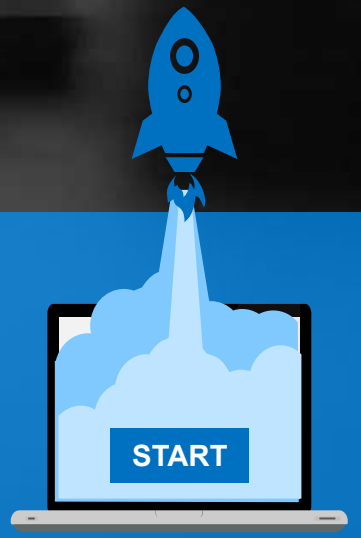
BIENVENIDOS

lsftn151 | AS | AYED II



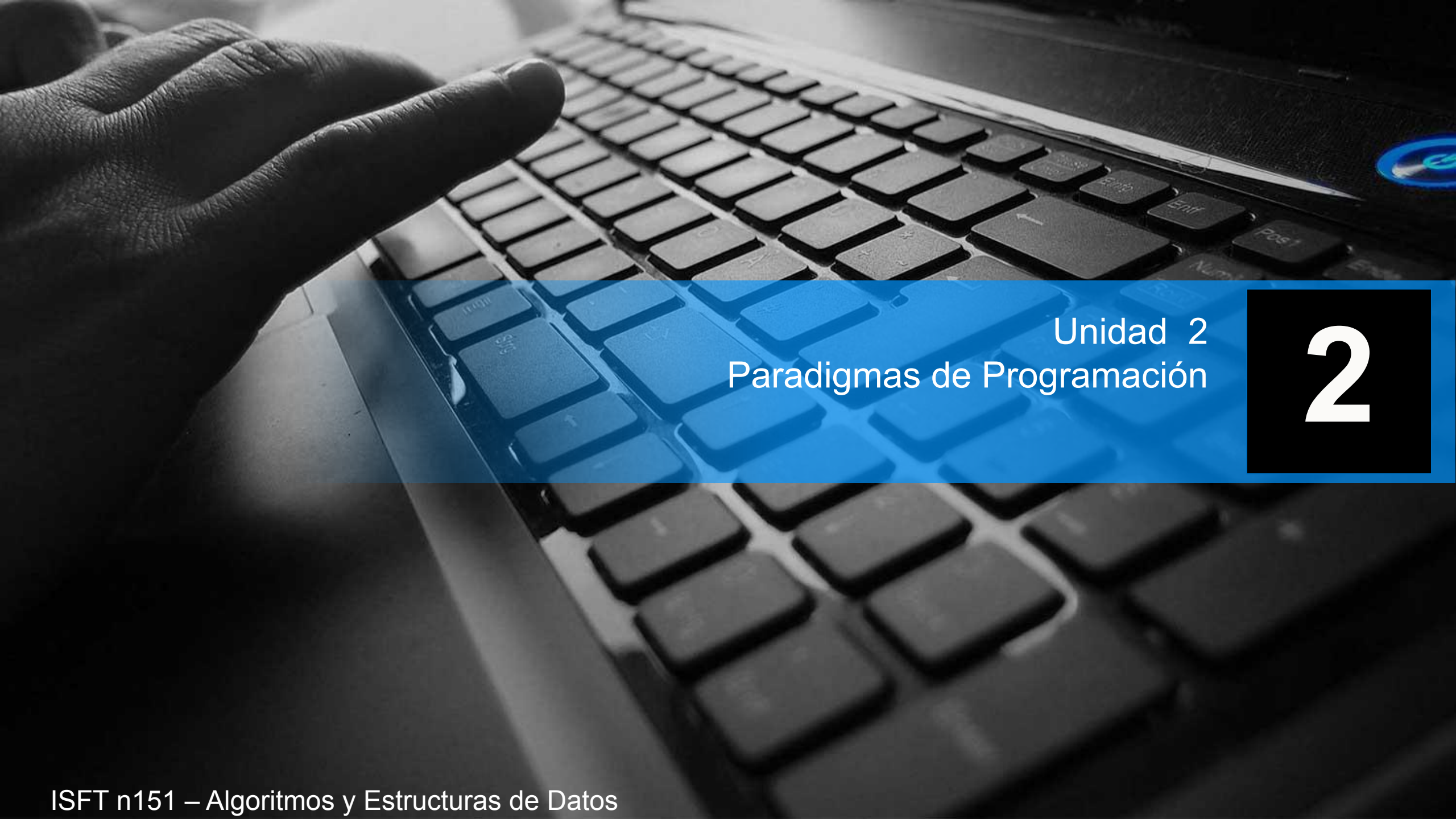
Un Momento por favor.

Bienvenidos

Hoy Comenzaremos este Tramo de formación juntos.
Sabemos que no será sencillo, pero material de sobra tenemos.

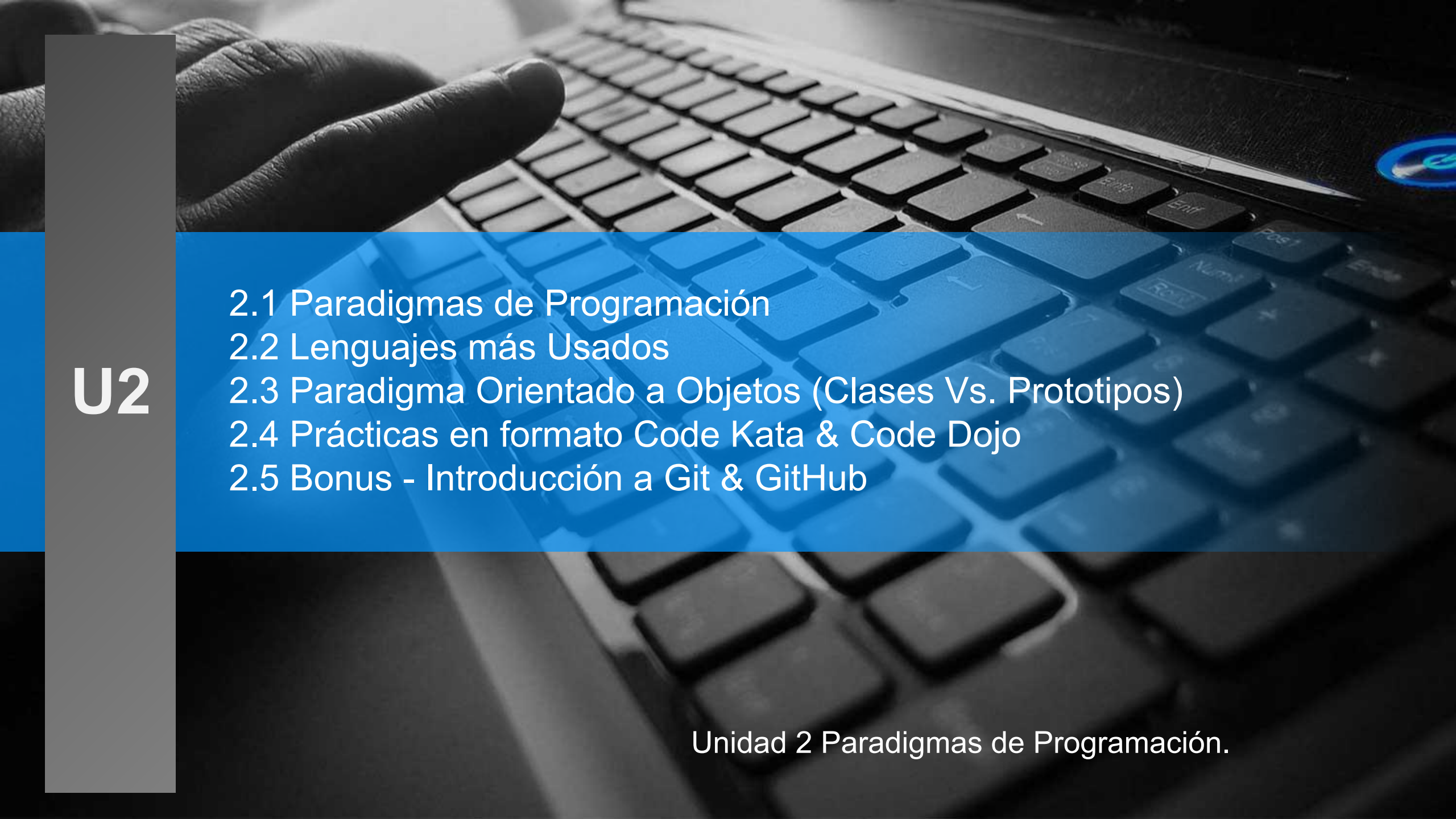
Vamos a poner a vuestra disposición una batería de conocimientos que sin dudarlo, les servirán para desempeñarse en el futuro laboral.

Acompañennos en este tramo poniendo lo mejor de Ustedes… para Ustedes.



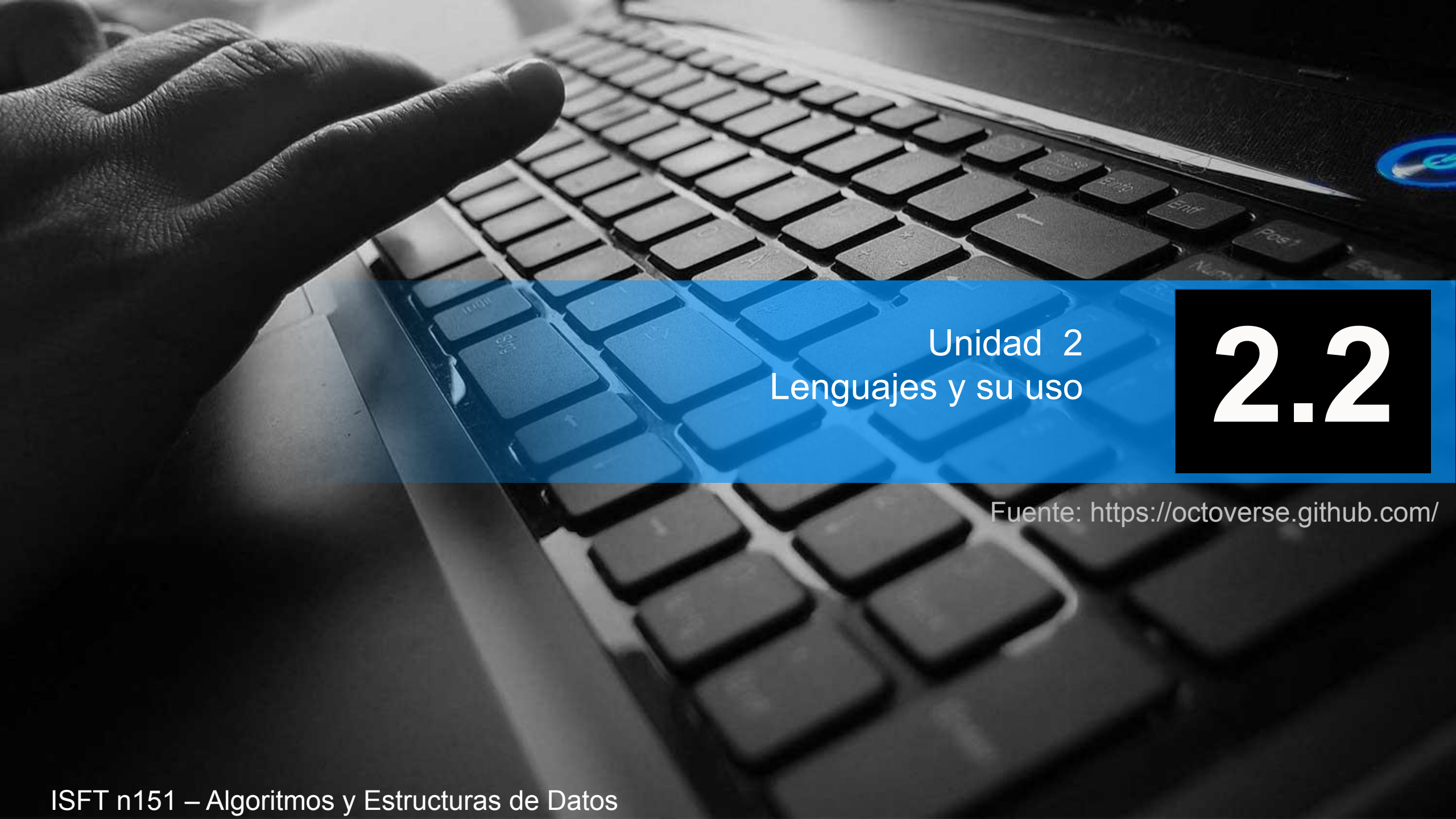
Unidad 2
Paradigmas de Programación

2



U2

2.1 Paradigmas de Programación
2.2 Lenguajes más Usados
2.3 Paradigma Orientado a Objetos (Clases Vs. Prototipos)
2.4 Prácticas en formato Code Kata & Code Dojo
2.5 Bonus - Introducción a Git & GitHub



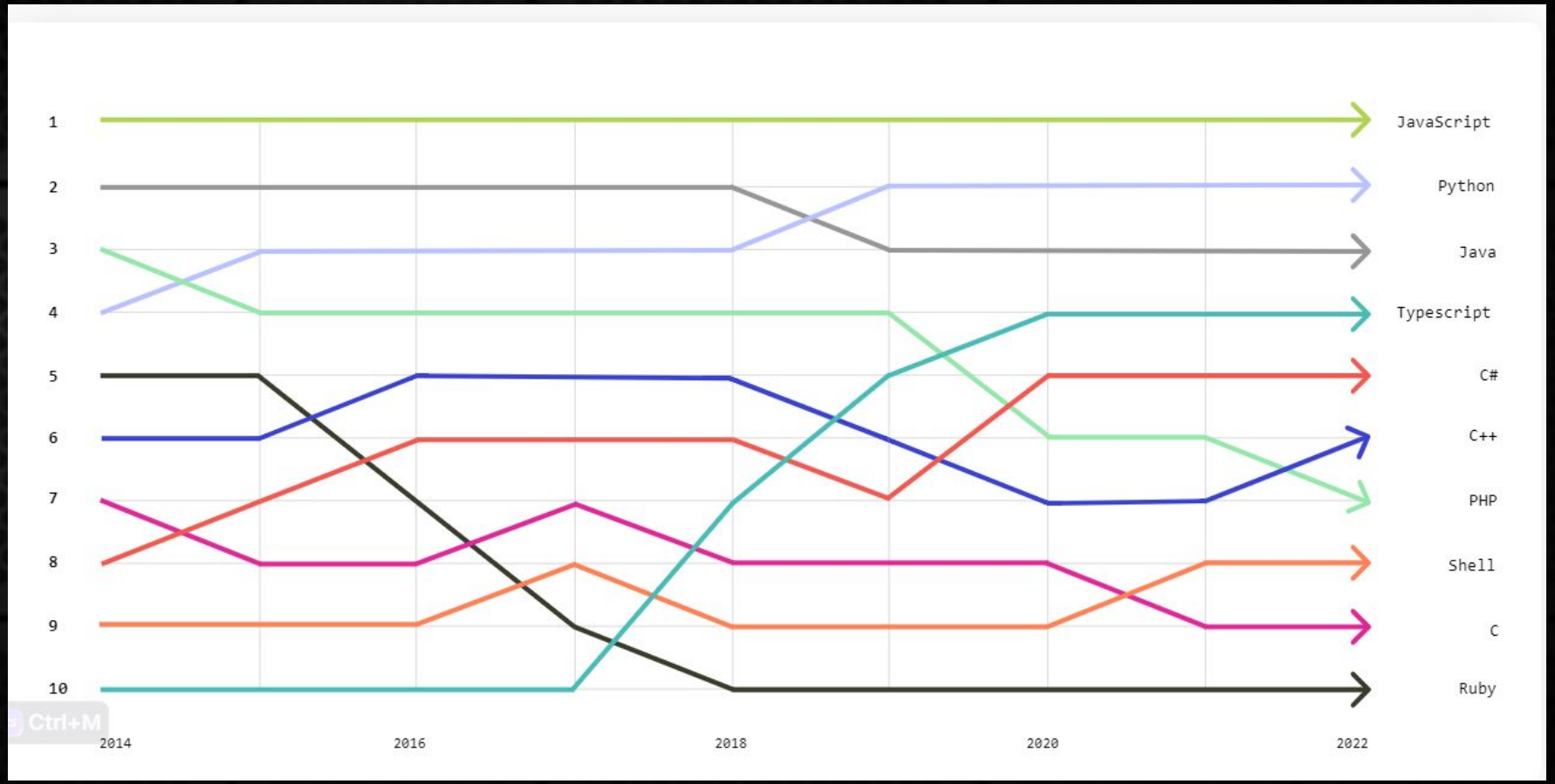
Unidad 2
Lenguajes y su uso

2.2

Fuente: https://octoverse.github.com/

Los Lenguajes mas Activos en GitHub.

Gracias al **Octoverse 2022**, un reporte que lanza GitHub cada año, podemos ver la gran actividad que tuvo la comunidad de desarrollo a nivel mundial. Y qué mejor forma de observar los lenguajes de programación más utilizados, si no es desde los repositorios de su código.
Con datos de más de 56 millones de programadores alrededor del mundo, estos son algunos de los lenguajes de programación más usados en la actualidad:



JAVASCRIPT

Este lenguaje de programación orientado a objetos, de prototipado y multiparadigma es interpretado, es decir, no requiere de compilación, ya que está pensado para correr en el navegador. Y cabe mencionar que JavaScript está soportado por una inmensa comunidad.
Es el lenguaje de programación web más utilizado en todo el mundo, ya que prácticamente todos los sitios web —incluso si fueron construidos con otro lenguaje—, tienen algo de JavaScript en ellos. Por eso su gran relevancia cuando hablamos del front-end.
Además, empresas de gran talla han dirigido esfuerzos creando frameworks como ReactJS de Facebook, AngularJS de Google o NodeJS, que permiten que el código escrito en JavaScript corra en el lado del servidor, es decir, pensado para el back-end. Por eso es uno de los lenguajes de programación más demandados para el 2021.

Python

Creado por Guido Van Rossum en la década de los 90, este lenguaje multiparadigma y multipropósito fue concebido como un “side-project” al igual que muchas tecnologías de esta lista de lenguajes de programación más usados.
Python ha ganado mucha relevancia en industrias muy populares como la Inteligencia Artificial, el Machine Learning, Big Data, entre otras.

Java

Java es uno de los lenguajes de programación más disruptivos de la historia, ya que en los 90 muchas empresas desarrollaban sus herramientas principalmente en C++, el cual era complicado y dependiente de la plataforma en la que este se desarrollara.
James Gosling y su equipo crearon una tecnología que prometía ser más fácil de aprender. Por eso es uno de los lenguajes de programación más utilizados en la actualidad.
Este es un lenguaje de programación orientado a objetos e independiente de su plataforma, por lo que el código que ha sido escrito en una máquina también correrá en otra, incluso, con sistemas operativos distintos gracias a la Máquina Virtual Java (o JVM por sus siglas en inglés).

Lenguaje de programación C#

Creado en 1999 por Anders Hellberg cuando laboraba en Microsoft, este lenguaje de programación posee características como C, pero orientado a objetos.
C# fue muy criticado, ya que era muy similar a Java, aunque con el tiempo tomaron caminos distintos. En la actualidad, es el quinto lenguaje de programación multiparadigma más usado, según el índice TIOBE.
Este lenguaje de programación es muy utilizado en la industria del gaming, la robótica, la impresión 3D, los controladores y las aplicaciones de escritorio no sólo en Windows, sino también en iOS y Android, gracias a Xamarin.

TypeScript

El crecimiento de TypeScript fue exponencial en 2020, ya que los estudios de Stack Overflow de inicios de ese año mostraron que este lenguaje —relativamente nuevo—, estaría situado en los últimos lugares de popularidad, pero al final del año se encontró dentro de los primeros 5 lenguajes de programación más utilizados en el mundo.
La razón de su aceptación es que es fácil para los desarrolladores escribir y mantener códigos. Empresas como Google, Slack, Medium y Accenture utilizan este lenguaje de programación.
Este lenguaje compila en JavaScript nativo y se convierte en código de TypeScript, pasando por los mismos procesos de JS sin que el navegador 'se entere' de que esto sucedió.
Además, este lenguaje de programación ofrece una descripción completa de cada componente del código y se puede utilizar para desarrollar grandes aplicaciones con una sintaxis estricta y menos errores
Esta tecnología creada en el 2012 por Microsoft tiene muchas funcionalidades o mecanismos de la programación orientada a objetos haciendo que cualquier aplicación o sitio construido con este lenguaje de programación sea más escalable.

PHP
Este gran lenguaje de programación multipropósito fue creado en la década de los 90, pensado inicialmente como una Interfaz de Entrada Común (o CGI) por el groenlandés Rasmus Lerdorf, el cual lo utilizaba para mantener su propio sitio web (de ahí el nombre de este lenguaje: "personal homepage").
Desde entonces, PHP fue evolucionando hasta convertirse en el lenguaje de programación que es hoy en día, utilizado principalmente para desarrollar aplicaciones en el lado del servidor, garantizando páginas web estables y con buen rendimiento

Xamarin

Xamarin es una plataforma de desarrollo que te permite escribir aplicaciones multiplataforma—aunque nativa—para iOS, Android y Windows Phone en C# y .NET. Xamarin proporciona bindings C# a las API nativas Android y iOS. Esto te da el poder para usar toda la interfaz de usuario nativo, notificaciones, gráficos, animación y otras características de teléfono— y todas usan C#.
Xamarin alcanza cada lanzamiento nuevo de Android y iOS, con un lanzamiento que incluye bindings para sus nuevas API. El puerto de .NET de Xamarin incluye características como tipos de data, genéricos, colección de papelera de reciclaje, language-integrated query (LINQ), patrón de programación asincrónica, delegación y un subconjunto de Windows Communication Foundation (WCF). Las bibliotecas son manejadas con un linger para incluir sólo los componentes referidos.
Xamarin.Forms es una capa que se encuentra sobre las otros bindings UI y la API Windows Phone, la cual proporciona una biblioteca de interfaz de usuario completamente multiplataforma.

C++

Este lenguaje de programación multiparadigma vio la luz en la década del 70, y como te imaginarás, está fuertemente influenciado y basado en C, con la finalidad de agregarle funcionalidades de orientación a objetos.
Al igual que su predecesor, este lenguaje de programación compila directamente a instrucciones de máquina y ofrece acceso completo al hardware, pero de una manera más compleja. Se utiliza ampliamente en bases de datos, navegadores web, compiladores o videojuegos.

Lenguaje de programación C

Compiladores como el de Python y PHP están escritos en C; esto nos habla de la gran relevancia que ha tenido para nuestra época, pues de este han salido diversos tipos de lenguajes de programación.
Este lenguaje de procedimientos y de propósito general, es uno de los primeros lenguajes de programación y ha servido inspiración para otros lenguajes como lo vimos con C++, que también ofrece acceso directo a la memoria y al hardware de las computadoras.Múltiples sistemas operativos y herramientas que utilizamos en la actualidad han sido construidos con este lenguaje de programación de alto nivel.
Sin embargo, el código escrito en C no es portable, ocasionando que todo lo que programas para algún sistema operativo pueda ser ejecutado sólo en esa máquina. Esto es una desventaja frente a otras tecnologías de esta lista de lenguajes de programación y, por tal motivo, lo vemos en el octavo sitio.

Ruby

Ruby fue creado en 1995 por Yukihiro Matsumoto como un lenguaje de scripting orientado a objetos, este lenguaje de programación de alto nivel eventualmente evolucionó a un lenguaje interpretado con tipado dinámico. Ruby es multiparadigma y multipropósito, y se encuentra implementado en el lenguaje de programación C.
Al igual que Python, Ruby se encuentra enfocado en la productividad del desarrollador y si tu intención es aprender a programar, esta puede ser una muy buena opción debido a su sencillez.
Este lenguaje de programación ha sido utilizado en proyectos como Twitter, GitHub y Airbnb, y posee una gran cantidad de herramientas y quizás el framework más disruptivo que se utiliza para desarrollar en el lado del servidor: Ruby on Rails.

Lenguaje de programación Go (Goland)

Es una tecnología creada por Google en 2012 y fue desarrollada para hacer que el uso e implementaciones del hardware de esta compañía fueran prácticamente ilimitadas.
Con este lenguaje de programación querían resolver un problema que les generaba C++ (que también es aplicado en hardware), ya que la compilación del código fuente de algunos programas tomaba alrededor de 30 minutos.
Go soporta concurrencia combinando la simpleza y productividad de Python junto al poder de C. Sin duda, es uno de los ejemplos de lenguajes de programación con muchísimo futuro.

Swift

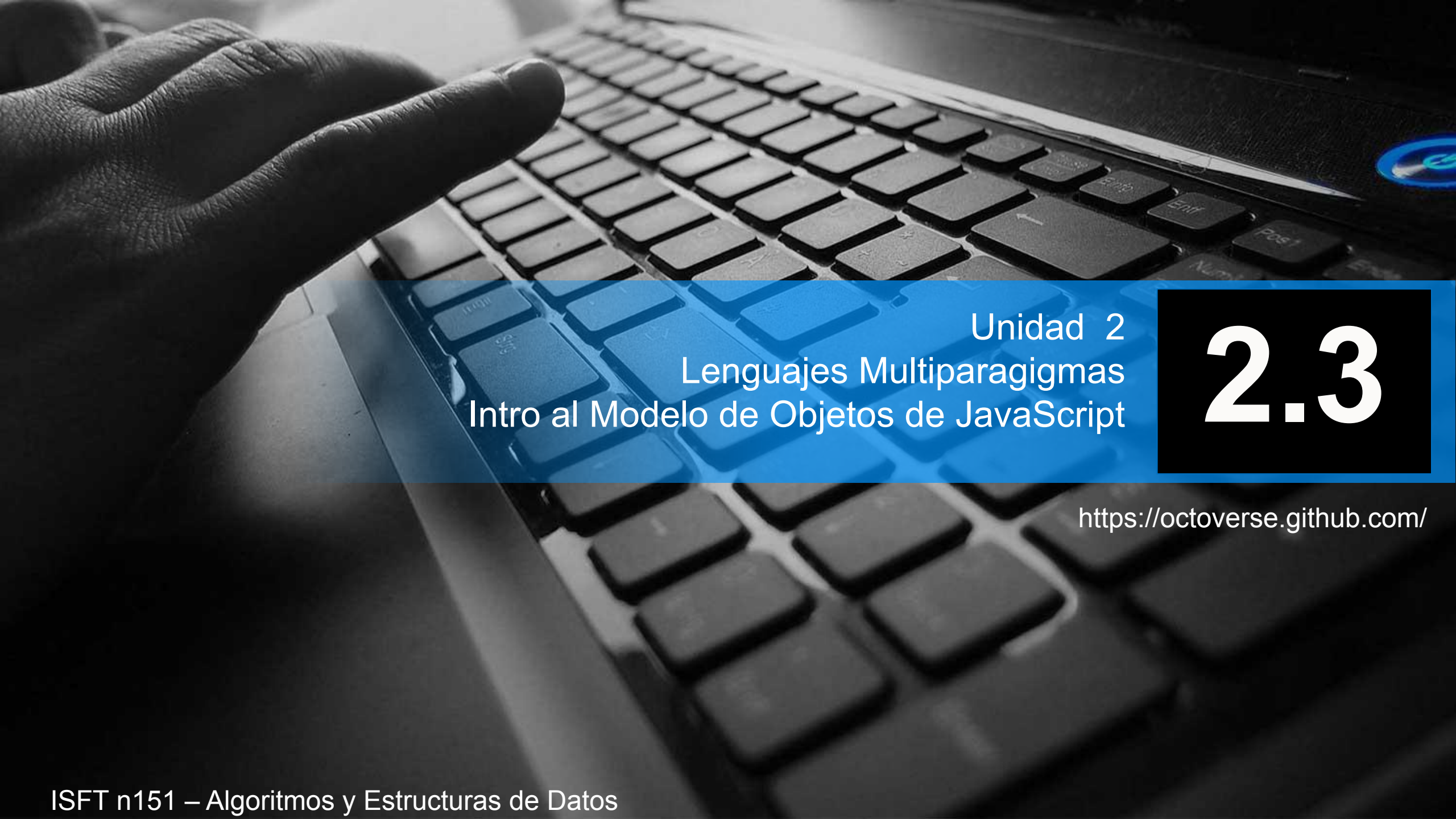
Swift, al igual que Go, fue creado en esta década por el gigante tecnológico Apple. Es un lenguaje de programación multiparadigma, compilado y de propósito general, ofreciendo así una alta productividad a sus desarrolladores.
Swift es el lenguaje de programación principal si deseas desarrollar aplicaciones en iOS y posee una sintaxis muy simple y concisa, ya que al ser un lenguaje compilado posee un rendimiento similar al de C++.
Creemos que este lenguaje está dentro del ranking de los lenguajes de programación más usados gracias a la popularidad que está ganando, ya que no sólo puedes desarrollar aplicaciones para iOS, sino que también en otros sistemas operativos.

Visual Basic

Según el índice TIOBE, Visual Basic es el sexto lenguaje de programación más usado en el 2021. Se trata de un lenguaje orientado a objetivos y dirigido por eventos, desarrollado por Alan Cooper para Microsoft.
Aunque es de propósito general, Visual Basic también provee facilidades para el desarrollo de aplicaciones de bases de datos usando Data Access Objects, Remote Data Objects o ActiveX Data Objects.

Kotlin

Desde que Google declaró que Kotlin era su lenguaje de programación más usado para desarrollar aplicaciones en Android, su popularidad ha crecido considerablemente. No solo se usa para Android, pues también sirve para el desarrollo web y de aplicaciones de escritorio.
Kotlin es un lenguaje de programación de propósito general de tipo estático que admite funciones de programación funcional y orientada a objetos.
Es compatible con Java y todas sus bibliotecas. Y esto no es gratuito, pues su objetivo es reemplazar a Java para hacer lo mismo, pero de forma mucho más sencilla. 😱 De esta manera, quieren facilitar la migración a este lenguaje de programación.



Unidad 2

Lenguajes Multiparagigmas

Intro al Modelo de Objetos de JavaScript

2.3

https://octoverse.github.com/

Lenguajes Multiparadigmas

Un Lenguaje de Programación Multiparadigma es el cual soporta más de un Paradigma de Programación. Según lo describe **Bjarne Stroustrup,** permiten crear “programas usando más de un estilo de programación”.

El objetivo en el diseño de estos lenguajes es permitir a los programadores utilizar el mejor paradigma para cada trabajo, admitiendo que ninguno resuelve todos los problemas de la forma más fácil y eficiente posible.

Por ejemplo, lenguajes de programación como C++, Genie, Delphi, Visual Basic o PHP, combinan el Paradigma imperativo con la orientación a objetos.

Paradigma Orientado a Objetos

Basados en Clases
Basados en Prototipos…

Prototipado vs Clases

En el paradigma orientado a objetos, cuando se suscita el problema de que existan **diferentes objetos con igual comportamiento pero diferente identidad y estado interno**, podemos abordarlo de diferentes formas.

Dos esquemas muy usados son el de **clases** y el que se basa en **prototipos**.

Cada lenguaje (o herramienta) suele implementar una sóla de éstas opciones, aunque existen variaciones.
Los lenguajes más tradicionales (como Java, Smalltalk, y C#) usan el esquema de **clases** y herencia, y en otros lugares (como Javascript, ó en el Object Browser para Smalltalk), se usa el esquema de **prototipado**.

Más allá de cuál de estos esquemas usemos, la base es la misma: objetos que se mandan mensajes en un ambiente aprovechando las ideas de encapsulamiento, delegación y polimorfismo.

Detalles del modelo de objetos

Lenguajes basados en clases vs. basados en prototipos

Los lenguajes orientados a objetos basados en clases, como Java y C++, se basan en el concepto de dos entidades distintas: clases e instancias.

- Una *clase* define todas las propiedades (considerando como propiedades los métodos y campos de Java, o los miembros de C++) que caracterizan un determinado conjunto de objetos. Una clase es una entidad abstracta, más que cualquier miembro en particular del conjunto de objetos que describe. Por ejemplo, la clase `Empleado` puede representar al conjunto de todos los empleados.

- Una *instancia*, por otro lado, es la instanciación de una clase; es decir, uno de sus miembros. Por ejemplo, `Victoria` podría ser una instancia de la clase `Empleado`, representando a un individuo en particular como un empleado. Una instancia tiene exactamente las mismas propiedades de su clase padre (ni más, ni menos).

Un lenguaje basado en prototipos, como JavaScript, no hace esta distinción: simplemente tiene objetos. Un lenguaje basado en prototipos toma el concepto de *objeto prototípico*, un objeto que se utiliza como una plantilla a partir de la cual se obtiene el conjunto inicial de propiedades de un nuevo objeto. Cualquier objeto puede especificar sus propias propiedades, ya sea cuando es creado o en tiempo de ejecución. Adicionalmente, cualquier objeto puede ser utilizado como el *prototipo* de otro objeto, permitiendo al segundo objeto compartir las propiedades del primero.

Definición de una clase y un prototipo

En los lenguajes basados en clases defines una clase en una *definición de clase* "separada". En esa definición puedes especificar métodos especiales, llamados *constructores*, para crear instancias de la clase. Un método constructor puede especificar valores iniciales para las propiedades de la instancia y realizar otro procesamiento de inicialización apropiado en el momento de la creación. Se utiliza el operador new junto al constructor para crear instancias de clases.

JavaScript sigue un modelo similar, pero sin tener la definición de clase separada del constructor. En su lugar, se define una función constructor para crear objetos con un conjunto inicial de propiedades y valores. Cualquier función JavaScript puede utilizarse como constructor. Se utiliza el operador **new** con una función constructor para crear un nuevo objeto. (Nota que ECMAScript 2015 introduce la declaración de clases)

Subclases y herencia

En un lenguaje basado en clases, creas una jerarquía de clases a través de la definición de clases. En una definición de clase, puedes especificar que la nueva clase es una *subclase* de una clase existente. Esta subclase hereda todas las propiedades de la superclase y - además -puede añadir nuevas propiedades o modificar las heredadas. Por ejemplo, supongamos que la clase Employee tiene sólo las propiedades name y dept, y que Manager es una subclase de Employee que añade la propiedad reports. En este caso, una instancia de la clase Manager tendría las tres propiedades: name, dept, y reports.

JavaScript implementa la herencia permitiendo asociar un objeto prototípico con cualquier función *constructor*. De esta forma puedes crear una relación entre Employee y Manager, pero usando una terminología diferente. En primer lugar, se define la función *constructor* para Employee, especificando las propiedades name y dept. Luego, se define la función *constructor* para Manager, especificando la propiedad reports. Por último, se asigna un nuevo objeto derivado de Employee.prototype como el prototype para la función *constructor* de Manager. De esta forma, cuando se crea un nuevo Manager, este hereda las propiedades name y dept del objeto Employee.

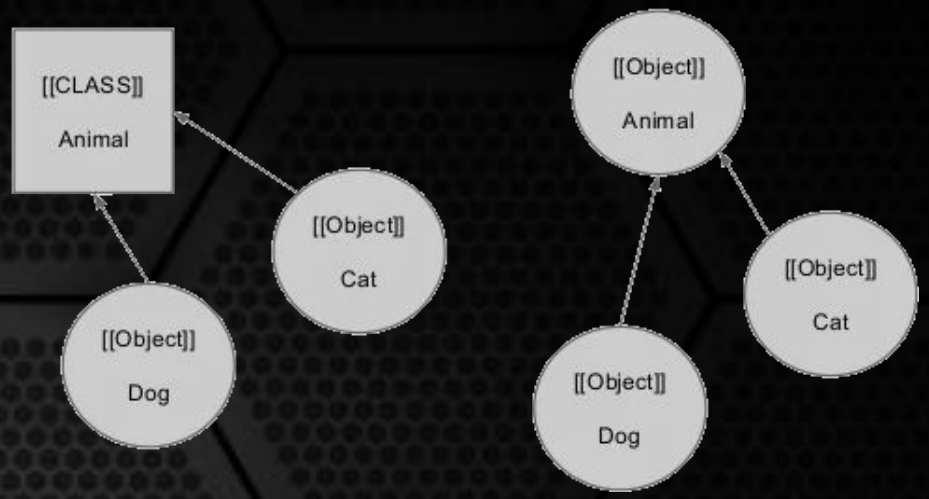
Resumen de las diferencias

Categoría	Basado en clases (Java)	Basado en prototipos (JavaScript)
Clase vs. Instancia	La clase y su instancia son entidades distintas.	Todos los objetos pueden heredar de otro objeto.
Definición	Define una clase con una definición *class*; se instancia una clase con métodos constructores.	Define y crea un conjunto de objetos con funciones constructoras.
Creación de objeto	Se crea un objeto con el operador `new`.	Se crea un objeto con el operador `new`.
Construcción de jerarquía de objetos	Se construye una jerarquía de objetos utilizando definiciones de clases para definir subclases de clases existentes.	Se construye una jerarquía de objetos mediante la asignación de un objeto como el prototipo asociado a una función constructora.
Herencia	Se heredan propiedades siguiendo la cadena de clases.	Se heredan propiedades siguiendo la cadena de prototipos.
Extensión de propiedades	La definición de una clase especifica *todas* las propiedades de todas las instancias de esa clase. No se puede añadir propiedades dinámicamente en tiempo de ejecución.	El conjunto *inicial* de propiedades lo determina la función constructor o el prototipo. Se pueden añadir y quitar propiedades dinámicamente a objetos específicos o a un conjunto de objetos.

Resumen – Conclusión.

¿En qué se diferencia el POO basado en prototipos del basado en clases?

La programación basada en prototipos es un estilo de programación orientada a objetos en el que la herencia se implementa a través del proceso de clonación de objetos existentes que sirven como prototipos. En los lenguajes basados en prototipos, los objetos heredan directamente de otros objetos, en los lenguajes POO basados en clases más clásicos, la herencia se basa en la relación "padre-hijo" entre las clases.



Diferencia con la herencia basada en clases

En la clase, la herencia y la instanciación del lenguaje OOP se sienten como dos cosas diferentes y separadas. Cuando se define una subclase, se crea una nueva clase que hereda los miembros y el comportamiento de su clase base y, a continuación, se extiende. Después, si necesita manipular el estado y los datos de su subclase recién creada, debe crear una instancia. Entonces, y solo entonces, puedes tratarlo como un objeto.
En el caso de la POO prototípica, los actos de "herencia" e "instanciación" se sienten como el mismo proceso. No hay noción de una "clase", por lo que simplemente omita el proceso de definición de una clase. Solo hay objetos. Los instancias en el momento de la creación.

Por ejemplo, en términos de POO basada en clases, y son clases base e hija respectivamente. Por lo general, es una abstracción, que describe la "fruta" general e incluso no se puede instanciar. es una clase concreta, que se puede instanciar, toma todas las propiedades de y las extiende con propiedades especiales a "manzana". En el caso de un prototipo basado en OOP no es una abstracción, es un objeto concreto así como , por lo que puede ponerlo en la cesta (matriz), realizar una operación de "comer" en él o enviarlo a un amigo (para que funcione como un argumento). Suena poco realista, ya que no se puede hacer esto para fructificar en la vida real, porque no hay "Fruta" abstracta, solo concretas. Pero lo que no está en línea con la vida real, podría ser más adecuado para las necesidades de programación

El **software libre** se enfoca en las libertades filosóficas que se les otorga a los usuarios, mientras que el **software de código abierto** se enfoca en las ventajas de su modelo de desarrollo.

Ecosistemas de Desarrollo

Java - .Net – (Python)

BENEFICIOS E IMPORTANCIA DEL SOFTWARE LIBRE Y CÓDIGO ABIERTO

Aunque el Software privativo y cerrado no desaparecerá, el Software Libre y Código Abierto les otorgan a las organizaciones que lo proveen y/o usan, beneficios importantes, para sí mismo y otros, que lo hacen invaluables para las misma. Y entre esos podemos destacar los siguientes:

Facilitar y abaratar los procesos de inserción e invención en el mundo moderno, sobre todo en el ámbito de la nueva economía digital, sin desmerito de la eficiencia, flexibilidad y seguridad necesaria para el logro de los objetivos.

Aumentar y favorecer la sinergia entre las Organizaciones y las Comunidades por medio de la innovación abierta para mejorar la adopción de las transformaciones tecnológicas y digitales, y así lograr de forma más ágil y efectiva, las demandas cambiantes del modelo de negocio de las organizaciones y la evolución de las sociedades.

Crear una Sociedad más humana, creativa y productiva mediante la difusión, el compartir y la colaboración del conocimiento y la creatividad, no necesariamente orientada al lucro.
Ahorrar costos en licencias y procesos de implementación, evitar el uso de Sistemas de Información privativos y cerrados, y favorecer el uso de arquitecturas abiertas, que faciliten la independencia tecnológica progresiva, ante monopolios de fabricantes, proveedores y gobiernos.

Modelo de Negocio **“Open Source”**
Formas en que los desarrolladores de código abierto pueden ganar dinero:

- Soporte pagado
- Software como servicio (OpenSaaS)
- Modelo de núcleo abierto
- Patrocinadores de GitHub
- Solicitudes de funciones de pago
- Reciba pagos por crear extensiones de código abierto para productos existentes

Microsoft Microsoft Open Source

Ecosistema de aplicaciones de Código Abierto de Microsoft

A nivel de Software Microsoft ha compartido y liberado los siguientes ítems:

KERNEL DE LINUX O SUBSISTEMA DE WINDOWS PARA LINUX (WSL)
Actualmente, Microsoft tiene incorporado el Kernel de Linux> dentro de Windows, y además dentro de su tienda web, permite la descarga de Distribuciones (Distros) completas y funcionales. Para detallar más al respecto que es el Subsistema de Windows para Linux (WSL), podemos citar directamente información oficial, que dice lo siguiente:

EL SUBSISTEMA DE WINDOWS PARA LINUX (WSL) ES UNA NUEVA CARACTERÍSTICA DE WINDOWS 10/11 QUE TE PERMITE EJECUTAR HERRAMIENTAS DE LÍNEA DE COMANDOS NATIVAS DE LINUX DIRECTAMENTE EN WINDOWS, JUNTO CON EL ESCRITORIO TRADICIONAL DE WINDOWS Y LAS APLICACIONES MODERNAS DE LA TIENDA. SECCIÓN DE DOCUMENTACIÓN DE MICROSOFT SOBRE WSL (LEER MÁS)

EL SUBSISTEMA DE WINDOWS PARA LINUX PERMITE A LOS DESARROLLADORES EJECUTAR UN ENTORNO DE GNU/LINUX, INCLUIDA LA MAYORÍA DE HERRAMIENTAS DE LÍNEA DE COMANDOS, UTILIDADES Y APLICACIONES, DIRECTAMENTE EN WINDOWS, SIN MODIFICAR Y SIN LA SOBRECARGA DE UNA MÁQUINA VIRTUAL. DOCUMENTACIÓN DEL SUBSISTEMA DE WINDOWS PARA LINUX

Y actualmente va por la versión 2, WSL 2, la cual tiene una arquitectura completamente nueva que usa un Kernel de Linux real. De esta nueva y actual versión vale la pena citar lo siguiente:

WSL 2 ES UNA NUEVA VERSIÓN DE LA ARQUITECTURA QUE PERMITE QUE EL SUBSISTEMA DE WINDOWS PARA LINUX EJECUTE ARCHIVOS BINARIOS DE ELF64 LINUX EN WINDOWS. SUS PRINCIPALES OBJETIVOS SON AUMENTAR EL RENDIMIENTO DEL SISTEMA DE ARCHIVOS, ASÍ COMO AGREGAR COMPATIBILIDAD COMPLETA CON LLAMADAS DEL SISTEMA. ESTA NUEVA ARQUITECTURA CAMBIA EL MODO EN QUE ESTOS ARCHIVOS BINARIOS DE LINUX INTERACTÚAN CON WINDOWS Y EL HARDWARE DEL EQUIPO, PERO PROPORCIONA LA MISMA EXPERIENCIA DE USUARIO QUE EN WSL 1. ACERCA DE WSL2

Herramientas de Desarrollo
.Net
ML.Net
Cognitive Toolkit de Microsoft
Visual Studio Code
Microsoft Application Inspector
Teams – Skype –MS Edge

Ecosistemas de Desarrollo Java - .Net – (Python)

Varios autores han expresado los elementos fundamentales que conforman los ecosistemas de software, como idea general varios coinciden en que los ecosistemas de software están orientados a crear alianzas entre diferentes organizaciones, enfoque a la externalización de las relaciones de las organizaciones, potenciando la productividad y compartir sectores de mercado (Bosch, 2012) (Piñero, y otros, 2015) (Lugo, 2015).

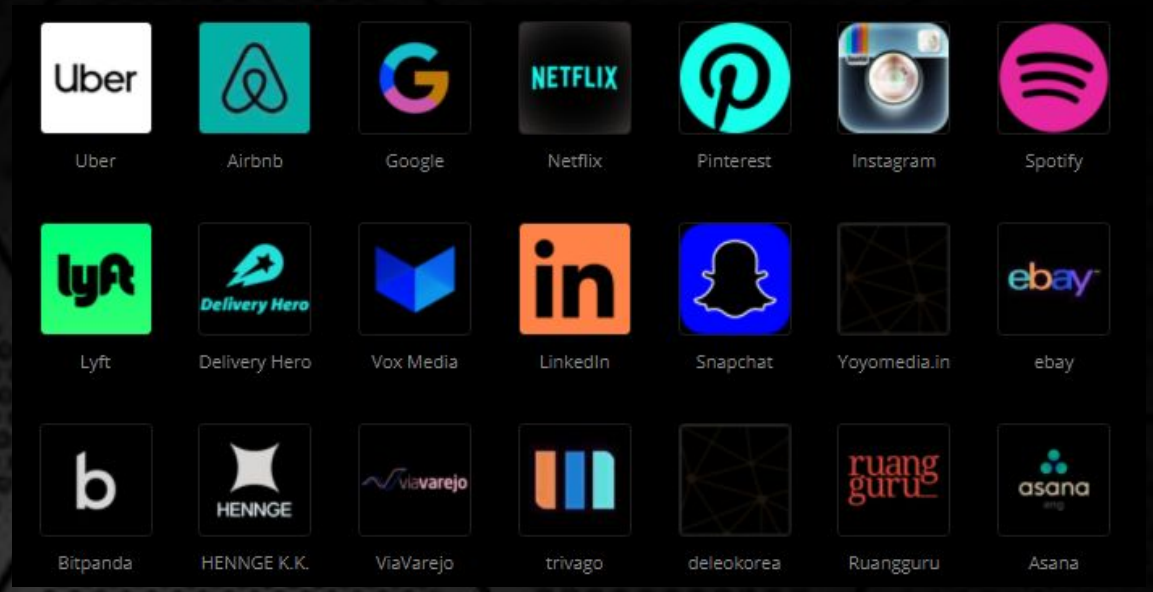
Bosh (Bosch, 2012), por ejemplo, define un ecosistema de software como el conjunto de soluciones de software que habilitan, automatizan y soportan cierta necesidad de negocio, mientras que, Recena (Recena, y otros, 2012) considera que la base para el desarrollo de los ecosistemas de software está en el establecimiento de sólidas arquitecturas empresariales que se apoyen en la integración de sus componentes.

La mayoría de las empresas de desarrollo web o móvil se enfrentan a varios dilemas cuando comienzan un nuevo proyecto. Uno de los principales dilemas a los que se enfrentan es seleccionar el lenguaje (Plataforma) de programación adecuado.
Si revisas, más de 600 lenguajes de programación están disponibles en el mercado. Don´t Panic. No necesita conocerlos todos, ya que no todos los idiomas se adaptan a su proyecto.
Sin embargo, ciertos lenguajes de programación son altamente preferidos por los desarrolladores de todo el mundo. Algunos ejemplos son .C# Net, Java y Python. Ahora, los desarrolladores de punto net siempre preferirán el lenguaje con el que están familiarizados. Lo mismo ocurre con los desarrolladores de Java y Python.
Entonces, siendo Analista, un gerente de proyecto o propietario de una empresa de desarrollo web, ¿cómo elegirá el lenguaje adecuado para un proyecto?

Java (ORACLE)

Java es un lenguaje de programación ampliamente utilizado que actúa como un lenguaje del lado del servidor para back-end, big data y desarrollo de Android. Es un lenguaje de programación orientado a objetos que se implementó originalmente como un lenguaje de programación para televisión interactiva.

El establecimiento de la especificación del lenguaje Java, escrita en julio de 1995, permitió el desarrollo de un compilador Java totalmente ejecutable en noviembre de 1997.



Twitter, Cash App, Spotify, Signal, Uber, Netflix y más son aplicaciones muy populares desarrolladas con Java.
Estrellas de Java GitHub: 34.1K | Bifurcaciones Java GitHub: 11.7K

Características principales de Java

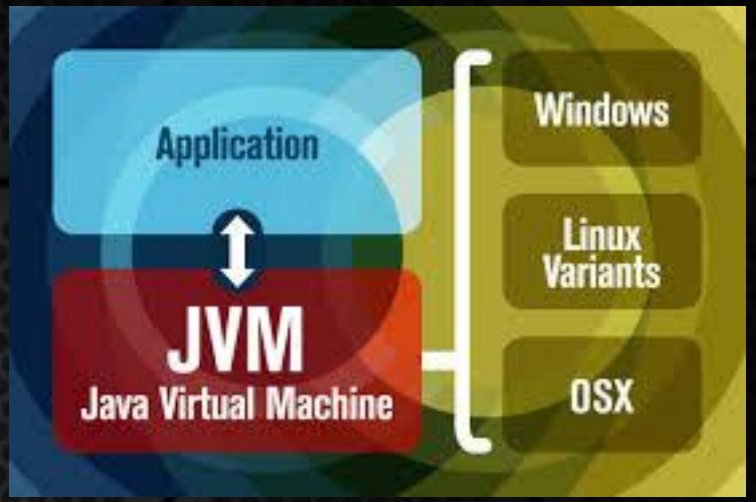
Orientado a objetos: El programa Java es una colección de objetos que se comunican invocando los métodos de los demás. Las técnicas de programación orientada a objetos (POO) le ayudan a lidiar con la complejidad identificando distintas áreas de responsabilidad y ocultando detalles de implementación detrás de interfaces bien definidas.

Arquitectura neutral: El compilador Java genera un formato de archivo de objeto independiente de la arquitectura, que hace que el código compilado sea ejecutable en muchos procesadores, con la presencia de un sistema de ejecución Java.

Multiproceso: Con la función multiproceso de Java, los desarrolladores pueden escribir programas que pueden realizar múltiples tareas simultáneamente. Permite a los desarrolladores construir aplicaciones interactivas que pueden funcionar sin problemas.

Java (ORACLE)

Ventajas de Java
Grandes bibliotecas: El ecosistema Java es enorme. Tiene numerosas bibliotecas excelentes que se pueden usar para diversos propósitos, como Hibernate para ORM, Jackson para procesamiento JSON, Guava para colecciones, Joda-Time para objetos de fecha y hora, y Lombok para reducir el código repetitivo, etc.



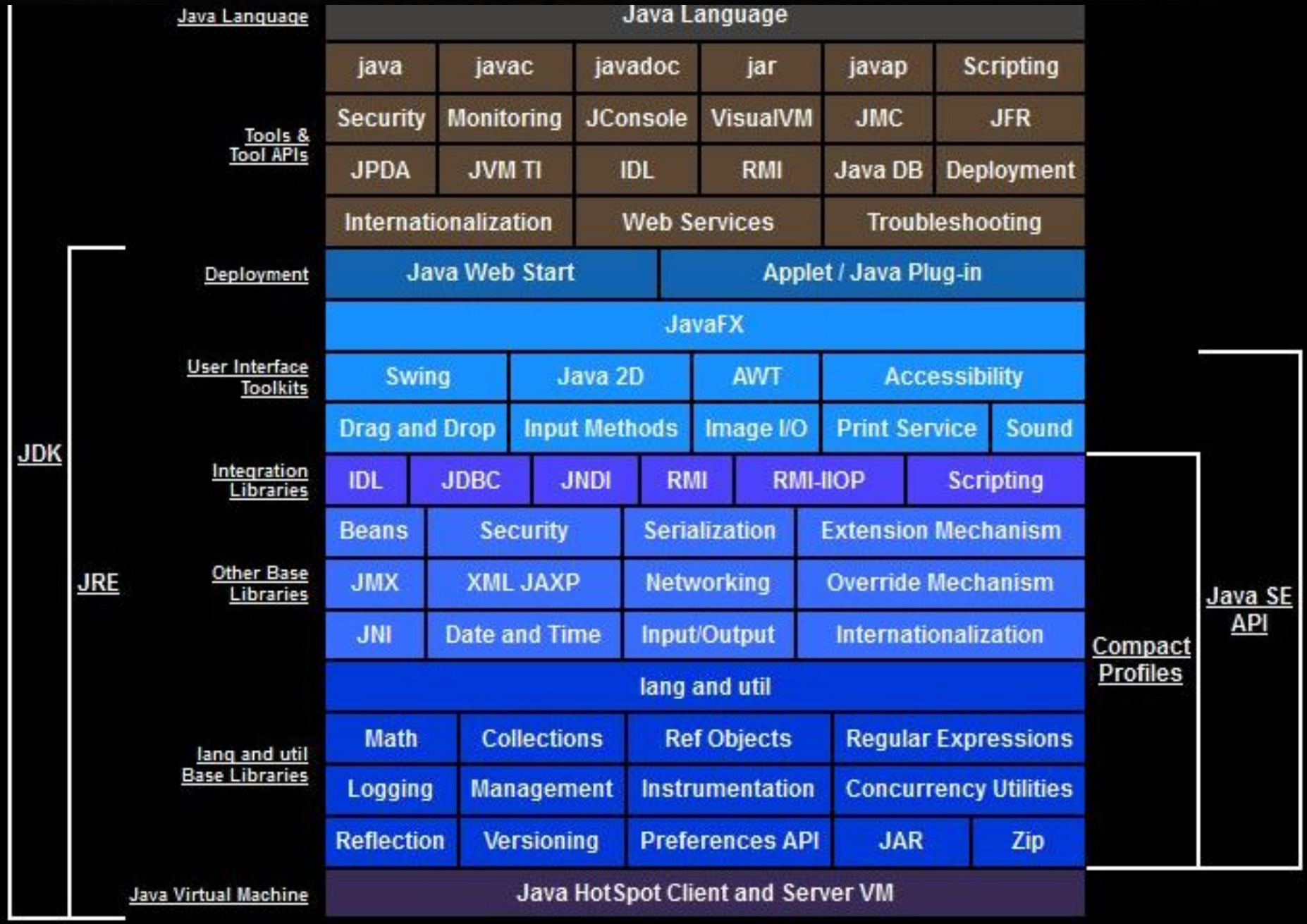
VM: La máquina virtual Java es una máquina abstracta que proporciona una plataforma para ejecutar programas Java. Es el corazón de la **plataforma Java** y permite que Java sea portátil a través de diferentes tipos de sistemas, ya sea que usen Windows, macOS o Linux.

Recolección automática de basura: Una de las grandes ventajas de Java es la recolección automática de basura. Recupera automáticamente la memoria de los objetos que ya no están en uso por el programa. Ayuda a evitar fugas de memoria, lo que puede ser un gran problema en idiomas que no tienen esta característica.

Contras de Java
Curva de aprendizaje empinada: Java tiene una curva de aprendizaje empinada, especialmente para principiantes. La sintaxis no es fácil de entender, y lleva algún tiempo acostumbrarse a la forma en que funcionan las cosas en Java.

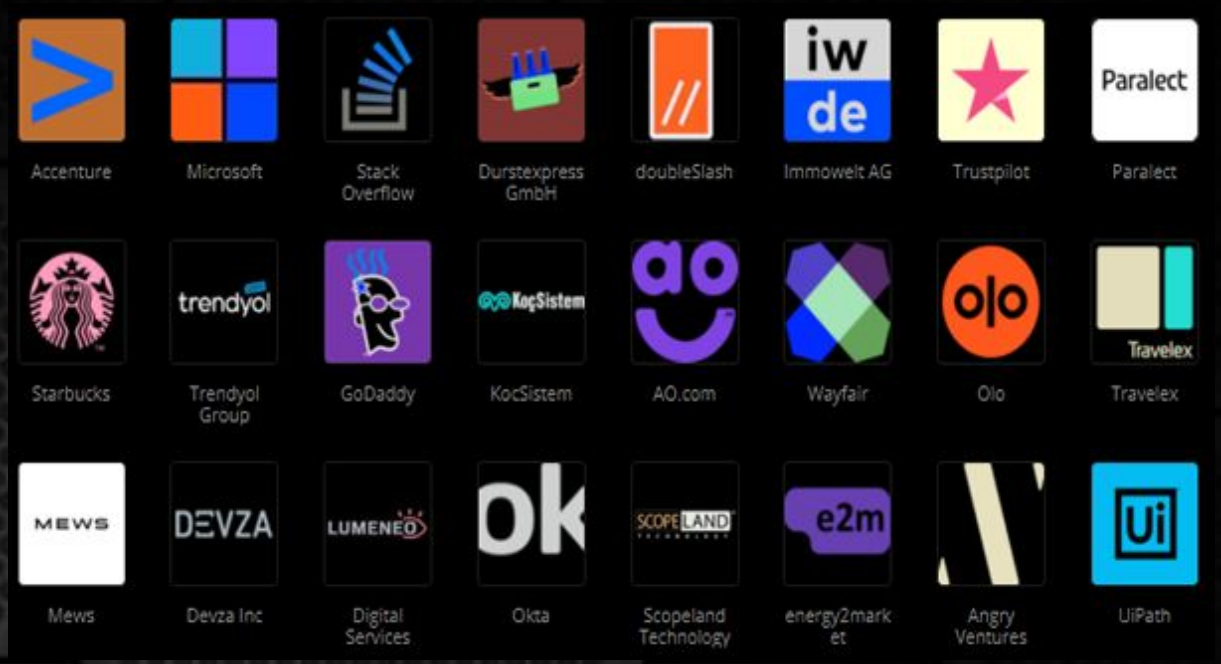
Java para el desarrollo web es la mejor opción para esto; puede preferir la opción de alquiler de codificadores de Java. Esto te ayudará a construir un excelente sitio web.

Java Platform



.Net (Microsoft)

.Net es una plataforma de código abierto a través de la cual se pueden desarrollar varios tipos de aplicaciones. Con esta plataforma, puede usar varios lenguajes, editores y bibliotecas para crear para la web, el escritorio, los juegos y el IoT. .Net admite varios lenguajes de programación como C#, C++, VB.Net y F#.



Accenture, Starbucks, Stack Overflow, Microsoft y muchas otras son marcas conocidas que usan .Net.
.Net GitHub Estrellas: 15.2K | . Bifurcaciones netas de GitHub: 3.8K?

Características principales de .Net
Motor de Common Language Runtime (CLR): Esto ayuda en la ejecución rápida de programas .Net. CLR también proporciona seguridad mediante la comprobación de la seguridad del tipo y la seguridad de la memoria.

Independencia lingüística: Es posible escribir programas .Net en varios lenguajes. Esta característica se logra mediante CLR, que convierte el código fuente en un lenguaje intermedio común.

Biblioteca de clases base: .Net proporciona una biblioteca de clases base, una colección de clases reutilizables que se pueden usar para desarrollar diversas aplicaciones.

.Net (Microsoft)

Ventajas de .Net
Código estable: El código escrito en .Net es más estable que otros lenguajes. Esto se debe a que CLR proporciona seguridad de tipo y seguridad de memoria, lo que ayuda a evitar bloqueos inesperados.

Great 3rd Party Libraries: .Net tiene un gran ecosistema de varias bibliotecas de terceros que se pueden utilizar para diversos fines, como desarrollo web, aplicaciones de escritorio, etc.

Gran comunidad: .Net tiene una gran comunidad de desarrolladores que siempre están dispuestos a ayudar y apoyar.

Contras de .Net
Rendimiento lento: El rendimiento de las aplicaciones .Net no es tan bueno en comparación con otros lenguajes como C ++. Esto se debe a que CLR necesita convertir el código fuente en un lenguaje intermedio común, lo que lleva algún tiempo.

Python

Python es un lenguaje de programación de propósito general utilizado para el back-end, el desarrollo de software y web, la ciencia de datos y la escritura de scripts de sistemas sobre otras cosas. La tecnología es fácil de aprender, la sintaxis mantiene la legibilidad y, por lo tanto, disminuye el gasto de mantenimiento del programa.

YouTube, Google, Quora, Pinterest, Instagram y más son aplicaciones muy famosas creadas con Python.
Estrellas de GitHub de Python: 35.8K | Bifurcaciones de GitHub de Python: 17.6K

Características principales de Python
Enfoque orientado a objetos: Python representa el enfoque orientado a objetos de script de sitio, lo que ayuda a desarrollar aplicaciones sólidas.

Lenguaje tipado dinámicamente: Python es un lenguaje de tipo dinámico, lo que significa que el tipo de una variable no se establece de antemano. Es fácil escribir programas ya que no es necesario declarar el tipo de variables de antemano.

Supports GUI: Python supports GUI programming, el cual ayuda a crea interfaces amigables hasta desde línea de comandos.

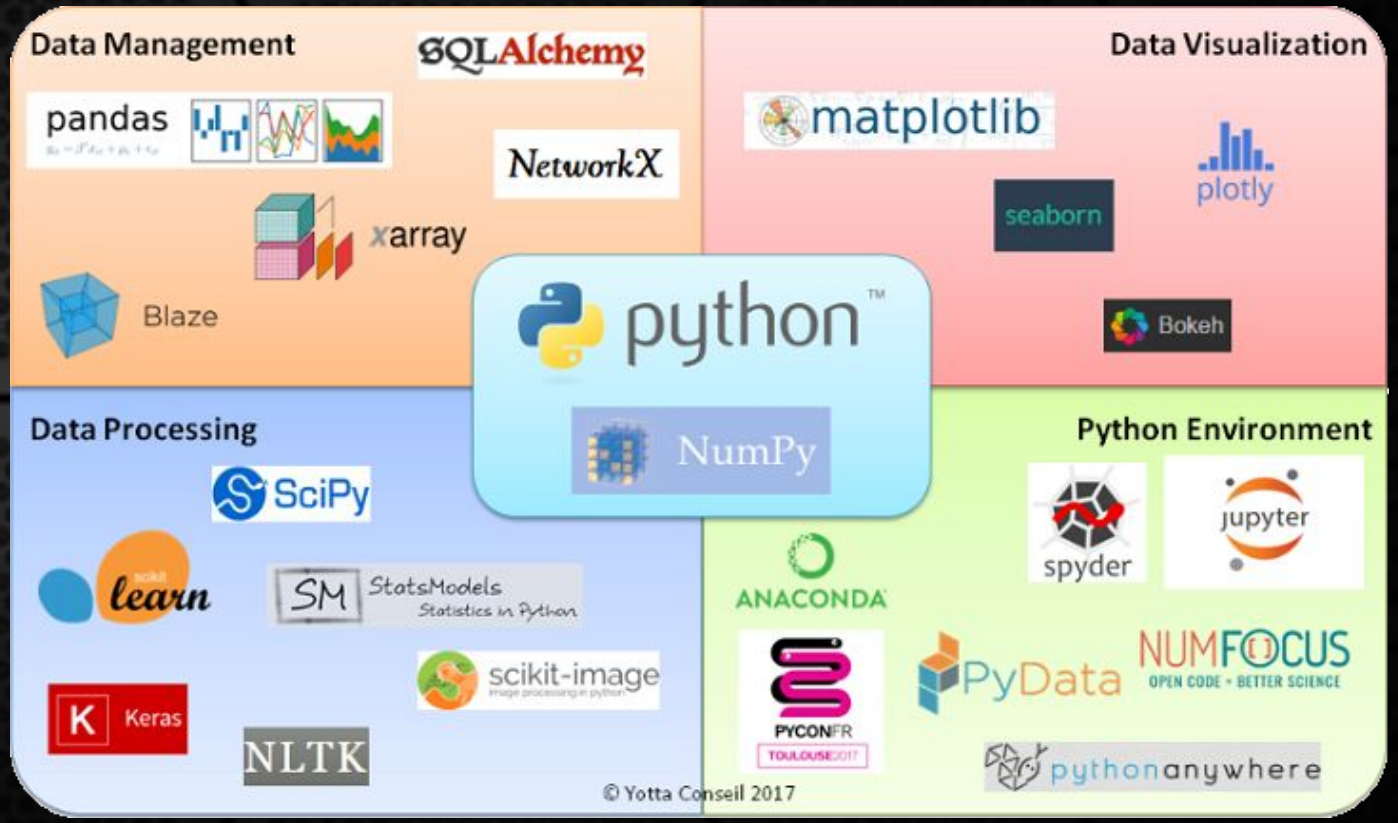
Python - ecosystem

Ventajas de Python
Fácil de aprender: Python es un lenguaje fácil de aprender, ya que tiene una sintaxis y legibilidad simples. Esto lo hace ideal para principiantes que desean comenzar con la programación.

Altamente escalable: Python es altamente escalable, lo que significa que se puede usar para aplicaciones a gran escala. Esto se debe a su enfoque orientado a objetos y escritura dinámica.

Escritura dinámica: La escritura dinámica de Python facilita la escritura de programas, ya que no es necesario declarar el tipo de variables por adelantado.

Contras de Python
Errores de tiempo de ejecución: Python es un lenguaje de tipo dinámico, lo que significa que los errores solo se pueden detectar en tiempo de ejecución. Es difícil depurar programas, ya que hay que esperar a que se ejecute el programa antes de encontrar errores.



¿Cómo elegir un lenguaje de programación o un ecosistema para un proyecto de Software?

La primera pregunta que te estarás haciendo es…. ¿para qué?
Si necesitas construir una casa desde cero, ¿cuál elegirías? ¿martillo o taladro?

Algunos consejos rápidos

- la lucha de los gerentes corporativos,
- popular, moderno y fresco. Si elige Lisp para un proyecto solo porque es un lenguaje funcional muy limpio y hermoso, luego puede convertirse en una decisión equivocada. Así que es bueno evitar este error.
- El lenguaje de programación para un proyecto debe basarse en las necesidades de su negocio, no solo porque tiene algo de azúcar sintáctico o porque está exagerado. Tal vez sienta que, dado que es el desarrollador, debe depender totalmente de usted, su elección para implementar cualquier idioma. Solo usted tiene toda la libertad de elegir la tecnología que desee, pero esto no funciona en una organización de TI y puede ser contraproducente.
- Como gerente técnico, primero debe prestar atención a todas las piezas móviles de su proyecto. Debes conocer todos los componentes para una mejor visualización y esto te ayudará a elegir un lenguaje de programación específico. Una buena vista al comienzo de su proyecto ayuda a elegir un lenguaje de programación sensato y esto conduce a menos tiempo invertido en
- En programación, si puede escribir un buen software en Java, C#, Python, PHP o cualquier otro lenguaje, también puede escribir un mal software usando estos lenguajes. Ningún idioma es la mejor opción para cualquier software. Algunos lenguajes y marcos se adaptan mejor a los proyectos que otros. Considere el ejemplo de Java. No era un buen lenguaje en el momento en que fue creado. Simplemente era más conveniente que los competidores . Al elegir un idioma, debe pensar de esta manera.

¿Cómo elegir un lenguaje de programación para un proyecto?

Preguntas para hacer al elegir un lenguaje de programación

¿Tiene el lenguaje el soporte adecuado del ecosistema?
¿Va a funcionar a largo plazo?
¿El soporte del proveedor está disponible para el idioma?
¿Cuál es el entorno del proyecto (web, móvil, etc.)?
¿Necesitamos considerar alguna infraestructura como la que necesitamos
¿Cuál es la preferencia del cliente?
Cualquier requisito específico de las bibliotecas son restricciones
¿Cuál es la consideración de rendimiento y los idiomas son adecuados para adaptarse a este rendimiento?
¿Cuál es la consideración de seguridad?
¿Necesitamos usar alguna herramienta de terceros?

Muchas de las preguntas anteriores aclaran la confusión de elegir un lenguaje de programación para su proyecto.
Tener en cuenta que la elección, es muy importante, avanzado le proyecto el costo de cambio es enorme.
Además, deberimaos considerar el aspecto de la Arquitectura y el diseño… esto lo verán en **Ingeniería de Software.**

Como lo Implementamos?

Muchas Gracias

Nos veremos en la Proxima.

Material Complementario

Off-Topic

El Presente Material Corresponde a la Edición 2022
De la Catedra AyED II, dados los Cambios
Curriculares Introducidos, el presente
Material forma Parte de
Practicas Profesionalizantes III.
Permanece aquí para consulta.

JS

JavaScript Objects

3. Function

```
function Book () //constructor
{
  this.property = "value";
  this.method = function () { }; //internal function
  this.method = method; //external function
}

Book.prototype.method() {} //prototype function

let Book = new Book(); //object blueprint
```

javascript keyword

property name

method name

Añadir y quitar propiedades

En lenguajes basados en clases típicamente se crea una clase en tiempo de compilación y luego se crean instancias de la clase, ya sea en tiempo de compilación o en tiempo de ejecución. No se puede cambiar el número o el tipo de propiedades de una clase una vez que ha sido definida.

En **JavaScript**, sin embargo, ***en tiempo de ejecución*** se pueden añadir y quitar propiedades a un objeto. Si se añade una propiedad a un objeto que está siendo utilizado como el prototipo de otros objetos, los objetos para los que es un prototipo también tienen la nueva propiedad añadida.

Employee

```
function Employee(){
  this.name = "";
  this.dept = "general";
}
```

Manager

```
function Manager(){
  this.reports = [];
}
Manager.prototype = new Employee;
```

WorkerBee

```
function WorkerBee(){
  this.projects = [];
}
WorkerBee.prototype = new Employee;
```

SalesPerson

```
function SalesPerson(){
  this.dept = "sales";
  this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;
```

Engineer

```
function Engineer(){
  this.dept = "engineering";
  this.machine = "";
}
Engineer.prototype = new WorkerBee;
```

El ejemplo employee

Este ejemplo utiliza los siguientes objetos:

- `Employee` tiene las propiedades `name` (cuyo valor por defecto es un string vacío) y `dept` (cuyo valor por defecto es "general").
- `Manager` está basado en `Employee`. Añade la propiedad `reports` (cuyo valor por defecto es un array vacío, en la que se pretende almacenar un array de objetos `Employee` como su valor).
- `WorkerBee` también está basado en `Employee`. Añade la propiedad `projects` (cuyo valor por defecto es un array vacío en el que se pretende almacenar un array de strings como su valor).
- `SalesPerson` está basado en `WorkerBee`. Añade la propiedad `quota` (cuyo valor por defecto es 100). También reemplaza la propiedad `dept` con el valor "sales", indicando que todas las salespersons están en el mismo departamento.
- `Engineer` se basa en `WorkerBee`. Añade la propiedad `machine` (cuyo valor por defecto es un string vacío) y también reemplaza la propiedad `dept` con el valor "engineering".


```
Employee
  function Employee(){
    this.name = "";
    this.dept = "general";
  }
```

```
Manager
  function Manager(){
    this.reports = [];
  }
  Manager.prototype = new Employee;
```

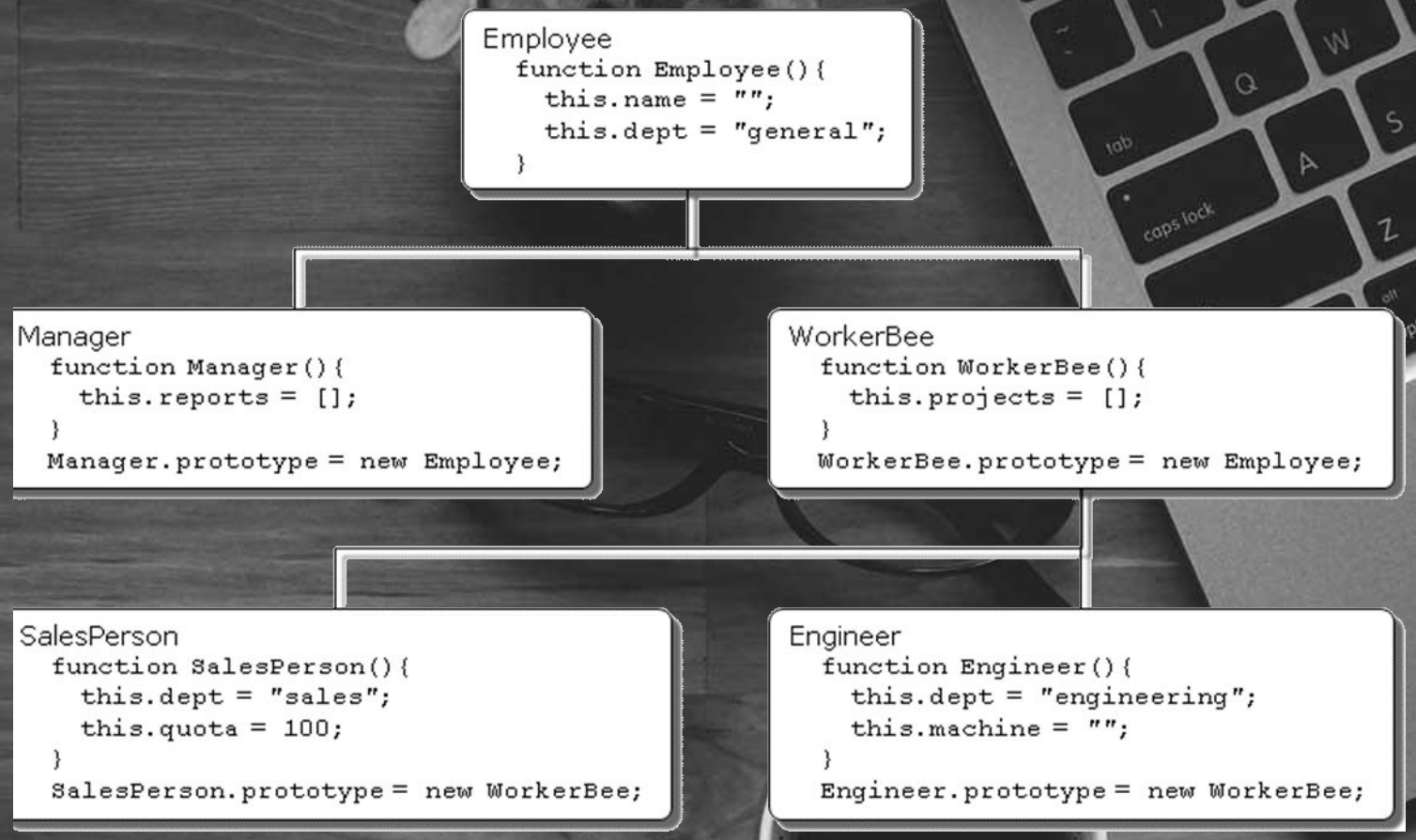
```
WorkerBee
  function WorkerBee(){
    this.projects = [];
  }
  WorkerBee.prototype = new Employee;
```

```
SalesPerson
  function SalesPerson(){
    this.dept = "sales";
    this.quota = 100;
  }
  SalesPerson.prototype = new WorkerBee;
```

```
Engineer
  function Engineer(){
    this.dept = "engineering";
    this.machine = "";
  }
  Engineer.prototype = new WorkerBee;
```

Creación de la jerarquía

Hay varias formas de definir funciones constructor para implementar la jerarquía Employee. Elegir una u otra forma depende sobre todo de lo que quieras y puedas ser capaz de hacer con tu aplicación.
Esta sección veremos como utilizar definiciones muy sencillas (y comparativamente inflexibles) para mostrar como hacer funcionar la herencia. En estas definiciones no puedes especificar valores de propiedades cuando creas un objeto. El nuevo objeto que se crea simplemente obtiene valores por defecto, que puedes cambiar posteriormente.
La sig. figura muestra la jerarquía con estas definiciones sencillas.



Nota: En una aplicación real probablemente definirías constructores que proporcionen valores a las propiedades en el momento de la creación del objeto (debiendo usar Constructores más flexibles). Estas definiciones sencillas nos sirven para mostrar como funciona la herencia.

Creación de la jerarquía

JavaScript

```
function Employee () {
  this.name = "";
  this.dept = "general";
}
```

Java

```
public class Employee {
   public String name;
   public String dept;
   public Employee () {
      this.name = "";
      this.dept = "general";
   }
```

Las siguientes definiciones de `Employee` en Java y en Javascript son similares, la única diferencia es que en Java necesitas especificar el tipo para cada propiedad, no así en Javascript (esto es debido a que Java es un lenguaje fuertemente tipado, mientras que Javascript es un lenguaje débilmente tipado).

Las definiciones de `Manager` y `WorkerBee` ilustran la diferencia a la hora de especificar el siguiente objeto en la jerarquía de herencia. En JavaScript se añade una instancia prototípica como el valor de la propiedad `prototype` de la función constructora, así sobre escribe `prototype.constructor` con la función constructora. Puede hacerse en cualquier momento una vez definido el constructor. En Java se especifica la superclase en la definición de la clase. No se puede cambiar la superclase fuera de la definición de la clase.

JavaScript

```
function Manager () {
  this.reports = [];
}
Manager.prototype = new Employee;

function WorkerBee () {
  this.projects = [];
}
WorkerBee.prototype = new Employee;
```

Java

```
public class Manager extends Employee {
   public Employee[] reports;
   public Manager () {
      this.reports = new Employee[0];
   }
}

public class WorkerBee extends Employee {
   public String[] projects;
   public WorkerBee () {
      this.projects = new String[0];
   }
```

Creación de la jerarquía

JavaScript

```
function SalesPerson () {
   this.dept = "sales";
   this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;

function Engineer () {
   this.dept = "engineering";
   this.machine = "";
}
Engineer.prototype = new WorkerBee;
```

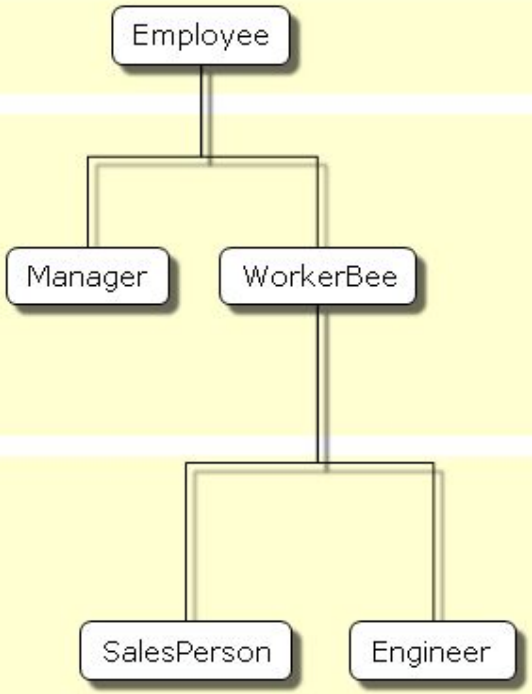
Java

```
public class SalesPerson extends WorkerBee {
   public double quota;
   public SalesPerson () {
      this.dept = "sales";
      this.quota = 100.0;
   }
}

public class Engineer extends WorkerBee {
   public String machine;
   public Engineer () {
      this.dept = "engineering";
      this.machine = "";
   }
}
```

Las definiciones de `Engineer` y `SalesPerson` crean objetos que descienden de `WorkerBee` y por tanto de `Employee`. Un objeto de éste tipo tiene todas las propiedades de los objetos por encima de él en la cadena. Además, estas definiciones reemplazan los valores heredados de la propiedad `dept` con nuevos valores específicos para estos objetos.

Object hierarchy



Individual objects

```
var jim = new Employee;
// jim.name is ''
// jim.dept is 'general'

var sally = new Manager;
// sally.name is ''
// sally.dept is 'general'
// sally.reports is []

var mark = new WorkerBee;
// mark.name is ''
// mark.dept is 'general'
// mark.projects is []

var fred = new SalesPerson;
// fred.name is ''
// fred.dept is 'sales'
// fred.projects is []
// fred.quota is 100

var jane = new Engineer;
// jane.name is ''
// jane.dept is 'engineering'
// jane.projects is []
// jane.machine is ''
```

Nota: El termino *instancia* tiene un significado técnico específico en lenguajes basados en clases, donde una instancia es un ejemplar individual de una clase y es fundamentalmente diferente a la clase. En JavaScript, "instancia" no tiene este mismo significado ya que JavaScript no hace diferencia entre clases e instancias. Sin embargo, al hablar de JavaScript, "instancia" puede ser usado informalmente para indicar que un objeto ha sido creado usando una función constructora particular. En este ejemplo, puedes decir que `jane` es una instancia de `Engineer`. De la misma manera, aunque los términos *parent, child, ancestor*, y *descendant* no tienen un significado formal en JavaScript; puedes usarlos informalmente para referirte a objetos que están por encima o por debajo de la cadena de prototipos.

Propiedades de objetos

Esta sección describiremos cómo heredan los objetos sus propiedades de otros objetos en la cadena de prototipos y qué ocurre cuando se añade una propiedad en tiempo de ejecución.

Herencia de propiedades

Supongamos que creas el objeto `mark` como un `WorkerBee` con la siguiente sentencia:

```
var mark = new WorkerBee;
```

Cuando el intérprete de JavaScript encuentra el operador `new`, crea un nuevo objeto genérico y establece implícitamente el valor de la propiedad interna [[Prototype]] con el valor de `WorkerBee.prototype` y pasa este nuevo objeto como `this` a la función constructora de WorkerBee. La propiedad interna [[Prototype]] (que puede observarse como `__proto__`, la propiedad cuyo nombe tiene dos guiones al principio y al final) determina la cadena de prototipo usada para devolver los valores de la propiedades cuando se accede a ellas. Una vez que estas propiedades tienen sus valores, JavaScript devuelve el nuevo objeto y la sentencia de asignación asigna el nuevo objeto ya inicializado a la variable `mark`.

Este proceso no asigna explícitamente valores al objeto `mark` (valores *locales*) para las propiedades que `mark` hereda de la cadena de prototipos. Cuando solicitas valor de una propiedad, JavaScript primero comprueba si existe un valor para esa propiedad en el objeto. Si existe, se devuelve ese valor; sino, JavaScript comprueba la cadena de prototipos (usando la propiedad `__proto__`). Si un objeto en la cadena de prototipos tiene un valor para esa propiedad, se devuelve ese valor. Si no existe en ningún objeto de la cadena de prototipos un valor para esa propiedad, JavaScript dice que el objeto no tiene esa propiedad.

Propiedades de objetos en javaScript

En el caso de nuestro objeto `mark`, éste tiene las siguientes propiedades y valores:

```
mark.name = "";
mark.dept = "general";
mark.projects = [];
```

El objeto `mark` hereda valores para las propiedades `name` y `dept` su objeto prototipico que enlaza en `mark.__proto__`. Se le asigna un valor local la propiedad `projects` a través del constructor `WorkerBee`.
De esta forma se heredan propiedades y sus valores en JavaScript
Debido a que estos constructores no permiten especificar valores específicos de instancia, esta información es genérica. Los valores de las propiedades son los valores por omisión, compartidos por todos los objetos nuevos creados a partir de `WorkerBee`.

Por supuesto se pueden cambiar después los valores de estas propiedades.
Por ejemplo podríamos dar valores con información específica a `mark` de la siguiente forma:

```
mark.name = "Doe, Mark";
mark.dept = "admin";
mark.projects = ["navigator"];
```

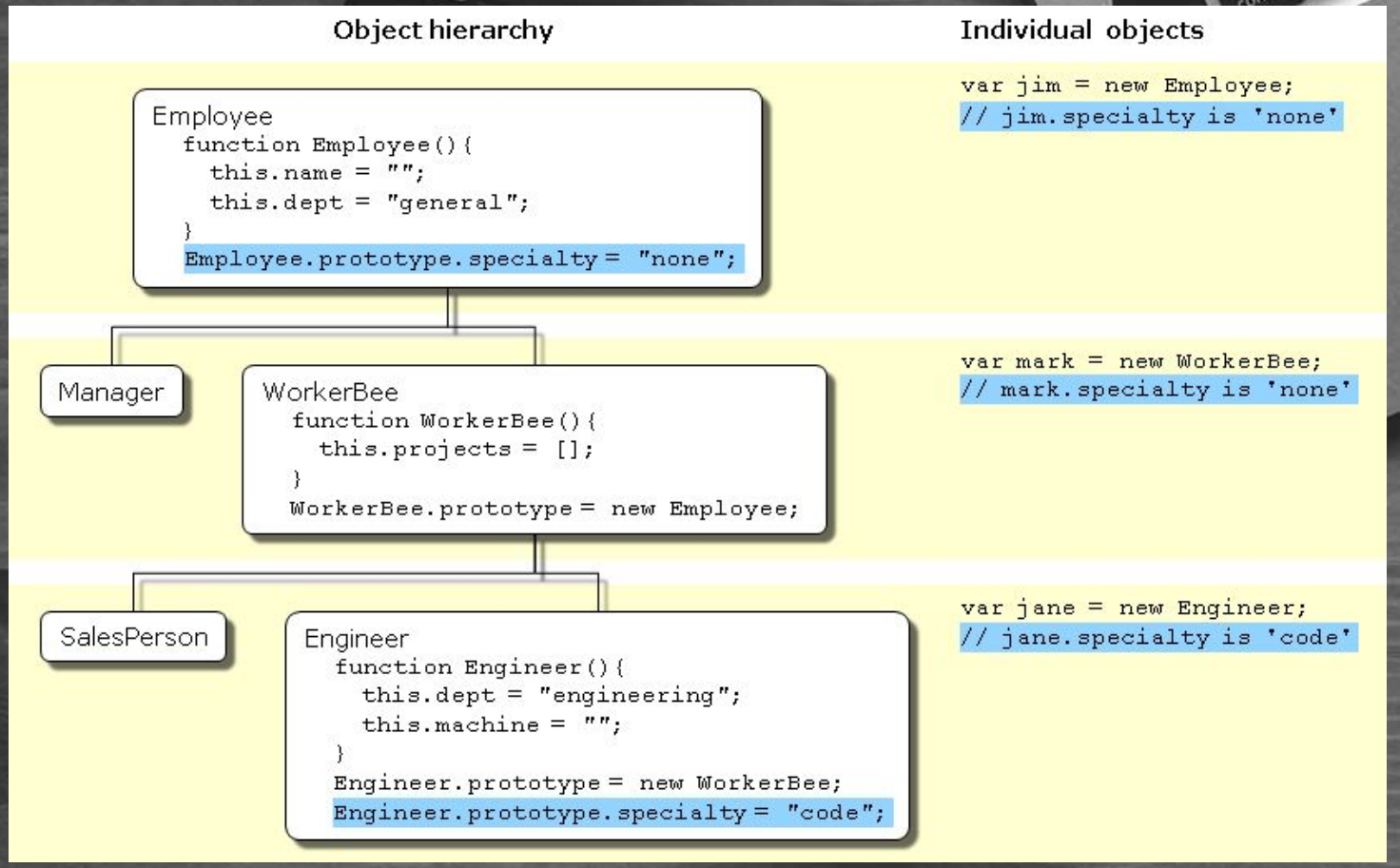
Añadir Propiedades a los objetos de JavaScript

En JavaScript puedes añadir propiedades a los objetos en tiempo de ejecución. No estás limitado a utilizar solo las propiedades que proporciona la función constructora. Para añadir una propiedad que es especifica para un objeto determinado, se le asigna un valor a la propiedad del objeto de la siguiente forma:

```
mark.bonus = 3000;
```

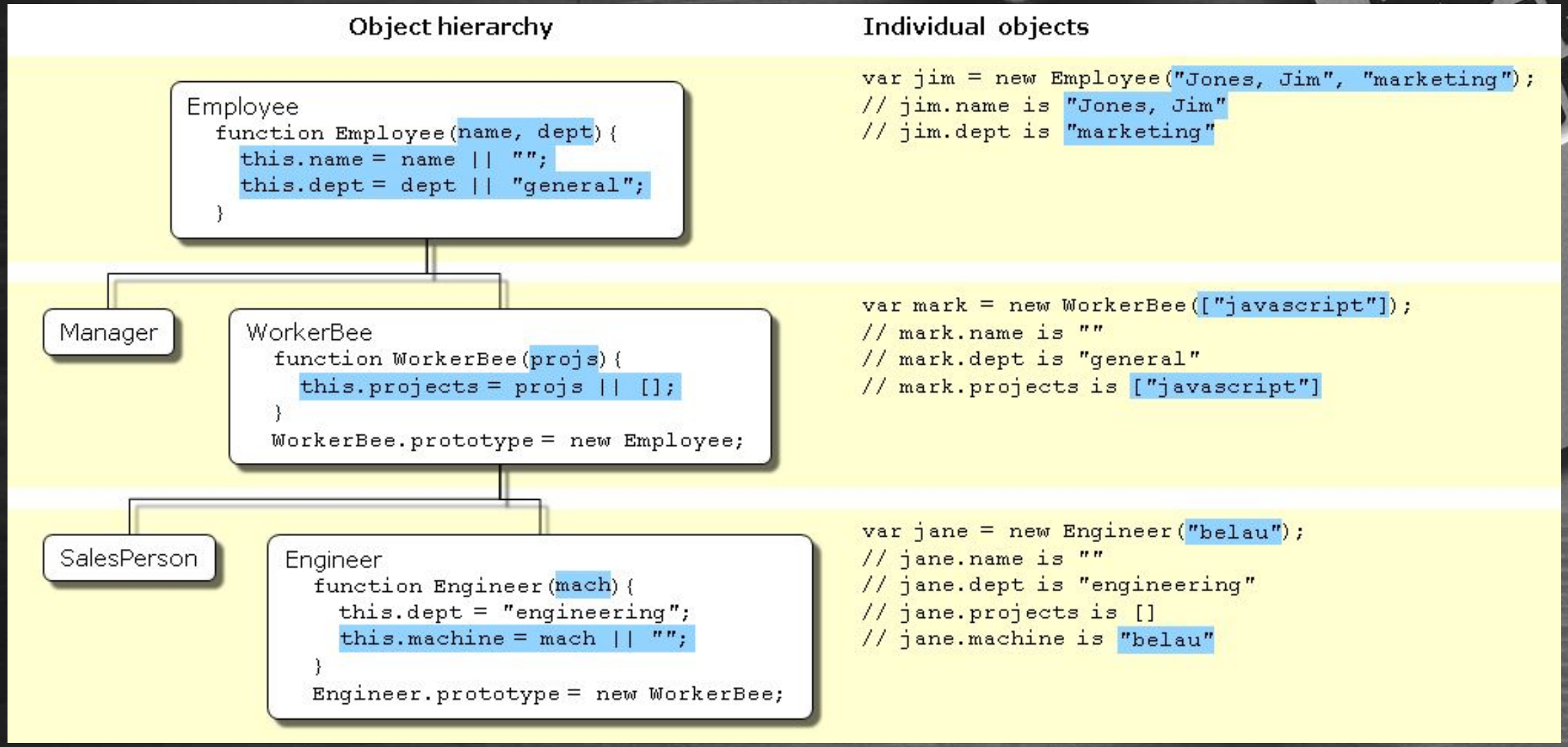
Ahora el objeto mark tiene una propiedad bonus, pero ningún otro objeto creado con la función *constructor* WorkerBee tiene esta propiedad. Si añades una nueva propiedad a un objeto que se esta utilizando como el prototipo de una función *constructor*, dicha propiedad se añade a todos los objetos que heredan propiedades de dicho prototipo. Por ejemplo, puedes añadir una propiedad specialty a todos los empleados con la siguientes sentencia:
Employee.prototype.specialty = "none";

Tan pronto JavaScript ejecuta esta sentencia, el objeto mark también tienen la propiedad specialty con el valor "none". La siguiente figura muestra el efecto de añadir esta propiedad al prototipo Employee y después reemplazarlo por el prototipo Engineer.



Constructores más flexibles en JavaScript

Las funciones constructor que se han mostrado hasta ahora no permiten especificar valores a las propiedades cuando se crea una instancia. Al igual que en Java, se pueden proporcionar argumentos a los constructores para inicializar los valores de las propiedades de las instancias. La siguiente figura muestra una forma de hacerlo.



Constructores más flexibles

JavaScript

```
function Employee (name, dept) {
  this.name = name || "";
  this.dept = dept || "general";
}
```

```
function WorkerBee (projs) {
  this.projects = projs || [];
}
WorkerBee.prototype = new Employee;
```

Java

```
public class Employee {
   public String name;
   public String dept;
   public Employee () {
      this("", "general");
   }
   public Employee (String name) {
      this(name, "general");
   }
   public Employee (String name, String dept) {
      this.name = name;
      this.dept = dept;
   }
}
```

```
public class WorkerBee extends Employee {
   public String[] projects;
   public WorkerBee () {
      this(new String[0]);
   }
   public WorkerBee (String[] projs) {
      projects = projs;
   }
}
```

```
function Engineer (mach) {
   this.dept = "engineering";
   this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
```

```
public class Engineer extends WorkerBee {
   public String machine;
   public Engineer () {
      dept = "engineering";
      machine = "";
   }
   public Engineer (String mach) {
      dept = "engineering";
      machine = mach;
   }
}
```

Constructores más flexibles

Estas definiciones JavaScript realizan un uso idiomático especial para asignar valores por defecto:

```
this.name = name || "";
```

El operador lógico OR de JavaScript (`||`) evalúa su primer argumento. Si dicho argumento se convierte a true, el operador lo devuelve. Si no, el operador devuelve el valor del segundo argumento. Por tanto, esta linea de código comprueba si `name` tiene un valor útil para la propiedad `name`, en cuyo caso asigna a `this.name` este valor. En caso contrario asigna a `this.name` el string vacío. En este apartado se emplea este uso idiomático por abreviación. Sin embargo puede resultar chocante a primera vista.

Nota: Esto puede no resultar según lo esperado si la función *constructor* es llamada con argumentos que se convierten a `false` (como `0` (cero) y una cadena vacía (`""`). En este caso el valor por defecto resulta elegido en lugar del valor proporcionado en la llamada al constructor.
Con estas definiciones, cuando creas una instancia de un objeto, puedes especificar valores para las propiedades definidas localmente. Puedes utilizar la siguiente sentencia para crear un nuevo `Engineer`:

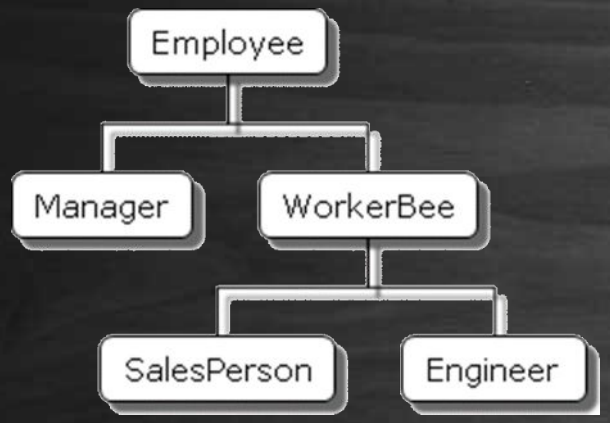
```
var jane = new Engineer("belau");
```

Ahora las propiedades de `jane` son:

```
jane.name == "";
jane.dept == "engineering";
jane.projects == [];
jane.machine == "belau"
```

Nota que con estas definiciones no puedes dar un valor inicial a las propiedades heredadas como `name`.
Si quieres especificar un valor inicial para las propiedades heredadas en JavaScript tienes que que añadir más código a la función constructora.

Constructores más flexibles



Hasta ahora, la función constructora ha creado un objeto genérico y ha especificado propiedades y valores locales para el nuevo objeto. Puedes hacer que el constructor añada más propiedades llamando directamente a la función *constructor* de un objeto que esté más arriba en la cadena de prototipos. La siguiente figura muestra estas definiciones.

Veamos los detalles de una de estas definiciones. Aquí tenemos la nueva definición del *constructor* `Engineer`:

```
function Engineer (name, projs, mach) {
this.base = WorkerBee;
this.base(name, "engineering", projs);
this.machine = mach || "";
}
```

Supongamos que se crea un nuevo `Engineer` de esta forma:

```
bar jane = new Engineer("Doe, Jane", ["navigator", "javascript"],
"belau");
```

Object hierarchy	Individual objects

Employee
`function Employee(name, dept){`
`this.name = name || "";`
`this.dept = dept || "general";`
`}`
	`var jim = new Employee("Jones, Jim", "marketing");`
	`// jim.name is "Jones, Jim"`
	`// jim.dept is "marketing"`

Manager
WorkerBee
`function WorkerBee(projs){`
`this.projects = projs || [];`
`}`
`WorkerBee.prototype = new Employee;`
	`var mark = new WorkerBee(["javascript"]);`
	`// mark.name is ""`
	`// mark.dept is "general"`
	`// mark.projects is ["javascript"]`

SalesPerson
Engineer
`function Engineer(mach){`
`this.dept = "engineering";`
`this.machine = mach || "";`
`}`
`Engineer.prototype = new WorkerBee;`
	`var jane = new Engineer("belau");`
	`// jane.name is ""`
	`// jane.dept is "engineering"`
	`// jane.projects is []`
	`// jane.machine is "belau"`

Constructores más flexibles

```
var jane = new Engineer("Doe, Jane",
["navigator", "javascript"], "belau");
```

JavaScript sigue los siguientes pasos:

El operador new crea un nuevo objeto genérico y le asigna su propiedad __proto__ a Engineer.prototype.
El operador new pasa el nuevo objeto al *constructor* Engineer como el valor de la palabra reservada this. El *constructor* crea una nueva propiedad llamada base para ese objeto y le asigna el valor del constructor WorkerBee. Esto hace que el constructor WorkerBee pase a ser un método del objeto Engineer. El nombre de esta propiedad (base) no es especial. Puede usarse cualquier nombre de propiedad, si bien base evoca el uso que se le va a dar.

Veamos los detalles de una de estas definiciones. Aquí tenemos la nueva definición del *constructor* `Engineer`:

```
function Engineer (name, projs, mach) {
this.base = WorkerBee;
this.base(name, "engineering", projs);
this.machine = mach || "";
}
```

Supongamos que se crea un nuevo `Engineer` de esta forma:

```
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"],
"belau");
```

El constructor llama al método base, pasándole como argumentos dos de los argumentos que se le han pasado al constructor ("Doe, Jane" y ["navigator", "javascript"]) y también el string "engineering". Usar explícitamente "engineering" en el constructor indica que todos los objetos Engineer tienen el mismo valor para la propiedad heredada dept, y este valor reemplaza el valor heredado de Employee.
Como base es un método de Engineer, en la llamada a base, JavaScript liga la palabra this al objeto creado en el paso 1. De esta forma, la función WorkerBee a su vez pasa los argumentos "Doe, Jane" y "engineering" a la función constructor Employee. Cuando retorna la llamada de la función constructor Employee, la función WorkerBee utiliza el resto de argumentos para asignarle un valor a la propiedad projects.
Cuando la llamada al método base retorna, el constructor Engineer inicializa la propiedad machine del objeto con el valor "belau".
Una vez creado, JavaScript asigna el nuevo objeto a la variable jane.

Constructores más flexibles

Podrías pensar que al haber llamado al constructor WorkerBee desde el constructor Engineer ya dejas establecida la herencia para los objetos Engineer. Pero no es así. Al llamar al constructor WorkerBee se garantiza que un objeto Engineer comience con las propiedades especificadas en todas las funciones del constructor que se llaman.

Pero si luego se añaden propiedades a los prototipos de Employee o de WorkerBee, estas propiedades no se heredan por los objetos Engineer. Por ejemplo, veamos las siguientes sentencias:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

El objeto `jane` no hereda la propiedad `specialty` añadida al prototipo de `Employee`. Sigue siendo necesario dar valor al prototipo de `Employee` para que la herencia buscada se establezca. Veamos las siguientes sentencias:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;
var jane = new Engineer("Doe, Jane", ["navigator", "javascript"], "belau");
Employee.prototype.specialty = "none";
```

Constructores más flexibles

Ahora el valor de la propiedad `specialty` del objeto `jane` si es "none".

Otra forma de llamar al constructor es mediante el uso de los métodos `call()` / `apply()`:

```
function Engineer (name, projs, mach) {
  this.base = WorkerBee;
  this.base(name, "engineering", projs);
  this.machine = mach || "";
}
```

```
function Engineer (name, projs, mach) {
  WorkerBee.call(this, name, "engineering", projs);
  this.machine = mach || "";
}
```

Usar el método Javascript `call()` da como resultado una implementación más limpia ya que `base` ya no es necesaria. Mediante `call()` se llama a la función constructor `WorkerBee` como un método, pasándole explícitamente `this`. El efecto es el mismo que el producido al llamar al constructor a través de la propiedad `base`: en la llamada a `WorkerBee`, `this` está ligado al objeto que se está creando en `Engineer`.

Herencia de propiedades revisada

Valores locales frente a valores heredados

Cuando accedes a una propiedad de un objeto, JavaScript realiza estos pasos:

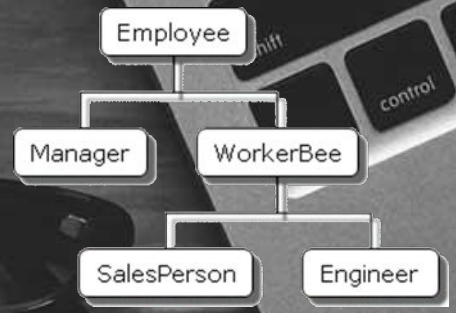
1. Comprueba si el valor existe localmente. Si existe, se devuelve ese valor.
2. Si no existe un valor local, comprueba la cadena de prototipos (usando la propiedad `__proto__`).
3. Si algún objeto en la cadena de prototipos tiene un valor para la propiedad especificada, devuelve ese valor.
4. Si no encuentra la propiedad en la cadena de prototipos, el objeto no tiene la propiedad.

El resultado de estos pasos depende de cómo se definan las cosas en el camino. El ejemplo original tenía estas definiciones:

```
function Employee ()
{
this.name = "";
this.dept = "general";
}
```

```
function WorkerBee ()
{
this.projects = [];
}
```

```
WorkerBee.prototype = new Employee;
```



Con estas definiciones, supongamos que se crea `amy` como una instancia de `WorkerBee` con la siguiente sentencia:

```
var amy = new WorkerBee;
```

El objeto `amy` tiene una propiedad local, `projects`. Los valores de las propiedades `name` y `dept` no son locales para `amy` y por eso se obtienen de la propiedad `__proto__` del objeto. Por ello, `amy` tiene estos valores en sus propiedades:

```
amy.name == "";
amy.dept == "general";
amy.projects == [];
```

Ahora supongamos que cambias el valor de la propiedad `name` en el prototipo asociado a `Employee`:

```
Employee.prototype.name = "Unknown"
```

A primera vista, esperarías que el nuevo valor se propague hacia abajo a todas las instancias de `Employee`.
Pero no es esto lo que ocurre.,,

Herencia de propiedades revisada

Cuando se crea una instancia del objeto `Employee`, ésta obtiene un valor local para la propiedad `name` (la cadena vacía). Esto significa que cuando se da valor al prototipo de `WorkerBee` mediante la creación de un nuevo objeto `Employee`, `WorkerBee.prototype` tiene un valor local para la propiedad `name`. Por tanto, cuando JavaScript busca la propiedad `name` del objeto `amy` (una instancia de `WorkerBee`), JavaScript encuentra el valor local de esa propiedad en `WorkerBee.prototype`. Por tanto no busca más arriba en la cadena hasta `Employee.prototype`

Si quieres cambiar el valor de una propiedad de un objeto en tiempo de ejecución y conseguir que el nuevo valor sea heredado por todos los descendientes del objeto, no puedes definir la propiedad en la función constructor del objeto. En su lugar, la tienes que añadir al prototipo asociado al constructor. Por ejemplo, supongamos que cambiamos el código anterior por este otro:

En este caso, la propiedad `name` de `amy` si pasa a ser "Unknown" tras la ultima sentencia.
Tal como muestran estos ejemplos, si quieres tener valores por defecto para propiedades de objetos, y se necesitas cambiar los valores por defecto en tiempo de ejecución, tienes que asignar las propiedades al prototipo del constructor, y no asignarlas dentro de la función *constructor*.

```
function Employee () {
  this.dept = "general";
}
Employee.prototype.name = "";

function WorkerBee () {
  this.projects = [];
}
WorkerBee.prototype = new Employee;

var amy = new WorkerBee;

Employee.prototype.name = "Unknown";
```

Información global en los constructores

Cuando creas constructores tienes que tener especial cuidado si se asigna información global en el constructor. Por ejemplo, supongamos que quieres tener un ID único que se asigne automáticamente a cada nuevo empleado. Podrías utilizar la siguiente definición para `Employee`:

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}
```

Con esta definición, cuando creas un nuevo `Employee`, el constructor le asigna el siguiente ID y luego incrementa el contador global ID. Por tanto, tras ejecutar el siguiente código, `victoria.id` es 1 y `harry.id` es 2:

```
var victoria = new Employee("Pigbert, Victoria", "pubs")
var harry = new Employee("Tschopik, Harry", "sales")
```

A primera vista puede parecer razonable. Sin embargo, `idCounter` se incrementa cada vez que se crea un nuevo objeto `Employee`, cualquiera que sea su propósito. Si creas la jerarquía completa de `Employee` mostrada en estos ejemplos, el constructor `Employee` es llamado cada vez que se asigna valor a un prototipo.

Constructores en JavaScript

```
var idCounter = 1;

function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   this.id = idCounter++;
}

function Manager (name, dept, reports) {...}
Manager.prototype = new Employee;

function WorkerBee (name, dept, projs) {...}
WorkerBee.prototype = new Employee;

function Engineer (name, projs, mach) {...}
Engineer.prototype = new WorkerBee;

function SalesPerson (name, projs, quota) {...}
SalesPerson.prototype = new WorkerBee;

var mac = new Engineer("Wood, Mac");
```

Supongamos además que las definiciones que se omiten tienen la propiedad `base` y se llama al constructor que tienen encima en la cadena de prototipos. En este caso, cuando se llega a crear el objeto `mac`, `mac.id` es 5. Dependiendo de la aplicación, puede o no importar que el contador se haya incrementado esas veces extra. En caso de que importe, una solución es utilizar este constructor:

```
function Employee (name, dept) {
   this.name = name || "";
   this.dept = dept || "general";
   if (name)
      this.id = idCounter++;
}
```

Cuando se crea una instancia de `Employee` para usarla como prototipo, no se especifican argumentos para el constructor. Mediante esta definición del constructor, cuando no se proporcionan argumentos, el constructor no asigna un valor al id y no actualiza el contador. Por tanto, para que se asigne a un `Employee` un id, hay que especificar un `name` al employee. En este caso `mac.id` seria 1.

Sin herencia múltiple

Algunos lenguajes orientados a objetos tienen herencia múltiple. Es decir, un objeto puede heredar las propiedades y valores de varios objetos padre distintos. JavaScript no proporciona herencia múltiple.
La herencia de valores de propiedades se produce en tiempo de ejecución por JavaScript buscando en la cadena de prototipos de un objeto para encontrar un valor. Debido a que un objeto tiene un solo prototipo asociado, JavaScript no puede heredar dinámicamente de más de una cadena de prototipos.
En JavaScript se puede hacer que desde una función constructor llame a una o más funciones *constructor*. Esto da la ilusión de herencia múltiple. Considera, por ejemplo, las siguientes definiciones:

```
function Hobbyist (hobby) {
   this.hobby = hobby || "scuba";
}

function Engineer (name, projs, mach, hobby) {
   this.base1 = WorkerBee;
   this.base1(name, "engineering", projs);
   this.base2 = Hobbyist;
   this.base2(hobby);
   this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;

var dennis = new Engineer("Doe, Dennis", ["collabra"], "hugo")
```

Consideremos, además, la definición de `WorkerBee` que se usó antes.
En este caso, el objeto `dennis` tiene estas propiedades:

Sin herencia múltiple

```
dennis.name == "Doe, Dennis"
dennis.dept == "engineering"
dennis.projects == ["collabra"]
dennis.machine == "hugo"
dennis.hobby == "scuba"
```

Por tanto `dennis` obtiene la propiedad `hobby` del constructor `Hobbyist` . Sin embargo, si luego añades una propiedad al prototipo del constructor de `Hobbyist`:

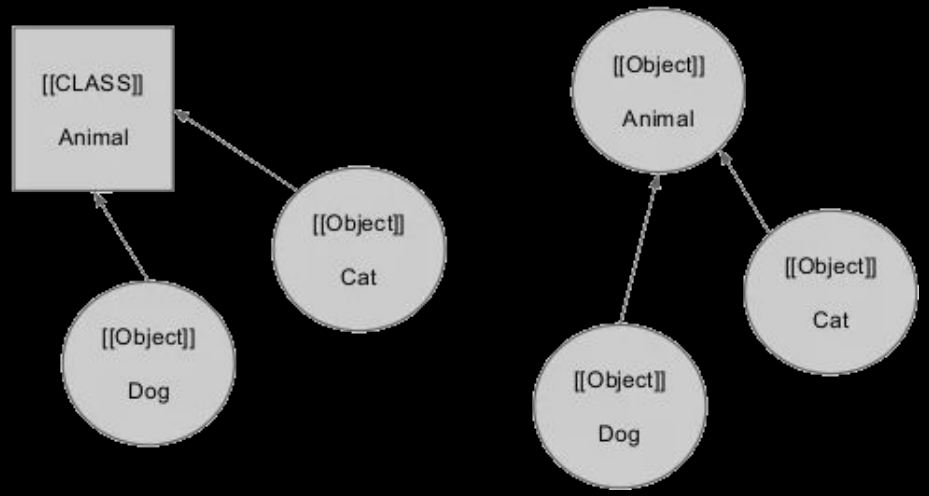
```
Hobbyist.prototype.equipment = ["mask", "fins", "regulator", "bcd"]
```

El objeto `dennis` no hereda esta nueva propiedad porque no está en su cadena de prototipos.

Resumen – Conclusión.

¿En qué se diferencia el POO basado en prototipos del basado en clases?

La programación basada en prototipos es un estilo de programación orientada a objetos en el que la herencia se implementa a través del proceso de clonación de objetos existentes que sirven como prototipos. En los lenguajes basados en prototipos, los objetos heredan directamente de otros objetos, en los lenguajes POO basados en clases más clásicos, la herencia se basa en la relación "padre-hijo" entre las clases.



Diferencia con la herencia basada en clases

En la clase, la herencia y la instanciación del lenguaje OOP se sienten como dos cosas diferentes y separadas. Cuando se define una subclase, se crea una nueva clase que hereda los miembros y el comportamiento de su clase base y, a continuación, se extiende. Después, si necesita manipular el estado y los datos de su subclase recién creada, debe crear una instancia. Entonces, y solo entonces, puedes tratarlo como un objeto.
En el caso de la POO prototípica, los actos de "herencia" e "instanciación" se sienten como el mismo proceso. No hay noción de una "clase", por lo que simplemente omita el proceso de definición de una clase. Solo hay objetos. Los instancias en el momento de la creación.

Por ejemplo, en términos de POO basada en clases, y son clases base e hija respectivamente. Por lo general, es una abstracción, que describe la "fruta" general e incluso no se puede instanciar. es una clase concreta, que se puede instanciar, toma todas las propiedades de y las extiende con propiedades especiales a "manzana". En el caso de un prototipo basado en OOP no es una abstracción, es un objeto concreto así como , por lo que puede ponerlo en la cesta (matriz), realizar una operación de "comer" en él o enviarlo a un amigo (para que funcione como un argumento). Suena poco realista, ya que no se puede hacer esto para fructificar en la vida real, porque no hay "Fruta" abstracta, solo concretas. Pero lo que no está en línea con la vida real, podría ser más adecuado para las necesidades de programación

JavaScript // https://www.mycompiler.io/es/new/nodejs

```
function Employee (name) {
    this.name = name;
    this.dept = "general 22";
}

function Manager () {
    this.reports = [];
}
Manager.prototype = new Employee;

function WorkerBee (name,projs) {
    this.base = Employee;
    this.base(name);
    this.projects = projs;
}
WorkerBee.prototype = new Employee

function SalesPerson () {
   this.dept = "sales";
   this.quota = 100;
}
SalesPerson.prototype = new WorkerBee;

function Engineer (name, projs, mach) {
    this.base = WorkerBee;
    this.base(name, projs);
    this.machine = mach || "";
}
Engineer.prototype = new WorkerBee;


jane = new Engineer("Doe, Jane", ["Front End", "Back Endt"], "Maquina 1");
console.log("User: "+ jane.name);
console.log("Proyectos: "+ jane.projects);
console.log(" ");

jose = new Engineer("jose, Luis", ["C++", "javascript"], "Maquina 2");
console.log("User:"+ jose.name);
console.log("Proyectos: "+ jose.projects);
```

Este super-simple ejemplo busca mostrar la tecnología de JavaScript dentro de un Browser y comparar el interprete en ejecución.

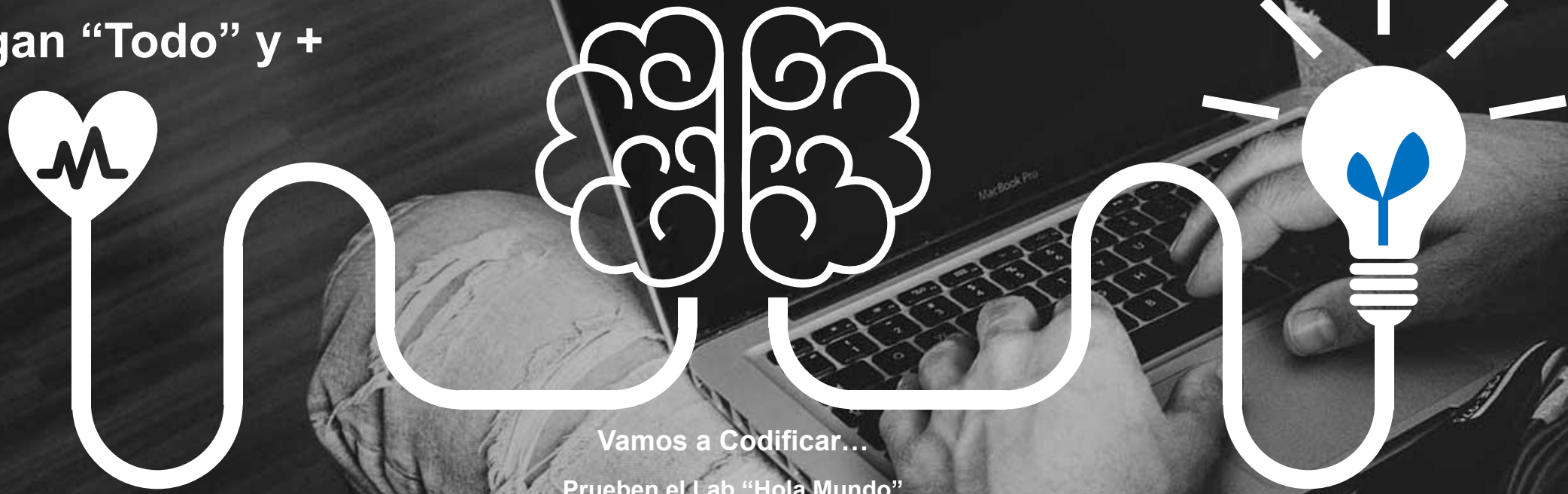
Entender que hay distintas implementacione so Interpretaciones del Paradigma que aportan soluciones diferentes…

JavaScript Objects (w3schools.com)

- Ahora todo depende de Ustedes
- Vamos al Laboratorio

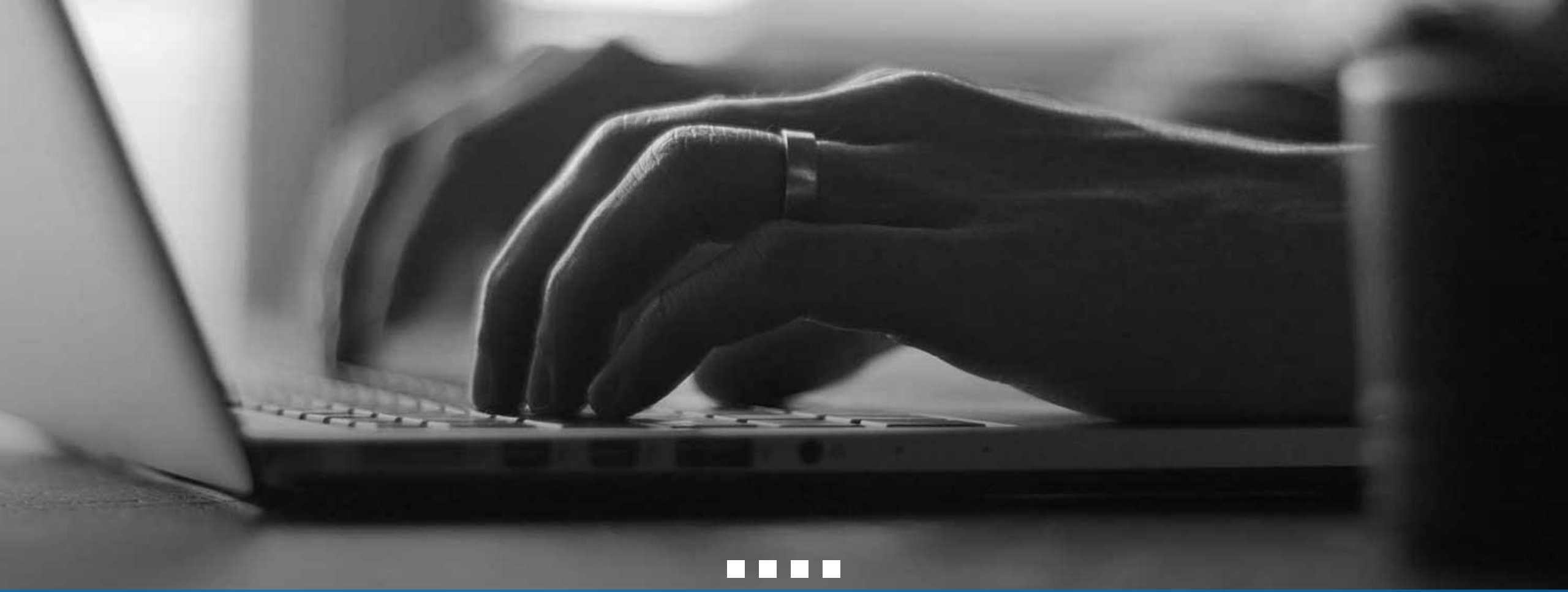
Los Resultados Llegarán

Pongan “Todo” y +



Vamos a Codificar…

Prueben el Lab “Hola Mundo” conceptualizando lo Conversado en Clase



MUCHAS GRACIAS

{Hasta la Próxima}

Muchas Gracias

Nos veremos en la Proxima.