Escuela de Ingeniería Informática
Universidad de Oviedo

Lenguajes y Paradigmas de la Programación

Tema 1

Contenido

Lenguaje de Programación

- Un **lenguaje de programación** es un lenguaje artificial utilizado para escribir instrucciones, algoritmos o funciones que pueden ser ejecutadas por un ordenador
- Es un lenguaje que permite **acercar el nivel de abstracción** del humano al nivel de abstracción de una máquina

Lenguaje Humano

Castellano
Inglés

Lenguaje de Programación

```
class Persona {
  ...
}
```

Lenguaje Máquina

1010010011010

Traductor, Compilador e Intérprete

- Un **traductor** es un programa que procesa un texto fuente (origen) y genera un texto objeto (destino)
- Un **compilador** es un traductor que transforma código fuente de un lenguaje de alto nivel al código fuente de otro lenguaje de bajo nivel
  - Un compilador es, por tanto, un caso particular de un traductor
- Un **intérprete** ejecuta las instrucciones de los programas escritos en un lenguaje de programación

Clasificación de los Lenguajes según sus Características

- Los lenguajes de programación suelen **clasificarse** en base a una serie de **características**
  1. Nivel de abstracción
  2. En función de su dominio
  3. Soporte de concurrencia
  4. En función de su implementación (compilados o interpretados)
  5. En función de cuándo se realizan las comprobaciones de tipos
  6. En función de cómo se representa el código fuente
- Profundizaremos en cada una de las clasificaciones

Nivel de Abstracción

1. **Nivel de abstracción** (alto / bajo)
   - Suele medir en qué medida el nivel de abstracción del lenguaje está más próximo al lenguaje o sistema cognitivo humano (alto nivel) o más próximo al ordenador (bajo nivel)
   - El lenguaje ensamblador y código máquina están considerados como lenguajes de **bajo nivel**
   - El C es considerado como un lenguaje de **medio nivel** por algunos autores
   - El resto de lenguajes de programación son comúnmente considerados como lenguajes de **alto nivel**
   - Los lenguajes específicos del dominio (DSLs) suelen tener **aún un mayor nivel de abstracción**

En Función del Dominio

2. **En función de su dominio**: De propósito general o específicos de un dominio (DSLs)
   - Los **DSL** son lenguajes específicos de un dominio
     - Ejemplos: SQL (bases de datos), Logo (dibujo), R (estadística) o Csound (música)
   - Los lenguajes de **propósito general** pueden ser utilizados para resolver cualquier problema computacional
     - Ejemplos: Java, C++, Python, C#, Pascal...
   - Los DSLs han aumentado su uso en los últimos años debido al Modelado Específico del Dominio (DSM) y Desarrollo Dirigido por Modelos (MDD)

Soporte de Concurrencia

3. **Concurrencia**: Aquellos lenguajes que permiten la creación de programas mediante un conjunto de procesos o hilos que interaccionan entre sí y que *pueden* ser ejecutados de forma paralela
   - Los programas concurrentes pueden ser ejecutados:
     - **En un único procesador**, intercalando la ejecución de cada uno de los procesos
     - **En paralelo**, asignando cada proceso o hilo a procesadores distintos que pueden estar en una misma máquina o distribuidos en una red de computadores
   - Lenguajes concurrentes: Go, Ada, Erlang
   - Muchos lenguajes no ofrecen primitivas de concurrencia como parte de ellos, pero sí como parte de su librería **estándar**: Java, C#, C++11, Python, Smalltalk
     - No concurrentes: C (¡el más usado en supercomputación!) y C++ 2003

En Función de su Implementación

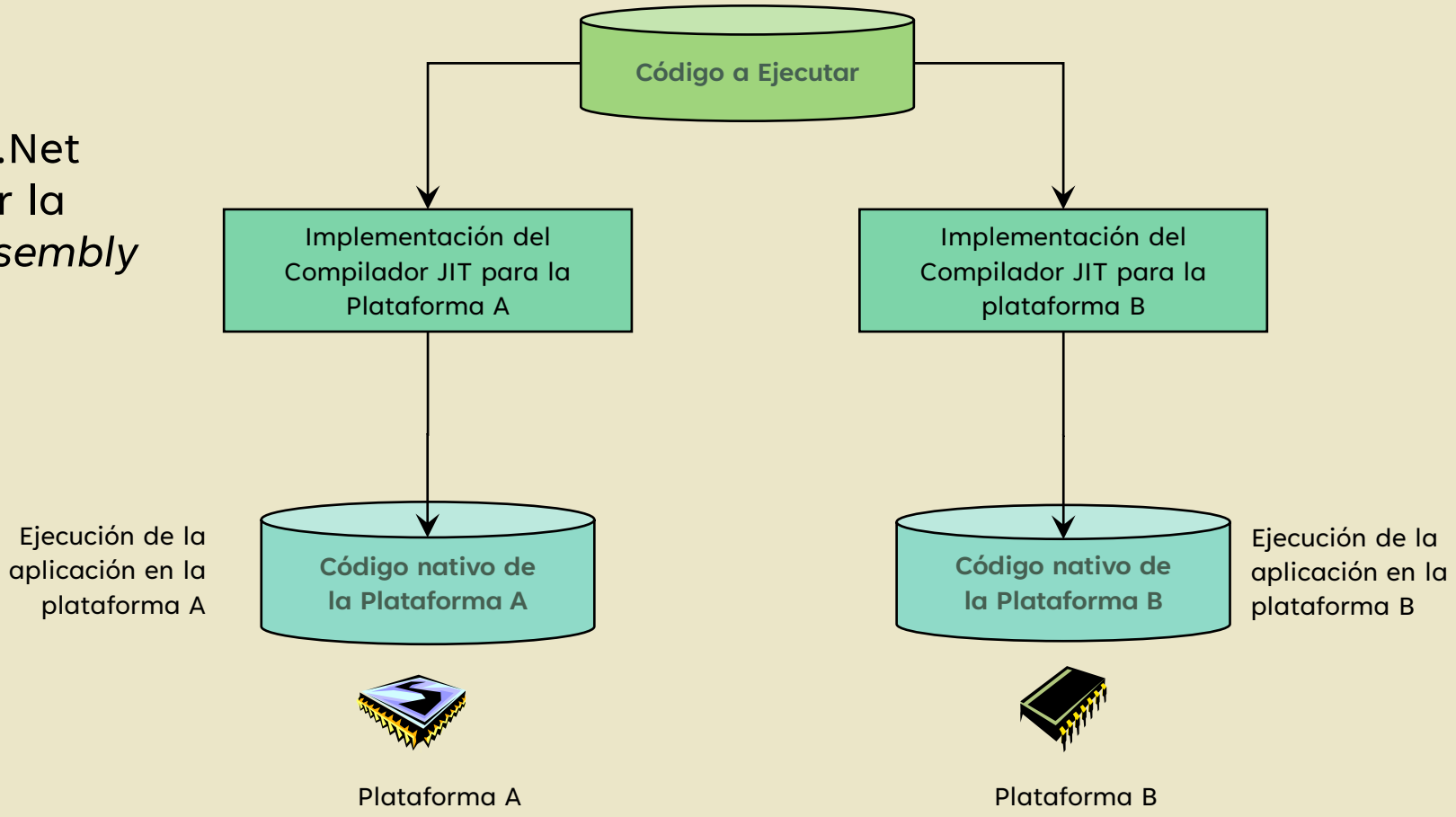
4. **Implementación:** Compilados o Interpretados

   Esta es una característica más ligada a su implementación que al lenguaje en sí

   - La clasificación no es necesariamente excluyente: Java y C# son compilados e "interpretados"
   - Con la existencia de **compilación JIT** (*Just in Time*) esta diferenciación se ve difuminada
     - Un compilador JIT transforma el código a interpretar en código nativo propio de la plataforma donde se ejecuta el intérprete, justo antes de su ejecución: Self, Visual Works Smalltalk, Java, .Net
     - Si esta transformación se realiza previamente a la ejecución se denomina **AoT** (*Ahead of Time)*

Compilación JIT

- NGen es una utilidad de .Net que nos permite controlar la compilación JIT de un *assembly* (soporta AoT)



En Función de su Implementación

- Lenguajes de naturaleza interpretada: Perl, Tcl, JavaScript, Matlab, PHP
- Lenguajes de naturaleza compilada: C, C++, Go, Pascal, Eiffel, Ada
- Lenguajes diseñados para ser primero compilados y luego interpretados: Java, UCSD Pascal
- Lenguajes diseñados para ser **compilados JIT**: Self, C#
- Lenguajes con implementaciones tanto compiladas a código nativo como interpretadas: Haskell, Lisp

Sistema de Tipos

5. **En función de cuándo se realizan las comprobaciones de tipos**

   Las comprobaciones de tipo (comprobaciones de las operaciones que pueden ser aplicadas a una variable u objeto) pueden ser **estáticas** (en tiempo de compilación) o **dinámicas** (en tiempo de ejecución)

   - No es una clasificación excluyente: Java es principalmente tipado estáticamente pero también ofrece tipado dinámico
   - La comprobación **estática** de tipos suele ofrecer
     - Detección temprana de errores (tiempo de compilación)
     - Un mayor rendimiento (no se realizan comprobaciones en tiempo de ejecución)
   - La comprobación **dinámica** suele ir ligada a:
     - Una mayor adaptabilidad (flexibilidad) dinámica
     - Metaprogramación dinámica

Lenguajes Script y Dinámicos

- Los lenguajes con comprobación dinámica de tipos son comúnmente denominados:
  - Lenguajes de ***Scripting***: Lenguajes con comprobación dinámica de tipos comúnmente orientados a la unión (*glue*) de componentes
    - (ba)sh, Awk, Perl, Tcl, ActionScript, Lingo, JavaScript...
  - **Lenguajes dinámicos**: poseen características (acceso a bases de datos, GUIs, modulación, *threading* nativo) que permiten desarrollar aplicaciones finales utilizando sólo ese lenguaje de programación
    - Smalltalk, CLOS, Python, Ruby, Dylan, Groovy, Lua, JavaScript
- Es común ver ambos términos como sinónimos

Sistemas de tipos vs. Compilación

- Los lenguajes con sólo comprobación estática son compilados
- No existen lenguajes interpretados puros con sólo comprobación estática
- Existen varios lenguajes compilados con comprobación dinámica
- Los lenguajes que sólo poseen comprobación dinámica tienen una fase de interpretación

	Sólo comprobación estática	**Comprobación Estática y Dinámica**	**Sólo Comprobación Dinámica**
Compilados	C, Fortran, Pascal	C++ (RTTI), Ada, ObjectiveC, Scala	Python, Ruby 1.9+ (compilados e interpretados)
Interpretados		GHCi Haskell, OCaml (compilados e interpretados)	Smalltalk, Prolog, Ruby 1.8-

Representación del Código Fuente

6. **En función de cómo se representa el código fuente**

   Puede tratarse de lenguajes visuales o “textuales” (o no visuales)

   - Los lenguajes **visuales** representan las entidades de su dominio mediante una notación visual, en lugar de textual
   - Ejemplos de lenguajes visuales son (Executable) UML, Lava, LabVIEW y VisSim
   - Existen numerosos lenguajes visuales específicos de un dominio: Macromedia Authorware (multimedia), Game Maker (juegos), Ladder Logic (electrónica), MVPL (Robótica)
   - Los lenguajes **textuales** usan ficheros de texto para contener su código fuente
     - Ejemplos: Java, C#, Haskell, Pascal...

Paradigmas de programación

Breve introducción a los distintos paradigmas de programación existentes actualmente

Paradigmas de Programación

- Hemos visto características de los lenguajes de programación que suelen utilizarse en su clasificación
- Un **paradigma** de programación es una **aproximación** para construir programas **caracterizada** por las **abstracciones** y conceptos utilizados para **representar** dichos **programas** (objetos, funciones, restricciones, predicados)
  - El paradigma de programación se usa también para clasificar lenguajes de programación
  - Un lenguaje de programación puede seguir más de un paradigma (**multiparadigma**)
  - Un **paradigma de programación no constituye una característica** de un lenguaje

Imperativo vs. Declarativo

- Determinados autores los consideran en sí paradigmas
- Más bien suponen una **clasificación de los paradigmas**
  - En algunos casos, incluso, de lenguajes
- Un paradigma **imperativo** describe los programas en términos de **sentencias** que cambian **estado del programa** (o de la máquina)
  - El programador debe especificar al ordenador **cómo** realizar una tarea
  - Sus abstracciones suelen ser más cercanas al ordenador que los lenguajes declarativos
  - Ejemplos de lenguajes: C, Java, C#, Pascal
  - Ejemplos de paradigmas imperativos: Estructurado basado en procedimientos (a veces llamado simplemente imperativo) y orientado a objetos

Imperativo vs. Declarativo

- Los paradigmas **declarativos** se basan en escribir programas que especifican **qué** se quiere obtener, intentando no especificar cómo
  - El programador ***declara*** lo que quiere obtener
- Se utilizan las **abstracciones** del paradigma específico para representar el "qué"
  - Funciones, predicados, restricciones, consultas
- Ejemplos de paradigmas comúnmente identificados como declarativos son lógico, funcional, basado en restricciones y de rescritura de reglas y árboles
- Ejemplos de lenguajes declarativos: Prolog, SQL, CLP(R), Maude, Haskell

Principales Paradigmas

- **Estructurado** basado en **procedimientos**: Paradigma imperativo en el que se utilizan tres estructuras de control (secuencial, condicional e iterativa), pudiendo agrupar éstas en procedimientos o subrutinas
- **Orientado a objetos**: paradigma comúnmente imperativo que utiliza los objetos, unión de datos y métodos, como principal abstracción, definiendo programas como interacciones entre objetos
- **Funcional**: Paradigma declarativo basado en la utilización de funciones que manejan datos inmutables
- **Lógico**: Paradigma declarativo basado en la programación de ordenadores mediante lógica matemática

Estructurado Basado Procedimientos

- Este paradigma es comúnmente llamado simplemente **imperativo**
- Viene de la unión de **dos paradigmas históricos**
  - Paradigma estructurado
  - Paradigma procedural
- Paradigma **estructurado**: paradigma en el que se utilizan tres estructuras de control; secuencial, condicional e iterativa
  - Se **evitan** las instrucciones de **salto**, tanto condicionales como incondicionales
  - Se aumenta la **legibilidad** y **mantenibilidad** de los programas
  - Las estructuras de control se pueden anidar

Estructurado Basado Procedimientos

- Define el **procedimiento** o **subrutina** como el primer mecanismo de **descomposición**
  - Un procedimiento contiene una lista ordenadas de instrucciones
- **Incorporó la programación estructurada**
  - Por ello constituyen un único paradigma
- Incluye el concepto de **ámbito**, previniendo el acceso indebido a variables
- Un procedimiento que devuelve un valor se define como una **función**
- Importante: Este concepto es distinto al concepto de **función matemática** definido en la programación funcional
  - Pueden generar efectos colaterales
  - No suelen ofrecer funciones de orden superior
  - No implementan el concepto de cláusulas

Estructurado Basado Procedimientos

- Existen multitud de lenguajes estructurados basados en procedimientos
  - Algol, Ada, C, Pascal, Fortran, Cobol, PL/I
- Fortran, el primer lenguaje de alto nivel creado por IBM en 1957, sigue este paradigma
  - En sus primeras versiones, toda la memoria era estática

Orientación a Objetos

- Utiliza los objetos, unión de datos y métodos, como principal abstracción, definiendo programas como interacciones entre objetos
- Se basa en la idea de modelar **objetos reales**, o introducidos en diseño, mediante la codificación de **objetos software**
  - La idea es acercar el modelo del dominio al modelo del programa
- Un programa está constituido por un conjunto de objetos pasándose mensajes entre sí (interactuando)
- Típicamente es un paradigma **imperativo**
- Los **elementos** propios de este paradigma son el encapsulamiento, herencia, polimorfismo y enlace dinámico

Orientación a Objetos

- Existen dos modelos computaciones de orientación a objetos
  1. **Basados en clases**
     - Todo objeto tiene que ser **instancia** de una clase
     - La clase es el **tipo** del objeto
     - Las clase define la **estructura** y **comportamiento** de sus instancias
     - C++, Java, C#, Smalltalk, Eiffel
  2. **Basados en prototipos** (basados en objetos)
     - No existen el concepto de clase, los objetos son la única entidad
     - Se define un objeto **prototipo** y el resto de objetos con una estructura similar se **clonan** a partir de éste
     - Self, Cecil, JavaScript, Lua
- Un **lenguaje OO** se dice **puro** cuando toda abstracción ofrecida es un objeto: Smalltalk, Eiffel, Ruby

Paradigma Funcional

- Paradigma declarativo basado en la utilización de funciones que manejan **datos inmutables**
  - Los datos nunca se modifican
  - En lugar de ello, se llama a una función que devuelve el dato modificado sin modificar el original
- Un **programa** se define mediante un **conjunto de funciones invocándose entre sí**
- Las funciones no generan **efectos** (**co**)**laterales** (secundarios, *side effects*):
  - el valor de una expresión depende exclusivamente de los valores de los parámetros
  - devolviendo siempre el mismo valor en función de éstos (paradigma funcional puro)
- Se hace uso exhaustivo de la **recursividad**, en lugar de la iteración

Paradigma Funcional

- Los **lenguajes funcionales puros** no utilizan ni la asignación ni la secuenciación de instrucciones
- Sus orígenes se remontan al **cálculo lambda** definido por Church y Kleene en los años 30
- En la actualidad, los lenguajes comerciales están incorporando elementos de este paradigma
- Ejemplos de lenguajes funcionales puros: Miranda, Clean y Haskell
- Ejemplos de lenguajes funcionales: Scheme, Lisp, Erlang, ML (Standard ML, CamL, F#, OCaml)

Paradigma Lógico

- Paradigma declarativo basado en la programación de ordenadores mediante **lógica matemática**
- El programador describe un conocimiento mediante **reglas lógicas** y axiomas (hechos)
- Un demostrador de teoremas con **razonamiento hacia atrás** resuelve problemas a partir de **consultas**

```
madre(teresa, sandra).
madre(sandra, juan).
madre(teresa, tomas).
padre(tomas, sandra).
padre(miguel, tomas).
padre(tomas, elisa).
```

Hechos (axiomas)

Reglas

```
hermano(X, Y) :- padre(Z, X), padre(Z, Y), \=(X, Y).
hermano(X, Y) :- madre(Z, X), madre(Z, Y), \=(X, Y).
```

```
?- hermano(sandra, X).
X = elisa, X = tomas.
```

Consulta y solución

Paradigma Lógico

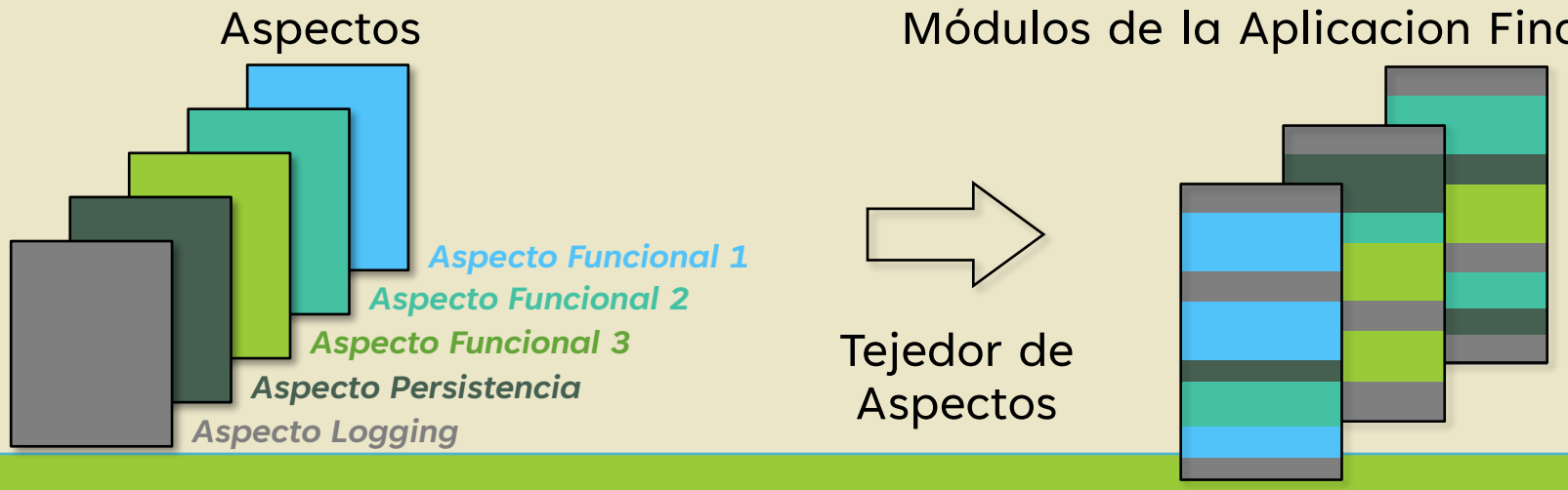
- Fue descrito por primera vez en 1958 por John McCarthy (creador del Lisp)
- El primer lenguaje considerado lógico fue Absys desarrollado en 1967
- El lenguaje de programación lógica por excelencia es **Prolog** (años 70), que cuenta con diversas variantes
  - **Concurrente**: Brain Aid Prolog (BAP), Muse, Aurora
  - Basado en **restricciones**: B-Prolog, CLP(R), Ciao Prolog, SICTus
- Existen otros lenguajes **lógico-funcionales** como ALF, Mercury, Mozart – Oz, Life y Toy

Otros paradigmas de programación

- Existen otros paradigmas de programación menos difundidos:
  - Orientación a aspectos
  - Basado en restricciones
  - Tiempo real
  - Guiado por eventos
  - Programación basada en autómatas
  - Programación reactiva

Orientación a Aspectos

- La programación orienta a aspectos consiste en **modularizar** la **funcionalidad** de una aplicación que se encuentre **entremezclada** y **dispersa** a lo largo del código
  - Ejemplos de aspectos son persistencia, seguridad, logging, tracing...
- Las herramientas que dan soporte a aspectos se basan, mayoritariamente, en técnicas de **instrumentación** o procesamiento (**tejido**) de **programas**



Orientación a Aspectos

- Es un paradigma **imperativo**
- El incremento de modularidad ofrecido por la orientación a aspectos ofrece **beneficios** en el desarrollo de software tales como

  Mantenibilidad, legibilidad, reusabilidad (de aspectos y componentes) y adaptabilidad
- **AspectJ** es la herramienta (lenguaje) más utilizada para programar en este paradigma
- La orientación a aspectos ha sido incluida en servidores de aplicaciones comerciales tales como JBoss y Spring
- **AOSD** *(Aspect Oriented Software Development)* es la disciplina que sigue la separación de aspectos en todas las fases del ciclo de vida de desarrollo de software (http://www.aosd.net)

Basados en Restricciones

- Las relaciones entre las variables están expresadas mediante **ecuaciones** (restricciones)
  - La ejecución del programa debe dar (conjuntos de) valores como resultado
- Es un paradigma **declarativo**
- Las restricciones se expresan sobre distintos **dominios**: booleano, entero, racional, lineales, finitos
- Suelen implementarse como
  - Un **lenguaje** propiamente dicho: ECLiPSe, Mozart – Oz, OPL
  - Una **ampliación de otro lenguaje**, comúnmente lógico: CLP(R), B-Prolog, Ciao Prolog, SICTus
  - Un **API** de un lenguaje imperativo: Comet, Disolver, Gecode, ChocoSolver

Basados en Restricciones

- Lo siguiente es un ejemplo en OPL (*Optimization Programming Language*) para resolver el problema del coloreado de mapas

```
enum Country {Belgium, Denmark, France, Germany,
              Netherlands, Luxembourg};
enum Colors {blue, red, yellow, gray};
var Colors color[Country];
solve {
     color[France] <> color[Belgium];
     color[France] <> color[Luxembourg];
     color[France] <> color[Germany];
     color[Luxembourg] <> color[Germany];
     color[Luxembourg] <> color[Belgium];
     color[Belgium] <> color[Netherlands];
     color[Belgium] <> color[Germany];
     color[Germany] <> color[Netherlands];
     color[Germany] <> color[Denmark];
};
```

Tiempo Real

- Un sistema en tiempo real es aquél que tiene que llevar a cabo **computaciones en un tiempo máximo dado**, independientemente de la carga del sistema
- Hay dos tipos de tiempo real
  - ***Hard real-time***: Aquellos que no realizar la operación en el tiempo esperado implicaría un fallo crítico del sistema
  - ***Soft real-time***: Aquellos que, de no realizar la operación en el tiempo esperado, proveen un mecanismo para recuperarse (dejar de computar determinados datos)
- Los lenguajes de programación en tiempo real tienen que tener en cuenta estas restricciones: Ada, Real-Time Java
- Es común programar en lenguajes compilados nativos con un API del SO: C + Real-Time Posix

Guiado por eventos

- En este paradigma el flujo de ejecución está **determinado por eventos**
- Los eventos pueden ser **acciones de usuarios** (clicks, pulsaciones de teclas), datos leídos de **sensores** o mensajes enviados por **otros programas** o **hilos**
- Es adecuado para cualquier programa que se base en ejecutar ciertas acciones en función de las entradas de los usuarios, como drivers de dispositivo o GUIs
- Las aplicaciones normalmente consisten en un bucle principal que escucha eventos y código en forma de **callback** que se ejecuta cuando se detecta su evento asociado
- Se pueden escribir programas que sigan este paradigma en casi cualquier lenguaje, pero es más sencillo si el lenguaje soporta abstracciones de alto nivel como *await* o clausulas
- La mayor parte de herramientas o arquitecturas para desarrollar GUIs la usan
  - **Java AWT framework** procesa todos los cambios a la interfaz de Usuario en un solo hilo (*Event dispatching thread*)
  - Todas las actualizaciones de la interfaz de usuario en el framework **JavaFX** se hacen en el *JavaFX Application Thread*
  - **Node.js** también está basado en eventos, y generalmente las aplicaciones web que usen JavaScript también

Programación basada en autómatas

- Los programas (o partes de un programa) son una **máquina finita de estados** (FSM) o bien **cualquier autómata formal** definido por la teoría de autómatas
  - Por tanto, los programas son un **ciclo de pasos definidos por dicho autómata**
  - *FSM-based programming* se refiere normalmente a lo mismo, aunque es un término menos general
- Los programas que siguen este paradigma tienen las siguientes propiedades
  - El tiempo de ejecución de un programa está separado en los pasos del autómata
  - Cada paso consiste en la ejecución de una sección de código que tiene un único punto de entrada
    - Cada paso podría dividirse en subpasos que se ejecutan dependiendo del estado actual
  - Cualquier comunicación entre pasos solo es posible mediante un conjunto de variables especial llamado **estado**
  - Entre dos estados un programa no puede tener otros componentes que determinen su estado que no sean estas variables de estado, como variables en la pila, direcciones de retorno, el puntero de instrucciones...
    - Por tanto, el estado del programa completo depende solamente de estas variables de estado
- **Usos**:
  - Se usa ampliamente en análisis **léxico**, **sintáctico** y para describir la **semántica** de algunos lenguajes de programación
  - Necesario en la programación guiada por eventos para implementar paralelismo
  - Los conceptos de máquina finita de estados y estados se usan habitualmente en el campo de la **especificación formal**
    - UML usa diagramas de estado para describir el comportamiento de los programas
    - Varios protocolos de comunicación crean su especificación usando el concepto de estado

Programación reactiva

- Es un paradigma de programación declarativo que usa los conceptos de **flujos de datos** y la **propagación del cambio**
- Permite expresar fácilmente flujos de datos estáticos (*arrays*) o dinámicos (emisores de eventos), así como expresar que existe una dependencia entre ellos con un modelo de ejecución asociado
  - Esto facilita la propagación automática de los cambios a los flujos de datos
  - Por ese motivo, se está popularizando en la actualidad
- Se basa en el patrón de diseño ***Observer***
  - Un objeto (***Publisher***) se comunica con muchos otros (***Subscribers***) enviándoles (*push*) o dejándoles disponibles (*pull*) mensajes hacia o para ellos
  - El conjunto de mensajes enviados por el *publisher* a sus *suscribers* se llamada ***data flow***
- **ReactiveX** o Rx es **el API más popular** para crear programas que usen programación reactiva
- Se basa en los patrones de diseño `Observable` e `Iterator` y en la **Programación Funcional**
- Rx tiene librerías para distintos lenguajes:
  - **RxJS** (Javascript): https://medium.freecodecamp.org/an-introduction-to-functional-reactive-programming-in-redux-b0c14d097836
  - **RxJava** (Java): https://github.com/ReactiveX/RxJava

Programación reactiva

- En .NET la programación reactiva se basa en dos interfaces existentes
  - Los *Publishers* implementan `IObservable<T>`
  - Los *Subscribers* implementan `IObserver<T>`
  - T representa el tipo de mensaje que un *publisher* envía a sus *suscribers*
- No obstante, para poder hacer programas que usen programación reactiva hay que **instalar el paquete NuGet `System.Reactive`**: `Install-Package System.Reactive`

```
var data = ReadDataFromInputAsynchronously(); //IEnumerable<int>
IObservable<int> observableData = data.ToObservable();

observableData.Subscribe(
   onNext: number => { Console.Write($"Se ha recibido: {number}\n "); },
   onCompleted: () => { Console.WriteLine("\nFin de la secuencia de números"); }
);
```

- Este programa imprime todos los números a medida que van llegando del origen sin necesidad de usar un bucle
- En .Net la programación reactiva se puede combinar con LINQ (ver tema de programación funcional)

Tecnología de la Programación

Alternativa seleccionada para impartir esta asignatura

Tecnología de la Programación

- El término "*tecnología de la programación*" es demasiado amplio
- En esta asignatura lo tendremos en cuenta como

  *aquellos **elementos** y **técnicas** ofrecidos por los distintos **paradigmas** y **lenguajes** de programación para **diseñar**, **construir** y **mantener** aplicaciones de forma **correcta**, **robusta**, **segura** y **eficiente***

- Veremos los fundamentos de los paradigmas **orientado a objetos**, **funcional** y **concurrente**
- Para cada paradigma,
  - Estudiaremos los elementos y técnicas ofrecidos para construir aplicaciones de forma adecuada
  - Analizaremos las **tendencias** existentes en la actualidad para acercar los lenguajes comerciales al **paradigma declarativo**

Lenguaje a Utilizar

- Podríamos dar **varios lenguajes** para aprender los conceptos...
  - ...pero resultaría **difícil** en el tiempo limitado que tenemos (6 ECTS)
- Hemos elegido un lenguaje
  - Que ofrezca múltiples elementos y técnicas de **distintos paradigmas de programación**
  - Que posea **distintas implementaciones** sobre distintas plataformas
  - Que esté descrito y **estandarizado**, de forma que terceras partes puedan realizar implementaciones estándar
  - Que posea un **alto uso comercial** para que el alumno pueda utilizarlo como parte de su carrera profesional

C#

- Elementos del **paradigma OO**: encapsulamiento, herencia, polimorfismo, genericidad, autoboxing
- Elementos del **paradigma funcional**: funciones lambda, funciones de orden superior, cláusulas, continuaciones
- API estándar de programación **concurrente** y **paralelización**
- Elementos incluidos en las nuevas **tendencias de lenguajes de programación**: reflexión, metaprogramación, generación dinámica de código, programación orientada a atributos, tipado estático y dinámico, lenguaje integrado de consulta (listas por comprensión)
- Posee implementaciones para Windows, Linux y Mac OS
- Estándar **ECMA** e **ISO**
- Lenguaje altamente utilizado a nivel profesional, con una tendencia ascendente desde que se creó (en 2002)

C#

- En base a los parámetros que hemos descrito, C# es
  - Lenguaje de alto nivel
  - Propósito general
  - Concurrente y paralelo mediante su API estándar
  - Compilado y ejecutado por una máquina virtual con compilación JIT (CLR / CoreCLR *.Net Framework*)
  - Comprobación estática y dinámica (y sistema de tipos estático y dinámico)
  - Textual (no visual)
  - Imperativo (principalmente)
  - Base orientada a objetos, pero incluye elementos de programación funcional