function to be invoked differently compared to a normal function, in that:

1. A new Person object is created that automatically inherits from `Person.prototype`
2. The newly created object is bound to **this**. (I.E. The newly created object becomes the execution context)

JavaScript Constructor Functions

There are two important things to notice here:

1. By convention, the name of a constructor function should start with a capital letter
2. Functions invoked using the **new** operator typically do not use the **return** statement

```
// Constructor function to initialise a new person object
function Person(fName, lName, doB) {
  this.firstName = fName || "";
  this.lastName  = lName || "";
  this.dateOfBirth = doB;
}

// Invoke the constructor function using the new operator
var someGuy = new Person("Harry","Hawk",new Date(76,08,03));
```

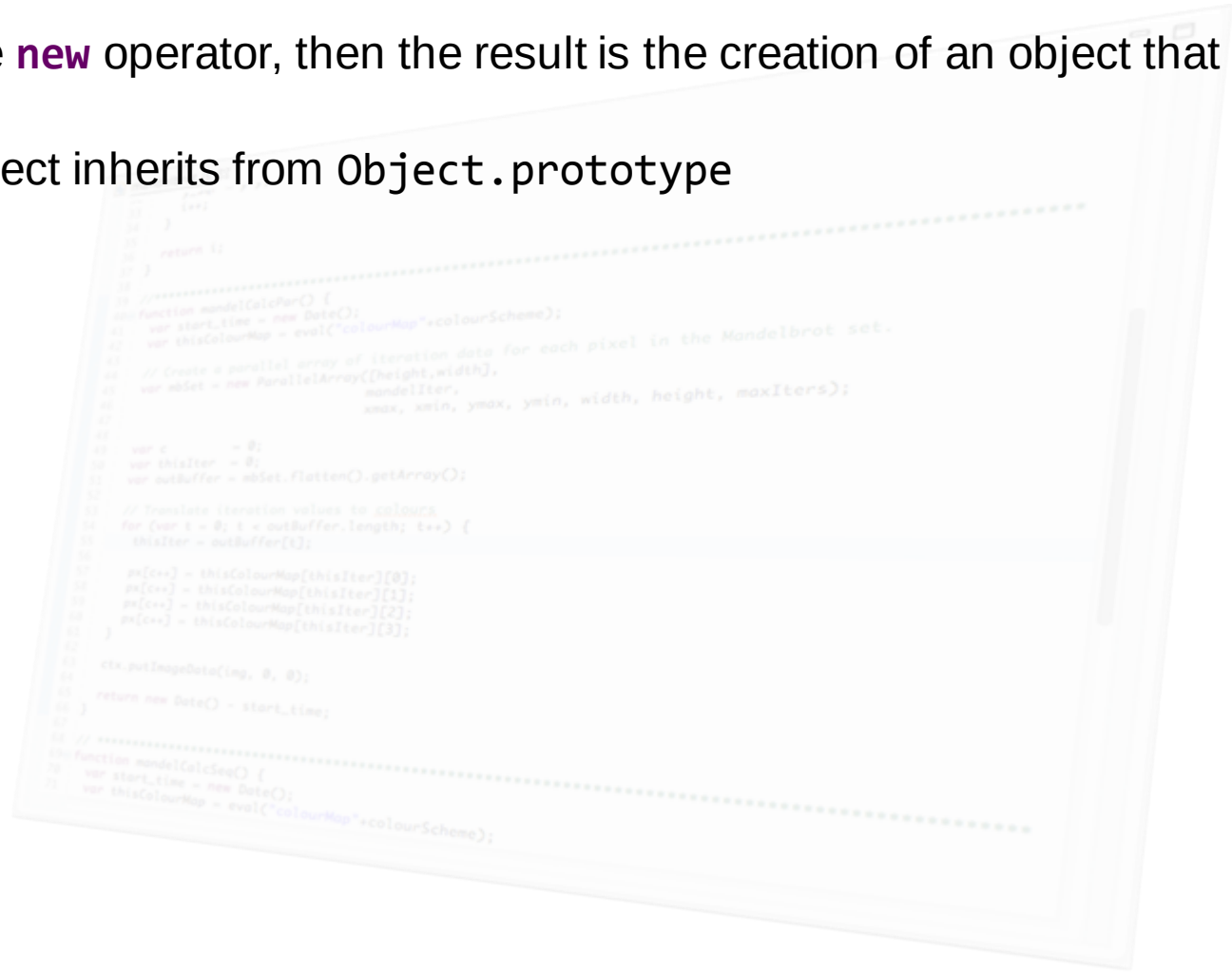
The **new** keyword causes a constructor function to be invoked differently compared to a normal function, in that:

1. A new Person object is created that automatically inherits from `Person.prototype`
2. The newly created object is bound to **this**. (I.E. The newly created object becomes the execution context)
3. If the **return** statement is not used, then the value of the constructor function is ***always*** a reference to the object created in step 1

The `new` Operator and the Prototype Chain

In general, if function `<ObjName>` is invoked using the `new` operator, then the result is the creation of an object that automatically inherits from `<ObjName>.prototype`.

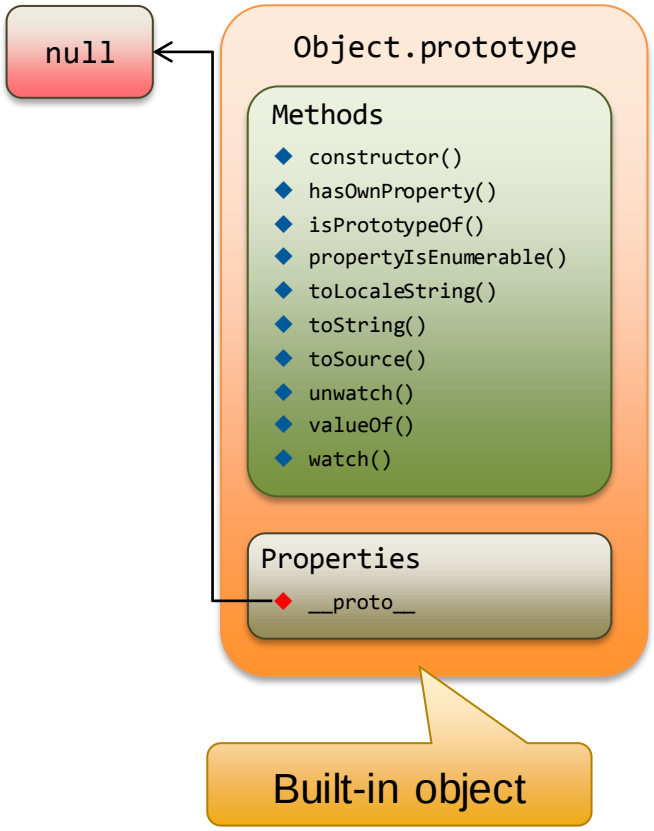
If `<ObjName>.prototype` does not exist, then the object inherits from `Object.prototype`



The `new` Operator and the Prototype Chain

In general, if function `<ObjName>` is invoked using the `new` operator, then the result is the creation of an object that automatically inherits from `<ObjName>.prototype`.

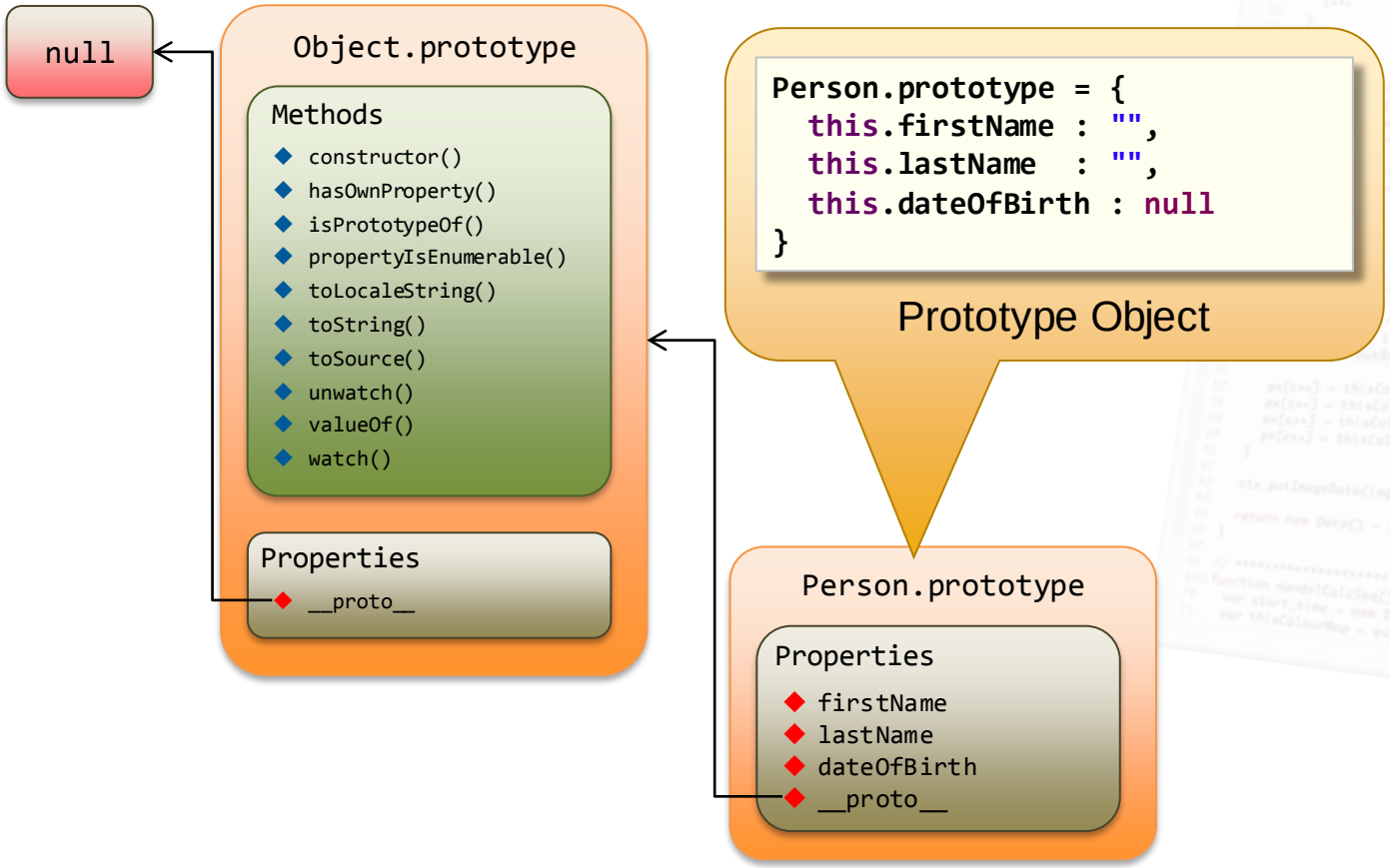
If `<ObjName>.prototype` does not exist, then the object inherits from `Object.prototype`



The new Operator and the Prototype Chain

In general, if function <ObjName> is invoked using the new operator, then the result is the creation of an object that automatically inherits from <ObjName>.prototype.

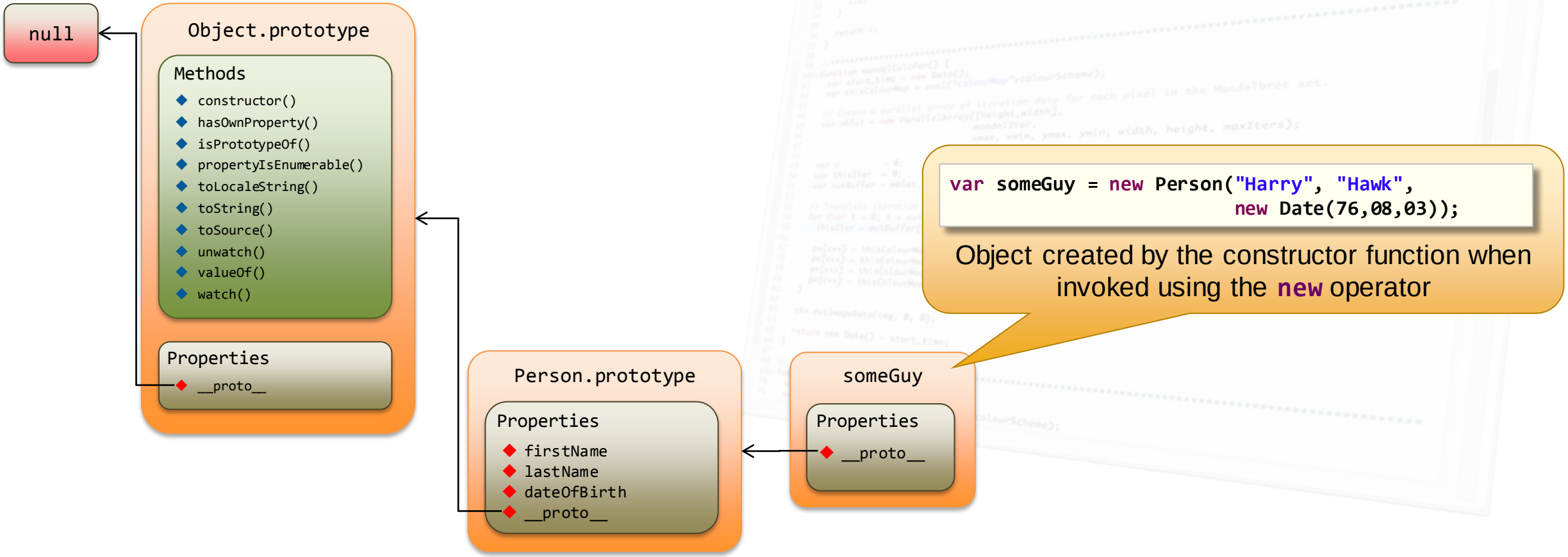
If <ObjName>.prototype does not exist, then the object inherits from Object.prototype



The new Operator and the Prototype Chain

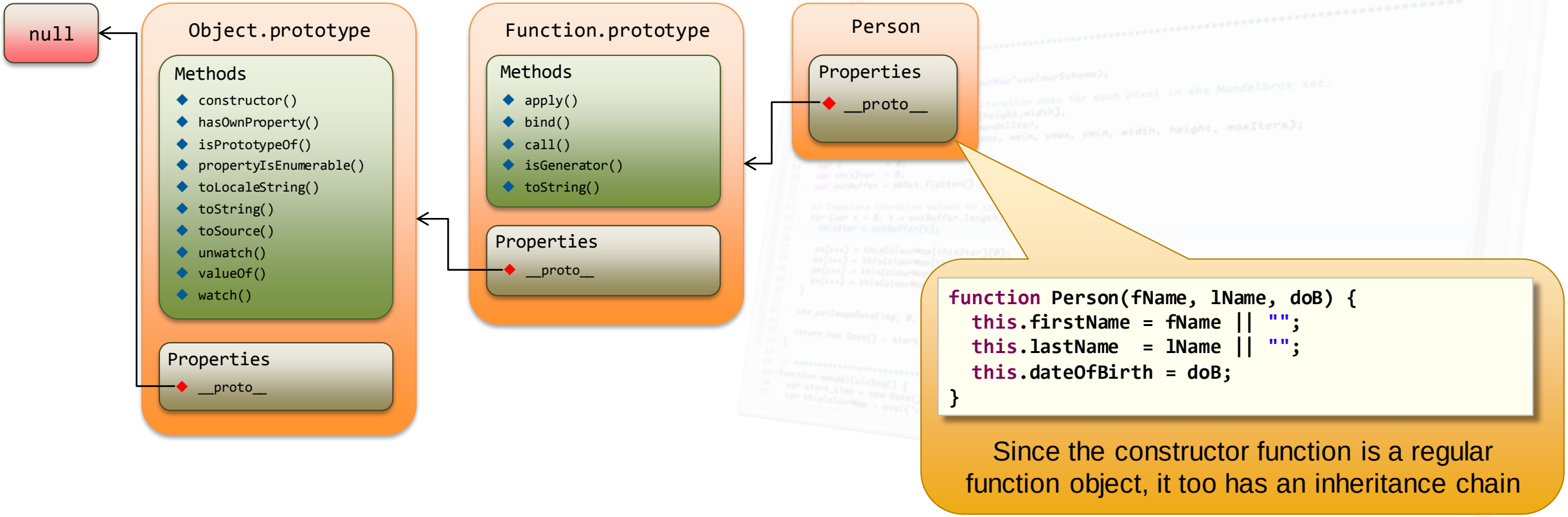
In general, if function <ObjName> is invoked using the new operator, then the result is the creation of an object that automatically inherits from <ObjName>.prototype.

If <ObjName>.prototype does not exist, then the object inherits from Object.prototype



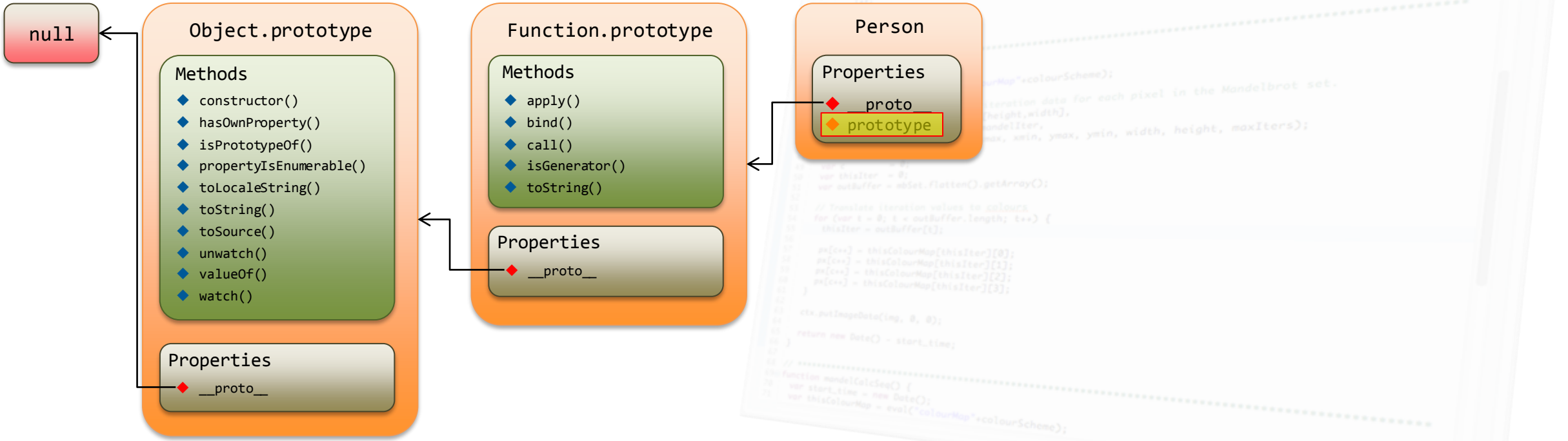
Constructor Functions and the Prototype Chain 1/3

A potential source of confusion in JavaScript inheritance is the fact that a constructor function is itself a function object. Therefore, like any other function object, has its own inheritance chain via its `__proto__` property.



Constructor Functions and the Prototype Chain 2/3

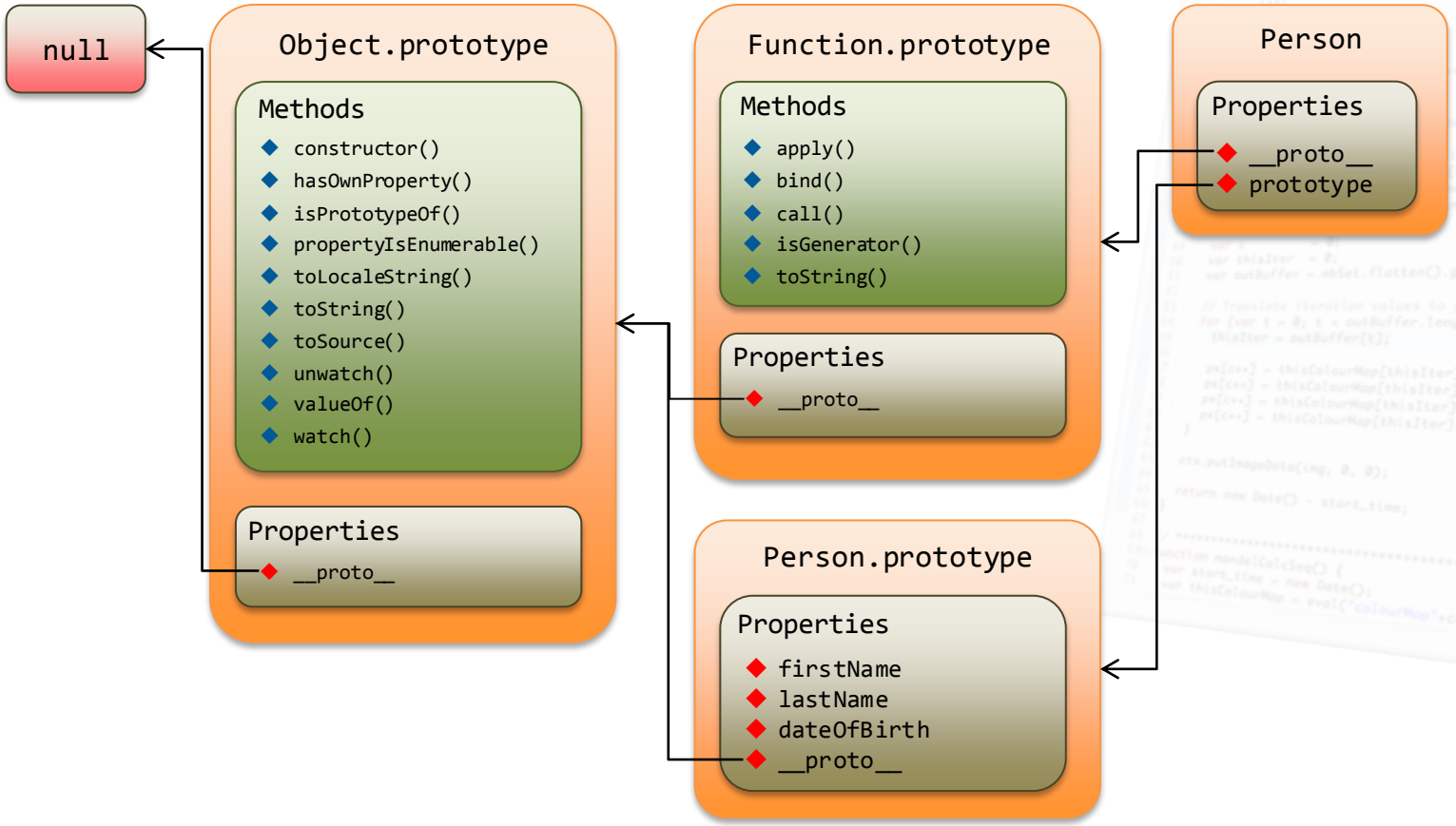
In addition to `__proto__`, all constructor functions have a second default property called `prototype`.



Constructor Functions and the Prototype Chain 2/3

In addition to `__proto__`, all constructor functions have a second default property called `prototype`.

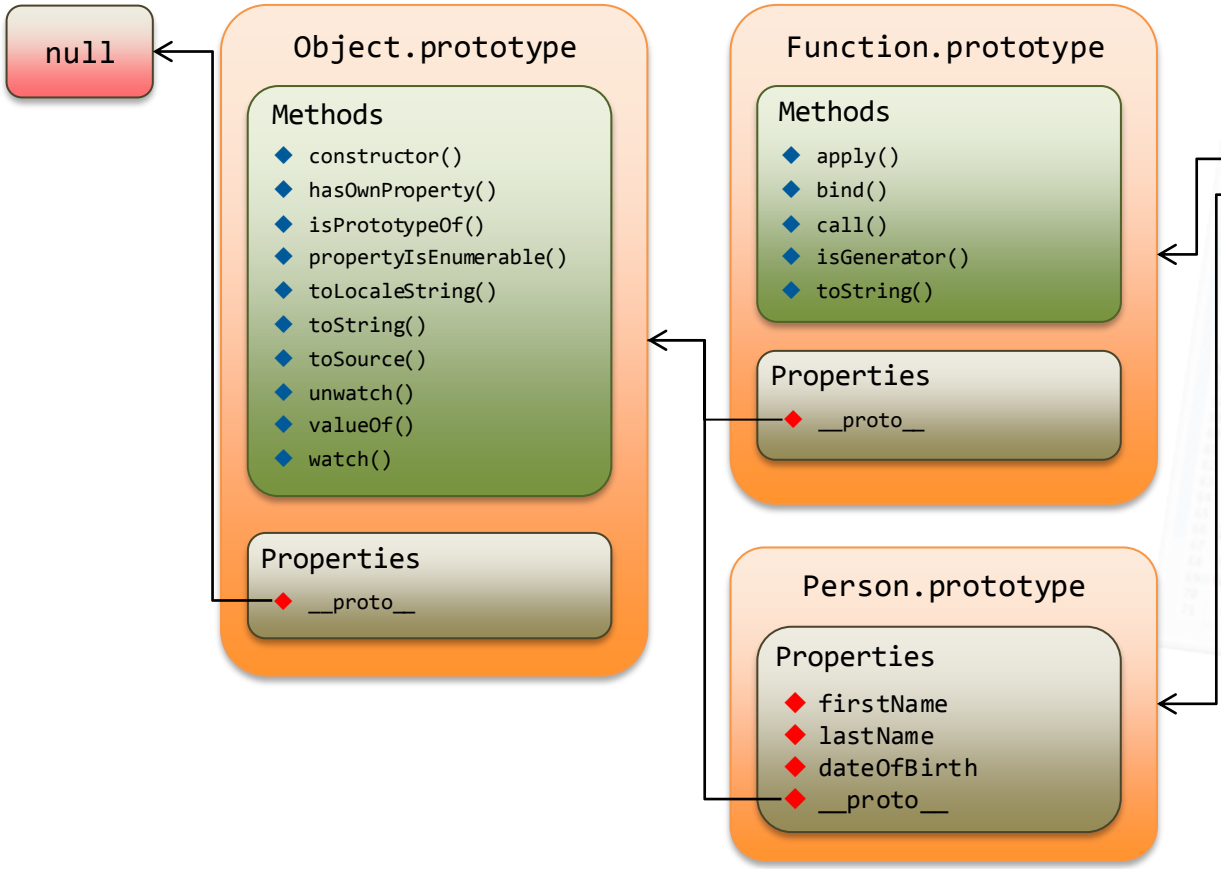
If a constructor function is invoked using the **`new`** operator, then the prototype property holds a reference to the object prototype.



Constructor Functions and the Prototype Chain 2/3

In addition to `__proto__`, all constructor functions have a second default property called prototype.

If a constructor function is invoked using the `new` operator, then the prototype property holds a reference to the object prototype.

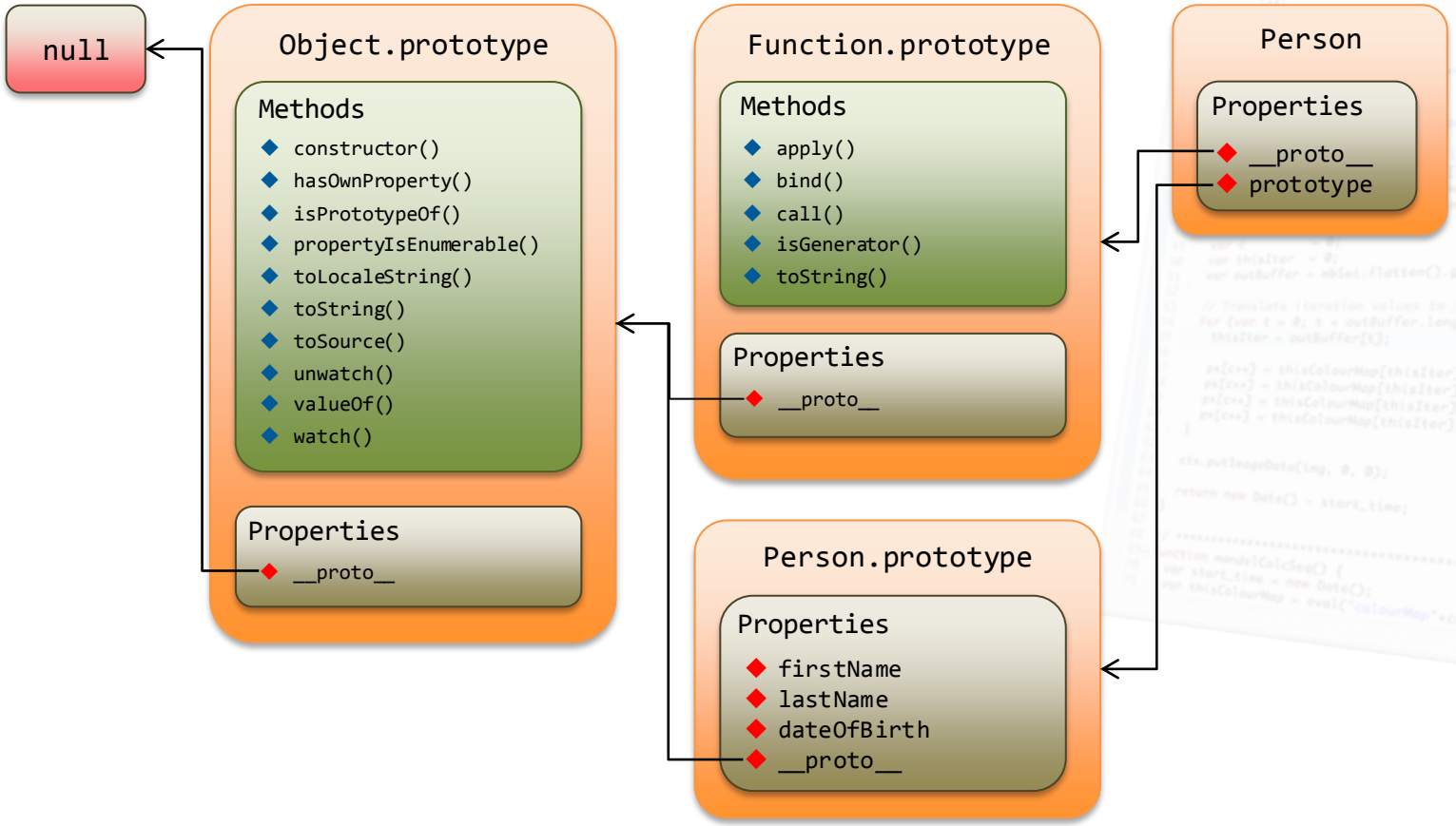


So in the case of our `Person` constructor function, by virtue of the fact that `Person` is just a regular function object, its `__proto__` property points to `Function.prototype`.

However, as a constructor function, its job is to create an instance of a new `Person` object, which in turn, must inherit from its own prototype; therefore, the prototype property points to `Person.prototype`.

Constructor Functions and the Prototype Chain 3/3

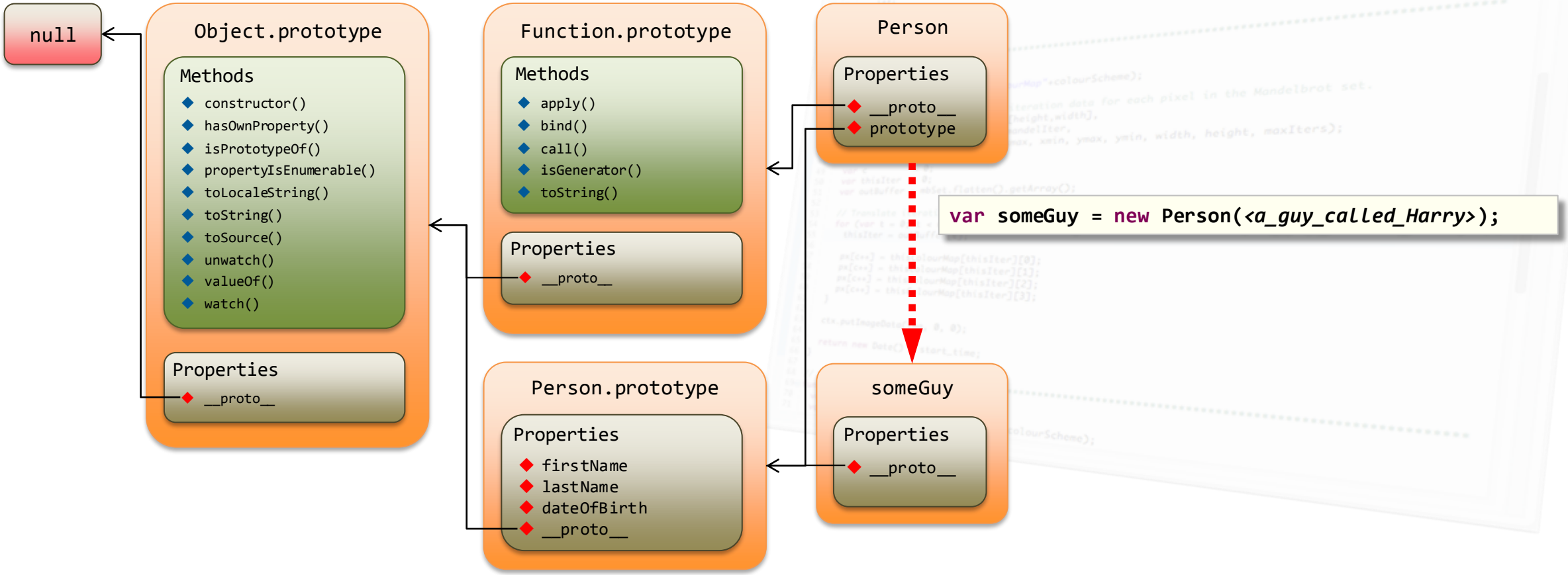
To complete the picture here, we can finally add the object created when the `new` operator is used to invoke the `Person` constructor function.



Constructor Functions and the Prototype Chain 3/3

To complete the picture here, we can finally add the object created when the `new` operator is used to invoke the `Person` constructor function.

Multiple objects of type `Person` can now be created that will all inherit from `Person.prototype`.



Constructor Functions and the Prototype Chain 3/3

To complete the picture here, we can finally add the object created when the `new` operator is used to invoke the `Person` constructor function.

Multiple objects of type `Person` can now be created that will all inherit from `Person.prototype`.