

CLASE 18/09/2020

De les tres maneres de estudiar programacio (Diapos 5/90), la que anem a fer servir en classe es la numero 3.

3. Programació en termes de conceptes. 10/90

Cada llenguatge te uns conceptes implementats i axi mirarem conceptes semantis i estructures d'implementacio.

11/90

Ara tenim diversos blocs en els quals cada bloc te una serie de conceptes associats. El bloc de kernel podria ser un llenguatge basic que se coses basiques. A partir d'ell, segons les nostres necessitas, buscarem en un bloc o un altre i implementariem els conceptes que ens pugueren interessar. Que un llenguatge estiga orientat, per exemple, a un llenguatge funcional i els seus conceptes, no implica que puga gastar conceptes d'altres blocs.

Un llenguatge de programacio tipo Java, pot tindre diversos blocs. Java en concret te característiques del bloc kernel, imperatiu i el OO (Orientat a Objectes).

Conceptes essencials 12/90

1Tipus i sistemes de tipus

2Polimorfisme

3Reflexió: Un programa pot consultarse a si mateix i canviar les instruccions.

4Pas de paràmetres

5Ambit de les variables

6Gestio de memoria: Tipo el garbage collector.

1. CONCEPTES**1 Tipus i sistemes de tipus**

El tipus representa els valors que pot adoptar una variable o expressió. Pej. Int, String, double... No tots els llenguatges treballen amb tipus.

-Ajuden a detectar **errors**. Pex. *Imaginem que tenim $5 + "6 4" / 6$ (int + String / int???)*. Ja que el + entre el 5 i el "6 4" podria implementarse com una concatenacio de string i un string no se pot dividir entre 6. . A això se referix amb el que ajuda a detectar errors.

-Ajuden a **estructurar** la informació, Pex. Els arrays, les llistes. Anem a servir valors amb alguna propietat en comú.

-Ajuden a manejar estructures de **dades** amb un mateix tipus mitjançant certes operacions. Operacions com la suma, la resta, la divisio... podrien anar dins del mateix tipus. Una + amb enters els suma, pero una + amb Strings, els concatena. Per això els símbols/operacions van també lligats al tipus.

Exemple de tipus de dades:

El tipus a baix nivell (en la RAM) afecta a la manera d'estructurar la informacio. Pots fer $3+2=5$ (com a int) i guardaria el 5 en RAM en Ca^2 pero per exemple si ferem $3.0 + 2.0 = 5.0$ (double) se guarda en ram com a coma flotant (al·lo que tenia mantisa). Siguent el mateix valor 5, el fet de estar un en int i l'altre en double, fa que se guardi diferent.

14/90

-Els llenguatges **tipificats** gasten tipus en les variables . Si no restringixen el rang de valors, son llenguatges **no tipificats**. *En el cas dels segons no significa que no tinguin tipus sino que totes pertanyen al mateix conjunt de tipus, tindriem enters, strings, dobles... tot en el mateix "sac"*

15/90

El sistema de tipus estableix que tipus d'associacio de variables es possible:

-El **valor** que intentes guardar en una variable te que tindre el mateix tipus de la variable on vas a guardarlo. No intentar guardar un enter en una variable de String.

-El **valor** associat a la variable pot ser daltres tipus **compatibles** relacionats amb el tipus de la variable. *En Java per exemple se pot guardar un int en un double pero no al revés. Pero jo en una variable doble podria guardar perfectament un enter.*

-D'una altra banda, el tipus pot ser:

Estático: El tipo de la variable no canvia durant la execucio. Si crees un "nom" de tipo "String" nomes podras posar lletres.

Dinámico: El tipus d'una variable cambia sobre la marcha. Quan declares la variable si li assignes un text, se fa string, si poses un nombre, se fa entera, double... Sense necessitat de dirli el tipo de variable que es. Basicament el que sol passar en Python, que les variables se declaren sense dir el tipus.

16/90

Llenguatges tipificats = Expressions de programa + Sistemes de tipus

El **sistema de tipus** ens diu si algo es correcte en el següent aspecte: $5+4+x$ per exemple seria correcte si x es un enter, pero si la variable x te assignat el tipus String, el sistema de tipus seria incorrecte encara que semànticament està ben escrit. $5+4+x$ se pot escriure, es correcte, pero el sistema de tipus se queixaria de que la x no es un enter.

La tipificació **explícita** el tipus forma part de la sintaxi. Java. `Int x = 20; String x = "Andreu"`. Tu tens que dir-li cada variable de quin tipus es.

La **implícita**, el tipus no forma part de la sintaxi. Python per exemple, que poses `x=5` i `x=Andreu` i el tipo varia sense tindre que dirli que es.

17/90

Esta diapositiva te exemples de llenguatges no tipificats (com Prolog), llenguatges tipificats (com Java) i altres de tipificació implícita... Pero... ¿que es la tipificacio implicita?

La tipificació implícita fa que una variable sapiga quin tipus li correspon basant-se amb el que ha passat abans. En el exemple de la diapos, se veu que possa fac `0 = 1`, aleshores sap que fac es un enter. Com en la següent linea posa fac `x= x...` La tipificacio implicita permet que sapiga que la x sera un enter (encara que la x en un principi tindria que ser un char), pero com es una crida recursiva, pues sap que es un enter.

18/90 **Esta diapos es important.**

Esta explicada en la clase de 18/Setiembre mas o menos sobre el minuto 28

Llenguatges d'expressions de tipus

Noms de tipus: Char, int, bool, double...

Existeixen **variables de tipus** que son capaces de guardar "noms de tipus". Podem dir que una variable "a" te assignat el tipus bool. $a = \text{bool}$.

Constructors de tipus:

→ per a definir funcions

x per a definir parells o tuples

[] per a definir arrays o llistes

Definit tot açò tenim les regles de construcció d'Expressions, que combinen lo anterior.

Regles de construcció d'expressions:

Classe2 18/9

The diagram consists of a large circle on the left containing the words "Int", "Bool", and "Char" stacked vertically. To the left of the circle is the word "tipus". To the right of the circle is a list of variables: "a", "b", and "c". Further to the right is a list of type constructors: "Bool", "Char", "Int", and a recursive rule $\tau \rightarrow \tau$. The text "Classe2 18/9" is in the top left corner of the diagram area.

Partim de que en un conjunt tenim els noms de tipus primitius (dins el cercle)

La lletra grega Tau τ es una **regla de producció** i pot ser un bool, un char, un int... Segueix la següent sintaxi:

$\tau ::= \text{bool} \text{ o } \text{char} \text{ o } \text{int} \text{ ó } \tau \rightarrow \tau \text{ ó } \tau \times \tau \text{ ó } [\tau]$

Arribat aci, a tau se li pot unir un altre tau. Donat algo que esta en tau, afegir algo de tau:

$\tau \rightarrow \tau$

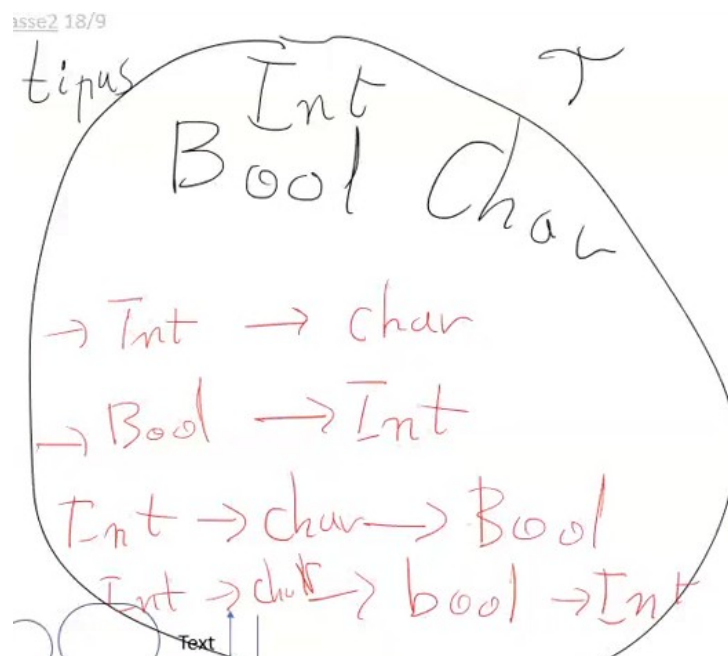
$T \rightarrow T$ significa... Agafa un element que esta dins el conjunt (per exemple `int`), ficar el constructor de tipus \rightarrow i despres agafar un altre element del conjunt (`char` per exemple)

$\text{int} \rightarrow \text{char}$

Ale, ja tenim un tipus nou anomenat " $\text{int} \rightarrow \text{char}$ " que el que fa es rebre un enter i transformar-lo en un caràcter. Per exemple clavar un enter en ASCII i ens torna un caràcter.

Mitjançant la regla de produccio tau, podem definir **infinit tipus recursivament**. Perque tu a la hora de crear una nova regla, pots agafar una regla ja creada. Tornant dos parragrafs enrere, ja teniem la regla " $\text{int} \rightarrow \text{char}$ " creada en el nostre conjunt aleshores jo podria crear per exemple:

Pex; $(\text{int} \rightarrow \text{char}) \rightarrow \text{bool}$ ó $(\text{int} \rightarrow \text{char}) \rightarrow \text{char}$



El que fa $\tau \times \tau$, es definir parells. Si tenim " $\text{char} \times \text{bool}$ " pues podria guardar parells de valor caràcter i V/F, $\tau \times \tau$ ens permet enriquir encara mes el nostre conjunt.

Si afegim els arrays $[\tau]$ més extens encara.

Per acabar, tornant a les **variables de tipus** que he mencionat al principi de la diapos. Aquestes variables poden enmagatzemar primitius o valor tau. Pex. " $a = \text{bool}$ " ó " $b = \text{int} \rightarrow \text{char}$ "

19/90

Tipus de dos tipus (min 48)

Un **tipus monomorfic o monotipus** es aquell que no te variables de tipus. Pex. “ $\text{int} \rightarrow \text{char}$ ”

Un **tipus polimorfic o politipus** es una variable amb infinits tipus. Per exemple “ $a \rightarrow \text{int}$ ”. La a pot ser, *int, double, boolean, char* o qualsevol combinació de taus lo qual ho fa infinit. La regla “ $a \rightarrow \text{int}$ ” el que fa es demanar totes les funcions dins del conjunt que tornen un *int*.

21/90

Exemplifica monomorfics i polimorfics.

22/90 (56:45)

2 Polimorfisme

El polimorfisme dit per damunt es que pot prendre diverses formes.

Es una característica dels llenguatges que permet manejar valors de diferents tipus usant una interfície uniforme. Exemple: *La interfície uniofrome en la summa (+) en Java s'utilitza per a diverses coses, la suma de Ca2 , la suma de coma flotant ó la concatenacio de Strings.*

Se pot aplicar tant a funcions com a tipus:

-Una **funcio** pot ser polimorfica pel que fa a un o varis dels seus arguments. *El exemple de la suma mencionat abans.*

-Un **tipus** de dades pot ser polimorfic pel que fa als tipus dels elements que conté. *Per exemple una llista amb elements de tipus arbitrari. $[\text{int}]$ ó $[\text{int} \rightarrow \text{int}]$ ó $[\text{int} \rightarrow \text{char}]$ etc*

Un altre exemple de llista podria ser $[(\text{int}, \text{int}) \rightarrow \text{int}]$ que numericament podria ser una operació matematica. Pases dos nombres i torna un nombre. $5+4 \rightarrow 9$

23/90 (1:05:25)

El Polimorfisme se pot dividir en dos parts, per una banda el Ad-Hoc i per l'altra l'universal. El Ad-Hoc ja l'hem vist en Java. **En les següents diapos se profunditzara en els casos adhoc i universal**

-**AdHoc** es quan se treballa sobre un nombre finit de tipus no relacionats. Per exemple els tipus primigenis, sense utilitzar Tau. Dins d'aquest tenim:

- Sobrecarrega
- Coercio.

-El **universal o vertader** treballa sobre un nombre potencialment infinit de tipus. Es el que hem vist abans de fer tipus utilitzant Tau. Dins d'aquest tenim:

- Paramètric (genericitat)
- Inclusió. (herència)

2.1 Polimorfisme: Sobrecarrega

24/90

La **sobrecarrega** : Existència de diferents funcions amb el mateix nom.

-Els operador aritmètics (+ - * /...) solen estar sobrecarregats. Donats els següents exemples:

```
(+) :: Int -> Int -> Int
(+) :: Float -> Float -> Float
(+) :: Complex -> Complex -> Complex
(+) :: Int -> Float -> Float
```

No es pot sumar (+) enters, que sumar flotants, que sumar complexos... La suma se sobrecarrega amb diverses funcionalitats; sumar cadascun d'aquests tipus diferents.

-L'operador + no pot rebre polítipus.

(+) :: a → a → a

Tenint en compte que la "a" pot ser qualsevol dels infinits tipus, no se pot sumar tot. Un enter i un String no se podrien sumar per exemple.

En Java vam fer servir la sobrecàrrega de la següent manera:

Una classe amb un mateix nom, se li assignen distints atributs i es totalment diferent. Per exemple: Tots se diuen "numbers" però com els paràmetres són diferents, està sobrecarregat.

```
public void numbers (int x, int y) {...}
public void numbers (double x, double y, double z) {...}
public void numbers (String st) {...}
```

Un altre exemple és el mètode "abs" de la llibreria Math de Java. La cosa és que abs està definit per a int, double, float...

2.2 Polimorfisme: Coerció

27/90 (1:14:55)

La **coerció** : Conversió (implícita o explícita) del valor d'un tipus a un altre.

La **implícita** es automàtica.

Si nosaltres definim x com a double tal que el següent, i guardem un enter en x...

```
double x;
```

x = 3; → com 3 és un int, automàticament fa la coerció i passa que la x sigui un int, encara que havíem dit que era double. ¿Per què? Pues perquè estem intentant guardar un int en Ca2 en una variable que està en float (double), per això te que fer la coerció automàticament.

La **explícita** és el que nosaltres coneguem en Java com a casting, forçar un double a que sigui int, per exemple. En la explícita nosaltres decidim que un tipus passi a ser un altre tipus. Com diu en la diapos, una variable primitiva d'un tipus bàsic a un altre.

El casting ens va a servir per a aplicar-lo a objectes que en un principi no tindria que deixar-nos perquè són de diferent tipus però amb la Herència sí que podem.

CLASE 23/09/2020

28/90

Exemples de casting implícit i explícit en Java.

2.3 Polimorfisme: Genericitat o parametric

29/90

Genericitat o Paramètric:

La definició d'una funció o la declaració d'una classe presenta una estructura que és comuna a un nombre potencialment infinit de tipus.

- En Haskell podem definir i usar tipus genèrics i funcions genèriques.
- En Java podem definir i usar classes genèriques i mètodes genèrics.

*En PRG es lo ultim que vam donar dels nodes que tenien una data de un tipus i un next node... La idea es definir una classe i poder pasarli un parametre. El atribut seria el data x i el next sería del tipus Node de la x. La **genericitat en Java** es aço... En lloc de assignar un int, un char etc a un node, assignar una variable. Així es fàcil canviar de tipus sense tindre que canviar de mode.*

30/90

Lo de pasar un parametre com a variable se pot fer amb altres llenguatges, el bloc de dalt es Haskell

31/90

En Java els exemples de la diapos anterior, se definixen com el primer bloc de la diapos.

La K y la V, son paràmetres de la classe. La k y la v, son paràmetres del constructor. En la següent diapos te un exemple de K y V.

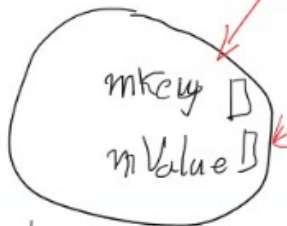
Els metodes statics poden estar en qualsevol classe (per exemple el metode Math) encara que no tinguin variables del seu tipus, els dinàmics nomes poden estar en metodes els quals tinguin variables del seu tipus.

32/90

La K y la V son el Integer y el String, dos tipus qualsevols que se passen com a paràmetres.

El getKey torna el integer mentre que el getValue torna el String. On possa

New Entry <Integer, String>(3, "hola")



elem1.getKey()

System.out.println(elem2.getValue()); tornaria "Programming" que es el String del elem1.

33/90

Polimorfisme: Genericitat

Una classe genèrica és una classe convencional, llevat que dins de la seua declaració utilitza una **variable de tipus** (paràmetre), que serà definit quan siga utilitzat.

Dins d'una classe genèrica es poden utilitzar altres classes genèriques.

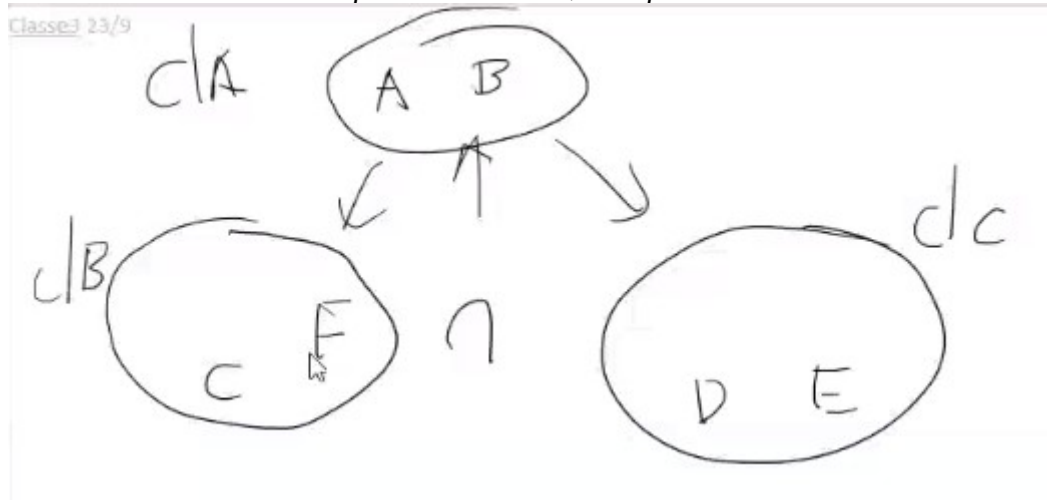
Una classe genèrica pot tenir diversos paràmetres.

2.4 Polimorfisme: Inclusion o Herencia

34/90

Polimorfisme: Inclusió / Herencia

Se definixen diversos atributs en diversos blocs i se fa la intersecció entre tots els blocs per a saber els comuns. Aquest elements comuns seran els heredats. El bloc dA es comu a dB i dC porque A i B estaven en dB i dC i al fer la intersecció, ixen fora.



Inclusió o Herencia: La definició d'una funció treballa sobre tipus que estan relacionats seguint una jerarquia d'inclusió

En l'orientació a objectes l'herencia es el mecanisme mes utilitzat per a permetre la **reutilització i extensibilitat**.

L'herència organitza les classes en una estructura jeràrquica formant **jerarquies de classes**.

35/90

Una classe B heretarà d'una classe A quan volem que B tinga l'estructura i comportaments de la classe A. A més podem:

- Afegir nous atributs a B
- Afegir nous mètodes a B

I depenent del llenguatge podem:

- Redefinir mètodes heretats
- Heretar de diverses classes (en Java solament podem heretar d'una classe)

36/90

Exemple de herencia en Java.

Protected se possa per a que siga visible en la classe en la que estem i totes les classes que hereden, pero no les demés.

37/90

Continuació diapos anterior.

Super s'utilitza per a cridar al metode de la classe de la que hereta. Crida al constructor en aquest cas.

Les subclasses es defineixen usant la paraula clau **extends** i se poden afegir atriburs, m'etodes, i redefinir metodes.

39/90

-En Java, la base de qualsevol jerarquia és la classe Object.

-Si una classe es declara com final, no es pot heretar d'ella

```
final class A {
    ...
}

class B extends A {
    ...
}
```

-Java solament té herència simple

-A una variable de la superclasse se li pot assignar una referència a qualsevol subclasse derivada d'aquesta superclasse, però no al contrari.

La herencia va de dalt cap avall. Un fill hereta del pare pero un pare no pot heretar del fill.

40/90

En Java s'usen qualificadors davant dels atributs i mètodes per a establir què variables d'instància i mètodes dels objectes d'una classe són visibles:

Private: cap membre o atribut private de la superclasse és visible en les subclasses o altres classes.

Si s'usa per a atributs de classe, hauran de definir-se mètodes que accedisquen a aquests atributs

Protected: els membres protected de la superclasse són visibles en la subclasse, però no visibles per a l'exterior.

Public: els membres públics de la superclasse segueixen sent públics en la subclasse.

	Classe	Paquet	Subclasse	Altres
Public	Sí	Sí	Sí	Sí
Private	Sí	No	No	No
Protected	Sí	Sí	Sí	No
Default	Sí	Sí	No	No

42/135

El toString de esta diapos reescriu un metode. El metode toString que s'hereta de Object per defecte, ací l'ha reescrit per a que retorne el que vol.

43/135

El super.toString se posa per que utilitze el toString de la classe pare des de la que hereta, que es Employee (diapos anterior).

Classe Abstracta de Java

44/135

PROPIETATS CLASE ABSTRACTA

Es una classe normal pero se declara abstracta per dos motius...

-No vols que tinga objectes (no permet fer un new), la vols per exemple per a que s'herete des d'ella.

-Té mètodes abstractes, mètodes que no dones implementació. Si la classe te un mètode abstracte, la classe te que ser abstracta si o si.

45/135

Exemple de classe abstracta.

Els metodes abstractes se gasten per a posar obligacions a les classes que hereten. En el exemple tenim dos metodes abstractes, un que calcula el perimetre i un que calcula el area.

Quan diguem que obliga es que... Obliga a les cases que hereten a que si volen ser classes concretes (concret es lo contrari de abstracte) si o si te que tindre els metodes perimetre i area. Perque si no implementa els metodes, aquestos seran abstractes i la classe tindra que ser abstracta si o si (segons diapos 43)

46/135

Esta classe ja es concreta porque utilitza els perimetre i el area que abans obligava a utilitzar. Com se veu ja te un return. Lo guay es obligar a cada figura es que implemente el seu propi perimetre y area, porque no se calcula igual el perimetre de un quadrat que de un cercle ...

47/135

Crear una nova classe que se anomenara triangle, que heretara de shape i definir el perimetre i el area.

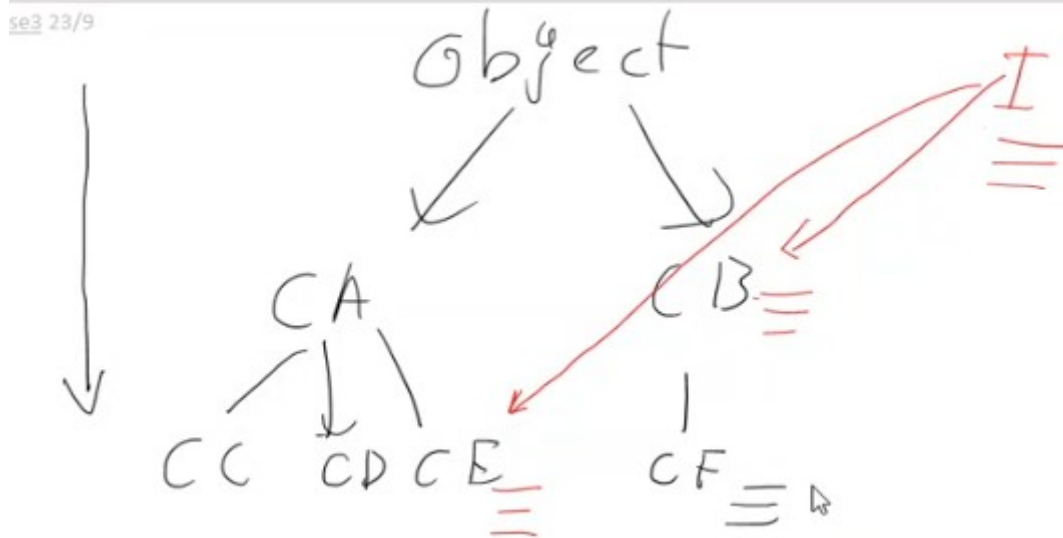
48/135

Interficies:

Les interficies son classes que **només tenen metodes abstractes**, de fet no fa falta mencionar que els metodes son abstractes, pero ho son. En la diapos hi ha un exemple de com se possa la classe, es interessant veure que **no se possa abstract en cap lloc** i que els mètodes no implementen res, no te claus.

A l'hora de implementar una interficie, se possa **implements NomInterficie** en lloc de extends que se possa en la herencia. Dins d'ací si que se possarien les claus per a implementar el mètode. {...}. El implements se possa pues, en resum, per a que una classe tinga que fer servir les obligacions possades en la classe interface.

La herència (abstract) va de dalt cap avall, mentre que les interfícies son transversals.
Exemple imatge:



Amb herència uns mètodes definits en CA, només els podria heretar CC CD CE i mai CF que no està en la mateixa rama; mentre que amb interfícies (I), al ser transversal, els mètodes de I, els pots implementar on vulgues. Ja siga CB o CE, que no estarien connectats per herència.

En resum, se poden implementar obligacions verticalment amb **abstract** i transversalment amb **interface**.

En PRG un array se podia fer amb arrays o amb llistes, Pues una bona interfície seria implementar la utilitat del array o la llista així se diu en la interfície la funcionalitat que se vol que se tinga, i se passa al array i a la llista. “Me dona igual com ho fasses, pero que fassa açò”

-Els mètodes d'una interfície poden ser abstractes, estàtics i default (inclouen una implementació per defecte). La seua visibilitat pot ser public, private i default; NO pot ser protected.

-Els atributs son estàtics i finals (constants). Se donen en la interfície.

-Una classe pot implementar diverses interfícies.

-Les interfícies poden heretar d'altres interfícies (extends)

49/135

No. Si que dona error. No deixa que un node de Shape se li pugui assignar un Node de circle. No s'exté aqueixa compatibilitat. En canvi si que funciona la classe Shape amb la classe Cercle.

Si que deixa dir que shape es un cercle. (shape s = new circle();)

Pero no deixa amb el node: Node<shape> n = new Node<circle>; ← Fallaria. No exté compatibilitat.

Shape Node<Shape>
 ↓ ✗ ↓
 Circle Node<Circle>

Shape s = new Circle(...);
 Node<Shape> n = ~~new Node<Circle>;~~

50/135

Pot una interfície heretar d'una classe?

No. Les interfícies son abstractes i les classes no, ni si quiera d'una classe que fora totalment abstracta. No ens deixa.

Es poden crear instancies de les classes interfície?

No. Es una classe abstracta total.

CLASE 25/09/2020

3 Reflexió

51/135

Que es la reflexió.

La reflexio busca reaccionar davant del que veus. En els Llenguatges de Programació, la **reflexió permet que el programa veja la estructura de la execució i manipularse a si mateix**. *Java no permet canviar la implementació dels mètodes sobre la marxa (pero si que permet consultar), hi han altres llenguatges que si.*

La **reflexio permet** monitoritzar la execucio eexecució del propi programa i inclus modificar-se a si mateix durant la seva propia execució per a adaptar-se dinàmicament a diferents situacions.

Un exemple de reflexió molt simple seria contruir un programa através d'un altre programa. M'explique... En un fitxer escriu un codi que el que fa aqueix xodi es escriure un altre fitxer amb un codi executable.

52/135

Llenguatges amb reflexió

Les instruccions del llenguatge son tractades com a valors d'un tipus específic. Tant instruccions com valors venen a ser equiparables a l'hora de tractarlos. *En Java sería com si un println en lloc de treure per pantalla, ho introduira de nou al programa i cambiara l'estructura; el printl seria un valor del meu programa.*

La reflexio, en llenguatges de programació que no la tenen, se pot veure com a cadenes de caracters.

Els llenguatges amb reflexió poden veure's com a metallenguatges del propi llenguatge. *Aixo vol dir que pot consultar o modificar les instruccions d'un altre llenguatge. Ex. Programes que manipulen programes com a compiladors, analitzadors etc.*

53/135

La reflexió ha d'usar-se amb cautela.

Si se utilitza mal la reflexió pot afectar:

-Al rendiment del sistema ja que sol ser costosa. *Pots caure en un bucle per exemple.*

-La reflexió no es segura. Si el programa pot canviar-se a si mateix durant la execució, pot acabar fent algo que no estiga ben vist... o esperat.

No es complicada d'implementar, especialment en llenguatges funcionals, gracies a la dualitat natural entre dades i programes (aquesta dualitat se l'anomena homoiconicitat).

54/135

La reflexió en Java

En java se pot utilitzar la reflexió important la llibreria “**java.lang.reflect**”, aquesta biblioteca proporciona classes per a representar de forma estructurada informació de les classes, variables, mètodes etc

Se pot inspeccionar classes, interfícies, atributs i mètodes sense coneixe els noms dels mateixos en temps de compilació. Per exemple podem:

- Obtenir i mostrar durant l'execució del programa el nom de totes les instàncies de la classe que s'han creat en temps d'execució.

- Llegir de teclat un String amb el qual poder crear un objecte amb aqueix nom o invocar un mètode amb aqueix nom.

55/135

Exemple d'ús de Reflexió en Java.

En el exemple s'importa la llibreria de la diapos anterior.

Se defineix la classe “MyClass” (que tindrà qualsevol cosa dins).

En un altre lloc creem un objecte del tipus MyClass anomenat myClassObj.

El interrogant son les variables anonimes. Dius que las vols com a qualsevol variable de tipus MyClass o inferior (heretada). Si no vas a fer-la servir (no vas a gastar-la), posses un interrogant, si vas a fer-la servir en altres metodes posteriorment, se possa una variable de tipus (una E per exemple, aquesta E pot ser qualsevol tipus)

La clase “Class” conté definicion de classes. Cada objecte de la classe Class conte definicions de classes y eixos objectes son els que se consulten per a saber quins atributs te una classe y demás cada vegada que ho necessita el interpret.

//Get the fields:

objMyClassInfo.getDeclaredFields(); torna tots els atributs del objecte. Això torna els atributs en un array de camp (Field[])

//Travel the fields

Amb el for, podries escriure els atributs que estan dins del array Field anterior i se poden treure tots els atributs d'una determinada classe

4 Procediments i control de flux

56/135

Procediments i control de flux:

Existeixen alguns conceptes relacionats amb el control de flux dels programes i la definició i cridada a procediment. Dos d'ells són:

-**Pas de paràmetres (diapos 57)**: Quan se fa una crida a un mètode o funció hi ha un canvi de context que pot fer-se de diferents formes.

-**Àmbit de les variables (diapos 73)**: És necessari determinar si un objecte o variable es visible en un moment determinat de l'execució i aquest calcul pot fer-se de forma estàtica o bé de forma dinàmica.

57/135 (20:00)

4.1 Procediments i control de flux. Pas de parametres

Es el que vam veure en primer, consta de dos parts:

-**Declaració de mètodes o funcions** que tenen paràmetres. Tenen un nom, tenen unes variables que seran els paràmetres. Aquests parametres se'ls anomena parametres formals.

-**Crida als mètodes o funcions** passant una serie de parametres . A aquest parametres sels anomena parametres reials./d'entrada

*El que ens interessa es el **pas del paràmetres** dels paràmetres reials als paràmetres formals.*

58/135

Tres TIPUS de pas de paràmetres:

- Pas **per valor** (call by value) ← Java utilitza aquest.
- Pas **per referencia** (call by reference // call by address)
- Pas **per necessitat** (call by need)

Existeixen més modalitats de pas de paràmetres, pero aquestes son les mes freqüents en els llenguatges de programacio.

59/135 a 64/135

Pas per valor

Es el pas de paràmetre que fa servir Java.

Quan cridem al mètode inc(a), se li passa el valor de a. "a" es 10 segons el codi pues passem el valor 10. Suma el 10+10 = 20

Quan acaba, "a" continua valent 10, el que passa es que la variable local del metode inc (la "v"), es la que cambia y ella si que val 20.

65/135 a 70/135

Pas per referencia

Aci passem la direcció de memoria de la variable.

*Quan cridem al mètode inc(a), se li passa la direcció de memoria de "a". Les "v = v+v" que apareix estan referenciant a la mateixa zona de memoria que "a" per tant la "v" es "a". Quan diu que la v=10+10 =20 el que farà es **actualitzar el valor de la "a" a 20** ja que treballem sobre l'adreça de memoria.*

Recorda que Java no te pas per referencia, el exemple anterior es per a entendre'ns d'alguna manera, pero no en te.

Actualització Diapos. Fins ara tot coincidia. Ara te 130 diapos.

71/130

Pas per necessitat

Els parametres reials no s'avaluen, només s'avaluen dins el mètode o funció si els fem servir. S'avaluen si els gastem.

Suposem que en Java funcionaria el pas per necessitat i tenim:

metode (int x, int y) {return x;} I li passem: metode(5,8);

Pues tornaria un 5 (perquè torna la x) i el 8 (que és la y i el mètode no el fa servir) pues ni el tindria en compte.

72/130

Algunes consideracions

En pas per valor, si es passa una expressió, aquesta s'avalua per a copiar el valor resultant (a diferència del pas per necessitat.)

En pas per referència, si es passa una expressió també s'avalua i es passa el valor resultant.

73/130

4.2 Procediments i control de flux. ÀMBIT DE LES VARIABLES

-Una variable és un nom que s'utilitza per a accedir a una posició de memòria.

-No tots els noms (de variables, funcions constants....) estan accessibles durant tota l'execució, encara que existisquen en el programa.

-L'àmbit o abast d'un nom és la porció del codi on aquest nom és visible (el seu valor associat pot ser consultat/modificat). No se pot accedir a variables que estan en un altre bloc.

74/130

Continua els punts de la diapos anterior:

-El moment en el qual es fa l'enllaç (l'associació) és el que es diu temps d'enllaçat.

-**Abast estàtic:** Tenim un compilador que tradueix i ja està. Se defineix en temps de compilació. *Va sobre la marxa. Poden hi haure més d'una variable amb el mateix nom però valors diferents, depenent en el bloc en el que estem. Pots tindre un nom2 que siga Pepe en un bloc i un nom2 que siga Juan en un altre bloc, la variable és nom2. Si intentes consultar una variable en un bloc, el que fa és pujar per l'arbre i trobar el pare amb el nom de la variable que busquem. Imaginant que són blocs, a l'hora de saber el valor d'una variable, se puja al bloc immediatament superior y se miren els valors.*

-**Abast dinàmic:** El porta la pila de registres en la execució. Defineix el àmbit de la variable. *Este es el de java. Se busquen les variables en la pila i se pot accedir des de qualsevol part del codi.* En altres paraules... **La variable arribat a un punt, pren el valor que portava d'abans a mesura que s'ha executat.**

-Tots els llenguatges de programació moderns usen l'abast estàtic.

-Cada llenguatge de programació estableix una manera de determinar l'abast dels elements.

-En Java per exemple s'usen els atributs private, public, protected i el sistema de jerarquia de paquets i classes. *Els paquets son els directoris en el disc dur on estan emmagatzemades les classes (es físic) i les classes son la herència amb el extends (es lògic). Recorda que segons el atribut (private, public protected...) se podia veure d'un lloc o un altre.*

CLASE 30/09/2020

81/130

5 Gestió de memòria

Gestió de memòria

Se refereix als diferents mètodes i operacions que s'encarreguen d'obtenir la **maxima utilitat de la memòria**, organitzant els processos i programes que s'executen en el sistema operatiu de manera tal que s'optimitze l'espai disponible.

Influeix en les decisions de disseny d'un llenguatge.

82/130

Necessitat de gestionar la memòria

- Codi de programa traduït. El codi després de compilar que va executant-se poc a poc
- Informació temporal durant l'avaluació d'expressions i en el pas de paràmetres
- Crides a subprogrames i operacions de tornada
- Buffers per a les operacions d'entrada i eixida.
- Operacions de inserció i destrucció d'estructures de dades en l'execució del programa
- Operacions de inserció i borrat de components en estructures de dades.

83/130

Tipus de gestió de memòria

ESTÀTIC:

Se calcula i s'assigna en temps de compilació. És eficient però incompatible amb recursió o estructures de dades dinàmiques. Com no saps quantes crides recursives vas a fer, no pots assignar la memòria durant la compilació

DINÀMIC:

Se calcula i assigna en temps d'execució. En pila o en un "heap".

No són excluyents una de l'altra. No es gasten dinàmic o estàtic al 100%. De normal se gasten les dos segons necessitats. Per exemple El estàtic per a les variables del codi però dinàmic per a les llistes enllaçades.

84,85 /130

Emmagatzematge estàtic:

Se sol usar amb:

- Variables globals: Per lo general les variables saps del tipus que són i el que ocupen i a partir d'ahí, reservar.
 - Programa compilat: Quan compiles, saps el que ocupa el programa compilat.
 - Variables locals:
 - Constants numèriques y cadenes de caràcters. Com no van a canviar perquè són constants, se poden guardar en l'estàtic
- Taules produïdes pels compiladors u usades per a ioperacions d'ajuda en temps d'execució

Enmagatzematge dinàmic en PILA:

86/130

A mesura que se necessita espai d'enmagatzement, van assignant-se els blocs en la pila un darrere l'altre a mesura que fa falta i sempre se guarda damunt el nou de manera que a l'hora d'alliberar, se fa de manera recursiva, siguent el de dalt de tot el primer en alliberarse i siguent el primer de tots que se va posar, l'últim.

Enmagatzematge dinàmic en HEAP:

87, 88 /130

Tot allò que no és estàtic i no sabem quin tamany va a tindre, va a parar al heap. Per exemple les llistes enllaçades, que no sabem lo grans que seran.

Els blocs no tenen per que ser sempre del mateix tamany.

Se pot accedir a ells de manera directa, no com la pila que només podem accedir al de dalt

La desassignació pot ser:

Explícita: On tens que dir d'alliberar la memòria

Implícita: Tipus Java per exemple. El propi Java ja s'encarrega d'assignar i alliberar a la seua manera com considere. Garbage collector per exemple.

2. PARADIGMES DE PROGRAMACIO

90/130

Factor d'exit d'un llenguatge de programacio

Potencia expressiva: Per a generar codi clar, concis i facil de mantenir.

Facil d'aprendre

Portable i amb garanties per a la seguretat.

Suportat per multiples plataformes

Suport economic.

Facil migracio d'aplicacions escrites en altres llenguatges

Multiples biblioteques per a gran varietat d'aplicacions

Disponibilitat de descarrega de codi obert escrit en el llenguatge.

91/130

Paradigma imperatiu

Imperatiu: La assignacio que gastavem de $x = v$.

Tenim tres coses per a controlar-lo. El punt y coma que indica una ordre darrere altre, el condicional if i els bucles.

El declaratiu cal programar tot el que volem que faça. Linia a linia.

1 Paradigma Imperatiu

92/130

Descriur la programacio com una seqüencia d'instruccions o comandos que canvien l'estat del programa fins a alcançar una sol.lucio.

-La característica principal es la que indica el nom. Hi han imperacions. Ordres. Se li va diguent que cal que fassa. Donar un algorisme pas a pas per que la maquina alcance la sol.lucio.

-Esta en un baix nivell, mes tirant a nivell de hardware. Això implica que siga eficient però va a ferlo difícil de verificar, modificar, te efectes laterals també.

94/130

Un efecte lateral es una manera de organitzar el codi i la manera d'accedir a les variables si son globals o dins d'una funcio que pot fer que, a l'hora de cridar a una funcio, done resultats diferents perquè per la manera en que se defineix, pot retornar un valor que no toca.

96/130

En resumen, las **características principales del paradigma imperativo** son:

- Pone el énfasis en el **cómo** resolver un problema
- Las sentencias de los programas se ejecutan en el orden en el que están escritas y dicho **orden de ejecución** es crucial
- **Asignación destructiva** (el valor asignado a una variable destruye el valor anterior de dicha variable) → efectos laterales que oscurecen el código
- El **control** es responsabilidad del programador
- Más **complejo** de lo que parece (así lo demuestra la complejidad de sus definiciones semánticas o la dificultad de las técnicas asociadas, e.g., de verificación formal)
- **Difícil de paralelizar**

- Los programadores están mejor dispuestos a sacrificar las características avanzadas a cambio de poder obtener mayor velocidad de ejecución

2 Paradigma Declaratiu

97/130

Paradigma declaratiu

La idea es minimitzar el “com” i deixar el “que”. El com deixarliho a la maquina.

En el declaratiu, nosaltres li diguem a la maquina el que volem i el ordinador ja veura com fer-ho.

*La **LÒGICA** es **QUE** volem fer mentre que el **CONTROL** es **COM** ho volem fer. Els declaratius s'enfoquen molt en la logica, de manera que son molt facilis d'entendre aquestos llenguatges llegint-los, de trobar errors, de depurar... En canvi... Enfocar-se tant en la logica fa que descuidem el **CONTROL** i això pot baixar la eficiencia. Els imperatius de abans tenen mes control i menys logica que un declaratiu.*

Hi ha dos tipus de paradigma declaratiu. Funcional i lògic.

2.1 Paradigma Declaratiu: Funcional i lògic

101/130

FUNCIONAL

Permet definir estructures de dades i funcions que manipulen les estructures.

Característiques:

-Polimorfisme.

-Ordre superior: Vol dir que a l'hora de cridar una funció se li pot passar una altra funcio com a parametre d'entrada.

LÒGIC

Característiques:

-Variables lògiques. Que se corresponen amb les variables de la lògica de primer ordre.

-Indeterminisme: Podem cridar a un procediment i que hi hagen diverses possibles execucions (i caldra explorar-les totes per trobar la solucio)

CLASE 02/10/2020

103/130

Característiques de la programació declarativa.

-Expressa **què** es la solucio a un problema.

-L'ordre de les sentencies i expressions no te per que afectar a la semantica del programa.

-Una expressio denota un valor independent del context

-Nivell mes alt de programacio: semantica mes senzilla, acontrol automatica, mes facil de paralelitzar, millor manteniment....

-Eficiencia comparable a Java.

-Costa aprendre un llenguatge declaraitu si ja saps un imperatiu perquè es pensar d'un altra manera.

104 i 105/130

Paradigma Imperatiu VS Paradigma Declaratiu**Imperatiu**

Transcripció d'un algorisme
Ordres a la màquina
Màquina d'estats
Referències a memòria

← **Programa** →← **Instruccions** →← **Model de computació** →← **Variables** →**Declaratiu**

Especificació d'un problema
Fòrmules lògiques
Màquina de inferències
Variables lògiques

3 Paradigma Orientat a Objectes

109/130

Paradigma Orientat a Objectes OO

Basat en la idea d'encapsular en objectes estat i operacions

Objecte: Te estats i operacions.

Concepte de: Classe, instància, subclasses i herència.

Elements fonamentals:

Abstracció: És la idea de classe. Objectes amb característiques similars però a la vegada diferents d'altres objectes.

Encapsulament: Quan dins de l'objecte se poden definir estats i operacions no visibles des de fora (o amb certes limitacions)

Modularitat: Agrupar classes interrelacionades entre elles.

Jerarquia: Mitjançant relacions d'herència per exemple

4 Paradigma Concurrent

111/130

Múltiples processos a la vegada. En el processador només hi ha una cosa executant-se però la idea és que si hi ha algo en espera... pues alliberar-ho de la CPU i donar pas a un altre procés i el que estava en un principi ja acabarà en un altre moment.

El procés de interrupció: Per a un programa en execució i fa que la CPU doni el control a una adreça donada

112/130

Problemes de la concurrència/Interrupció

-**Corrupció de dades**: Dos processos escriuen a la vegada. Pot ser que alguna dada no estigui actualitzada per exemple.

-**Interbloquejos**: Dos processos demanen el recurs que té l'altre procés i entren en un bucle que donem el teu recurs i l'altre te diu que li dones tu el seu perquè tu estàs ocupant-lo.

-**Inanició**: Un procés amb tan poca prioritat que tots els altres processos li passen davant i mai aplega a rebre recursos per executar-se.

-**Indeterminisme**: No saber en quin ordre s'executen les funcions dels programes. Pot ser que algo passi en una CPU una vegada cada 10000 vegades i és molt complicat adonar-se'n i més encara depurar-ho

113/130

Paradigma concurrent: Conceptes propis primeres abstraccions.

Programa concurrent: Processos sequencials asincrons que no fan suposicions sobre les velocitats relatives amb les quals progressen altres processos.

Els **semàfors** donen permís per a sincronitzar. Quan estas accedint a una variable global, el poses en roig per a que ningú accedisca a ella i hi haga corrupció de dades, quan acabes el poses en verd.

114/130

Regio critica: Es definir un bloc de instruccions i donar-li un nom. Això permet que si estem en la RC1 amb les seues instruccions i si el procés 1 esta utilitzant la RC1, si ve un procés 2, no podra gastar la RC1 fins que el primer procés la allibere, això evita interbloquejos (diapo112)

El que se fa es que els procesos reserven la RC per a gastar-la quan esta lliure.

La regio critica evita el problema dels interbloquejos.

Monitor: Es la regio critica però fent servir tipus de dades abstractes. Instruccions que se poden fer en una classe. Poder inserir, borrar... Es com una llista de Regio critica però amb ordres.

Sorgeixen models teòrics per a lluitar contra la concurrencia. Son models.

115/130

Thread i runnable tenen una serie de metodes, estan per internet en la API

116/130

Utilitza MyThread → Thread de la diapos anterior perquè si veiem la documentació de Thread, te el setpriority y start. El runnable no te aqueixos metodes. Açò il·lustra un poc el monitor de la diapo 144. El Mythread te una serie de metodes implementats per fer coses.

117/130

Concurrencia en Java

-Java te concurrencia de forma nativa mitjançant la classe *Thread*, no fan falta biblioteques.

-Un **fil** es com un procés. Els fils depenen de un pare, així que deuen tirar d'herencia.

-Hi han una serie de funcions per a crear, arrancar, avortar, prioritzar... fils.

-La maquina de java organitza els fils però el programador deu evitar que apareguen els problemes de la concurrencia (diapos 112)

-Memoria compartida

118/130

Programacio paralela

Dividir el temps d'execució mitjançant l'ús de diversos processadors, distribuint les dades entre els procesadors i repartint la carrega. *Es com dividir-se la feina.*

119/130

Paralela VS Concurrent

	Paral.lela	Concurrent
Objectiu	Eficiencia: Repartisc la carrega	Interactivitat: Diversos processos simultaneament
Processadors	Nomes es concep amb diverses CPU	Es compatible amb una CPU o varies.
Comunicació	Pas de missatges	Memoria compartida

OBJECTIU: Repartir molta tasca entre molts processadors. VS En le movil tindre diverses aplicacions interactuant entre elles. El compartir per exemple que pots passar algo de Whatsapp a Telegram.

Processadors: Google, molts pc VS un unic pc.

Comunicacio: Pas de missatges VS monitor, semafor

5 Altres Paradigmes: Basat en interacció

120/130

Paradigma basat en interacio

Paradigma tradicional: el programa descriu passos necessaris per a produir una eixida a partir d'una entrada. Es el que vam veure en els imperatius.

Paradigma com interacció: Les entrades se monitoritzen i les eixides son accions dutes a terme dinamicament. *En altres paraules... Programes que solen estar esperant, i quan hi ha una interaccio fan un algo. Una maquina de vending, un sensor d'un cotxe...*

121/130

Programa interactiu: Es una comunitat d'entitats (agents, bases de dades, serveis de xarxa...) que interactuen seguint certes **regles d'interacció**.

Les regles d'interaccion poden estar restringides per interficies, protocols i certes garanties del servei (temps de resposta, confidencialitat de dades, etc)

Caracteristiques del model de programacio interactiva:

- La prog conduida per esdeveniments: Es la important. A partir de la diapos 122 s'explica.
- Reactius/Encastrats (Empotrados): Ve a ser el software que pot hi haure en un cotxe, una nevera...
- Client / Servidor: Un a part genera dades i l'altra les consumeix.
- Programari basat en agents

S'utilitzen en: Aplicacions distribuïdes, disseny de GUI, programacio web, disseny incremental de programes (es refinen parts d'un programa mentre esta en execucio.)

122/130

Programació per esdeveniments:

El flux del programa està determinat per esdeveniments. Un esdeveniment com polsar el boto, manejar el ratolí, fer un dobleclick...

L'arquitectura típica sol ser d'un bucle principal que està executant-se continuament i té dos seccions independents aquest bucle:

- 1: Detecció o selecció d'esdeveniment. El bucle detecta l'event. 'Pulsar sobre minimizar'
- 2: Maneig dels esdeveniments. El bucle decideix que fer amb l'event una vegada passa. 'Minimizar pantalla.'

123/130

A l'hora de realitzar programació per esdeveniments, dir que se pot gastar qualsevol llenguatge de programació sempre que aquest permeti els dos punts de la diapos anterior: Detectar senyals, interrupcions de la CPU o aplicacions GUI (clicks del ratolí pEx); i a més gestionar una cola d'events per a reaccionar davant d'ells en l'ordre en que s'han rebut.

Desarrollar programes per esdeveniments se simplifica molt utilitzant **patrons de disseny**. *Un patró de disseny és una plantilla en la que, si tu saps que vols fer X cosa, busques una plantilla que s'adeqüi i a partir d'ella ja fas. Simplifica molt.* Un molt utilitzat és el patró **event-handler**.

124/130

Patró event-handler.

Dispatcher: Està sempre escoltant events. És un bucle. El dispatcher distribueix els events als manejadors handler.

125/130

Exemple de dispatcher.

126/130

Paradigma basat en interaccions: Consideracions finals.

-És un paradigma transversal, això vol dir que podríem utilitzar altres paradigmes de programació junt al de esdeveniments. Per exemple podem tenir un Orientat a Objectes i per esdeveniment, o un declaratiu lògic i per esdeveniments etc.

AVANTATGE: Simplifica la tasca del programador en proporcionar una implementació per defecte per al bucle principal i la gestió de la cua d'esdeveniments.

DESavantatges:

- Promou un model d'interacció excessivament simple.
- És difícil d'estendre
- És propens a error ja que dificulta la gestió de recursos compartits.

127/130

Altres paradigmes emergents:

-Bio-Computació: Inspirar-se en la biologia a l'hora d'escollir com se fa un llenguatge de programació. Amb això per exemple i formules logiques se pot aconseguir «programar» molècules d'una manera que quan te prens una pastilla, si se troben amb X situacio alliberen el medicament o no, així se pot atacar a les zones afectades, per exemple.

-Computació quàntica: Se substitueix el hw tradicional per portes quantiques, procesadors per procesadors quantics (els quals tenen molts errors perquè estan en desenvolupament...) Per ara, costa pensar que se puga. Programar aquests ordinadors serà molt complicat.