Multiplatform szoftverfejlesztés

Bevezetés

Tárgy adminisztráció

https://portal.vik.bme.hu/kepzes/targyak/VIAUMA04/

- Előadó: Rajacsics Tamás (raja@aut.bme.hu)
- Előadás/gyakorlat péntek és páros csütörtök
  - Csütörtök második héten lesz
  - Gyakorlatot lehet követni saját géppel
    - VS2019
    - Visual Studio Code
    - Qt Creator

Tárgy adminisztráció

- ZH
  - Minimum elégséges az aláíráshoz
  - Teljes C++ anyag (JS nincs benne)
  - Mindig az előadás idejében
  - Amikor odaérünk az anyaggal: kb. április vége
  - PótZH: ZH után 2 héttel
  - PótPótZH: pótlási héten
    - Ez egy elővizsga is egyben, akinek megvan az aláírás
- Vizsga minimum elégséges
- Végső jegy: ZH * 0,4 + Vizsga * 0,6

Tartalom

- Multiplatform célok
- Programozási környezetek
- Megoldások
  - HTML, CSS, JS
  - C++
  - Java
  - .NET
  - Xamarin
  - MonoGame
  - Unity, Unreal, CryEngine

Multiplatform célok

- Emberi erőforrás kezelés
  - Nem kell minden célplatformhoz fejlesztő
  - Könnyebb cserélni/pótolni az embereket
  - A teljes gárda jobban egyben tartható
    - Ez lehet közösségi cél is
  - Egy dologhoz érteni jobban lehet, a cég/csapat erősebb kompetenciát tud építeni

- Célközönség elérése
  - Olyan platformokra is ki lehet adni az appot, ami nincs benne a kiírásban (pl. Windows Phone…)

Multiplatform célok

- Költségek csökkentése
  - Minden multiplatform technológia olcsóbban hoz ki 2 platformot, mintha megírnánk mindkettőre
- Gyorsabb fejlesztés
- Támogatás
  - Hibát csak egyszer kell javítani, és mindegyikben megjavul
  - A platformok változása esetén a technológia változik alattunk
    - Esélyes, hogy nem kell hozzányúlni a kódhoz új platform verzió esetén
  - Új verzió esetén könnyebb zökkenőmentesen frissíteni

Multiplatform célok

- Újrafelhasználható kód
  - Idő és pénz
- Egységes dizájn
  - Ha nem cél, akkor meg lehet csinálni többfélére is
- Korszerű
  - Egyre nagyobb a multiplatform nyelvek/technológiák aránya

Multiplatform problémák

- Funkcionalitás hiánya
  - Nem tud mindenhez hozzáférni, amihez natív társa igen
    - Ezen valamikor lehet segíteni, de körülményes
- Technológia gyenge pontjai és hibái
  - Nem tudunk segíteni rajta általában

Mire jók a programozási környezetek?

- Hardverközeli programozás problémás
  - CPU, gépikód, assembly
  - I/O portok, eszközök, időzítés
- Cél
  - Absztrakciós szint növelése
    - Magasabb szintű fogalmak bevezetése (nyelvi és könyvtárbeli)
    - Szöveg, grid, rekord, reláció, lambda, stb.
  - Produktivitás növelése
    - Nem azonos az absztrakciós szinttel

Nyelvi konstrukciók

- Imperatív, vagy deklaratív megközelítés
- Metódusok / függvények
  - Változó paraméter lista
  - Névtelen, beágyazott
- Lokális változók (capture)
- Modulok – láthatóság
- OO koncepciók
  - Öröklés, polimorfizmus, virtuális metódusok
- Rekurzió, lista kezelés, stb.

Nyelvi konstrukciók implementációja

- Adatok ábrázolása a memóriában
  - Egész szám, még egy nyelven belül sem mindig azonos a jelentés (16 bit / 32 bit / 64 bit)
  - Szöveg (Karakter kódolás? Hossz/végjel?)
  - Lebegő pontos számok
- Típusok
  - Szerződés/specifikáció a különböző részek között
- OO koncepciók ábrázolása a memóriában
  - Virtuális függvénytábla
  - Futásidejű típus információ (RTTI, reflection, typeof)

Futásidejű konstrukciók

- „this” pointer – JavaScript vs. C++/C#
- Ki takarítja a vermet?
  - Pascal vagy C hívási konvenció
- Kivételkezelés
- Memória kezelés
  - Manuális vagy sem? Pinning?
- Debuggolás, Edit&Continue, …
  - Hot/Live reload nem Edit&Continue
  - Hot Module Replacement az

Megvalósítások 1

- A nyelvi szint megvalósítása „összenő” az infrastruktúrával a gépi kódot generáló fordítók esetében (C, C++, pascal, prolog, stb.)
  - Kód generálásakor meghatározza, hogy milyen hívási konvencióval, szám ábrázolással stb. dolgozik
- A köztes nyelvet használó megoldásoknál (Java, .NET, VB stb.) a futtató környezet határozza meg a konvenciókat
  - Interpretált
  - JIT fordító

Megvalósítások 2

- A megvalósítások általában erősen kötődnek egy adott operációs rendszerhez
  - Tipikusan C API hívások
  - Bootstrapping más
  - Paraméterezés más
  - Koncepciók különbözhetnek (thread, fork)

Nincs együttműködés

- A programozási környezetek zárt rendszerek
- Se a környezetek se a programnyelvek nincsenek felkészítve az együttműködésre
  - Általában egy C jellegű külső hívási megoldást támogatnak (pl. Java)

- Miért?
  - Elvi nehézségek (nyelvi koncepciók)
  - Technikai kihívások (pinning)
  - Motiváció hiánya

Multiplatform (Cross-platform)

- Nyelv
- Platform támogatás
- Eszközök
- Dokumentáció, Tutorial, Fórum, stb.
- Alapkönyvtárak (STL, BCL, stb.)
- UI
- Egyéb könyvtárak

Java – 1995

- Modern koncepciók
  - Automatikus memória kezelés (lisp, ‘60)
  - Egyszerű OO (egyszeres öröklés, COM)
  - C jellegű tömör szintaktika
  - Nincs szabvány
- „Írd meg egyszer, futtasd mindenhol.”
  - Szép cél és szinte sikerült
- A nyelv és a futtató környezet összenőtt
  - Kb. 300 nyelv készült hozzá kutatási jelleggel
  - Nehéz más nyelvi környezetekkel összeilleszteni

Miért született meg a .NET?

- A ’90-es évek vége: VB és C++ nem elég
  - Nincs produktív, modern nyelvi megoldás
- J++: MS saját Java implementációja
  - Új konstrukciók (COM): például események, tulajdonságok
  - Nyílt levelek, pereskedés – zsákutca
- Internet térnyerése
  - Minden mindennel össze lesz kötve
  - Nem asztali számítógép eszközök megjelenése

.NET – 2002

- Modern futtatókörnyezet
  - Automatikus szemétgyűjtés, biztonság stb.
- Új, korszerű, produktív nyelv
  - C#: kompromisszumok nélkül
- Több nyelv támogatása
  - Kompatibilitás: VB, C++, COM, …
- Platformfüggetlenség
  - Ekkor még csak cél

Szkript nyelvek

- Ha az interpreter elérhető más platformon, akkor általában a szkriptek is futtathatóak ott
  - Egy értelmező elkészítése egy új platformra kisebb költségű, mint egy fordító
- JavaScript, ActionScript, python, perl, tcl, php, ruby, …
- Gyors fejlesztés, kis kódbázis, prototipizálás
- Kiterjesztés: játékok (WebGL), tervezőprogramok stb.

Xamarin

- .NET, Mono (legújabb verzió hamar bekerül)
  - Saját CLR, JIT compiler
- Platform: Android, iOS, UWP
- Közös UI: Xamarin.Forms
- Ingyenes (2016 március 31-től)
- iOS fejlesztéshez kell Mac (Compile, Deploy, Debug)
- Szolgáltatások: Test Cloud, Crash Monitor

MonoGame

- XNA 4 API
  - Microsoft XNA kifutott
    - nincs már rá támogatás
- Az egyik legmagasabb szintű SDK
- Nem játékmotor, azt meg kell írni benne

MonoGame

- Platform támogatás
  - iOS, Mac OS, PlayStation Mobile, Android, Linux, BSD, Ouya, Windows Phone 8, Windows Store, Windows Desktop
  - iOS, Android: Xamarin
  - Windows, Windows Phone, Windows Store: SharpDX (ingyenes DirectX C# API)
  - OpenGL és OpenGL ES máshol (OpenTK)

Unity 3D

- Játékmotor (nem sima keretrendszer)
- Scriptelhető (programozható)
  - C#
  - .NET 4.7 (C# 6)
  - Játéklogika és sok minden más
- Plugin
  - C++ (vagy bármi, ami natív kódot hoz létre)
  - Scriptből hívható
  - Renderbe is bele lehet avatkozni
  - Platformonként meg kell írni, 32/64 bitre is

Unity 3D

- Ingyenes
  - Pro: $125/hónap
- Platform támogatás
  - Windows Phone, BlackBerry, Android, iOS, PlayStation, Web, Linux, Mac OS, Windows, Windows Store, Xbox, WiiU, PS Vita, PS Mobile
- Magas szintű támogatás

CryEngine/Unreal Engine

- Játékmotorok
- Nem ugyanaz megy mobilon
- CryEngine
  - Platform támogatás: Windows, Linux, PlayStation, Xbox, WiiU, iOS, Android
  - Ár: Ingyenes
- Unreal
  - Platform támogatás: Windows, Linux, PlayStation, Xbox, iOS, Android, Oculus Rift, OS X, HTML 5
  - Ár: 5% a teljes bevételből $1M felett

Kérdések?

Multiplatform szoftverfejlesztés

C++ ismétlés, új nyelvi elemek

Mikor használunk C++-t?

- Régi kódbázis (legacy kód)
- Bizonyos célplatformokon csak ez van
  - Tipikusan beágyazott rendszerek
- Összetett GUI alkalmazások
  - Amikor a vezérlők száma több ezer, azt kevés keretrendszer bírja
  - Törekszünk az egyszerű UI-ra, ha lehet
- CAD/multimédia alkalmazások
- Más nyelvek értelmezője/fordítója

Mikor használunk C++-t?

- Nagy adatmennyiség kezelése
  - Adatbázis motor, folyamatirányítás, videó enkódolás/dekódolás
- Operációs rendszerek
- Algoritmusok
- Játékok
  - Sok játék motor C#/egyéb felületet ad

Gyors, de mégis mennyire?

```
int x = 0;
for (int i = 0; i < 1000000; i++)
{
  int y = i % 1000 == 0 ? 1 : 0;
  for (int j = 0; j < i / 10; j++)
    x += y;
}
```

- C++: 1 ms (x += y * y => 2s)
- C#: 13s
- JS (Chrome): 51s
- JS (régi Edge): 41s
- JS (Firefox): 59s

Sebesség teszt

- Ha az adattípus csak double lehet, akkor JS gyors (mint C++)
  - Vagy használhatunk WebAssembly-t
- C++ akkor tud gyorsabb lenni, ha
  - Adatformátum kedvező
    - int: 32 bitre fordítva
    - long long: 64 bitre fordítva
  - A probléma megoldható tömör adattal
    - Uniók, bitműveletek, változó igazítás
  - Speciális utasításkészlet
    - SSE2, AVX2
  - Párhuzamos algoritmus, GPU

C++ ismétlés, alapok

Történelem

- Bjarne Stroustrup
- 1979: C with Classes
  - Cél: mechanizmusok kialakítása bonyolultság kezelésére nagy alkalmazások fejlesztésében
  - Alap: C, gyors, portable, széleskörben elterjedt
  - Elképzelés: C-t kiegészíteni osztályokkal, leszármazással
- 1983: C++, virtuális függvények, overloading, referencia
- 1989: C++ 2.0, static, többszörös öröklés, absztrakt osztályok, template-ek – ez terjedt el széles körben, majd szabvány 1998
- 2011: C++11 (eredeti nevén C++0x), számos új feature
- 2014: főleg javítások, apróságok
- 2017: fő verzió
- 2020: fő verzió, rengeteg nyelvi változás ismét

C++

- Definíció: Vékony absztrakciós rétegű programozási nyelv
  - Ma már egyre kevésbé igaz ez
- Fontosabb jelzők
  - Általános célú
  - Közvetlen hardver leképezés
  - Nulla overhead
  - Osztályok
  - Öröklés
  - Paraméteres típusok (template)

Függvények

Deklaráció (több is lehet egy függvényhez):

```
int add(int a, int b);
void print(std::string msg);
int add(int foo, int bar);
int add(int, int);
```

Definíció (Pontosan egynek kell lennie):

```
int add(int a, int b) {
    return a + b;
}
void print(std::string msg) {
    cout << msg;
}
```

Osztályok

```
class Stack {
public:
    void Push(int value);
    int Pop() {
        return stack[top];
    }
private:
    int top;
    int stack[10];
};

void Stack::Push(int
value) {
    stack[top++] = value;
}
```

- public látható kívülről, private nem
- class és struct kulcsszó majdnem ugyanaz, struct alapból public
- definíció lehet külön a deklarációtól (tipikusan deklaráció headerben, definíció a forrásfájlban)

Objektumok létrehozása

```
void foo()
{
    Stack s1; // Automatikus objektum
    s1.Push(10);
    Stack* s2 = new Stack(); // Dinamikus objektum
    s2->Push(10);
    delete s2; // Dinamikus objektum felszabadítása
               // ha nem volt exception
    // s1 automatikusan felszabadul,
    // amikor a scope-ból kilépünk
}
```

Objektumok átadása

```
void useStack1(Stack s) { s.Push(5); }
void useStack2(Stack* s) { s->Push(5); }
void useStack3(Stack& s) { s.Push(5); }

void foo2() {
    Stack s1;
    Stack s2 = s1; // Másolás
    Stack* s3 = &s2; // Address-of operátor, memóriacím
lekérése

    useStack1(s1); // Másolás (átadás érték szerint)
    useStack2(&s1); // Nincs másolás (átadás cím szerint)
    useStack3(s1); // Nincs másolás (átadás cím szerint)
}
```

Pointerek és referenciák

- Ajánlás: mindig használjunk referenciát. Pointert csak akkor, ha muszáj. Referencia biztonságosabb, könnyebben olvasható

	Pointer	Referencia
típus	`Stack* s`	`Stack& s`
member elérése	`s->Push(5)`	`s.Push(5)`
default érték	`nullptr`	nincs! `Stack& s; // ERROR, kötelező értéket adni`
átadás	Memóriacím	memóriacím
aritmetika	`++, --, +, -`	nincs
módosítás	Tetszőleges	nem lehet módosítani
többszörös indirekció	igen, pl.: `Stack** s`	nincs

Leszármazás

```
class Base {
private:
    void privFoo() { /* ... */ }
protected:
    void protFoo() { /* ... */ }
public:
    virtual void Foo();
};

class Derived : Base {
    void Foo() // Felülimplementálja Base::Foo-t.
    {
        privFoo(); // ERROR
        protFoo(); // OK
    }
};
```

Többszörös öröklés

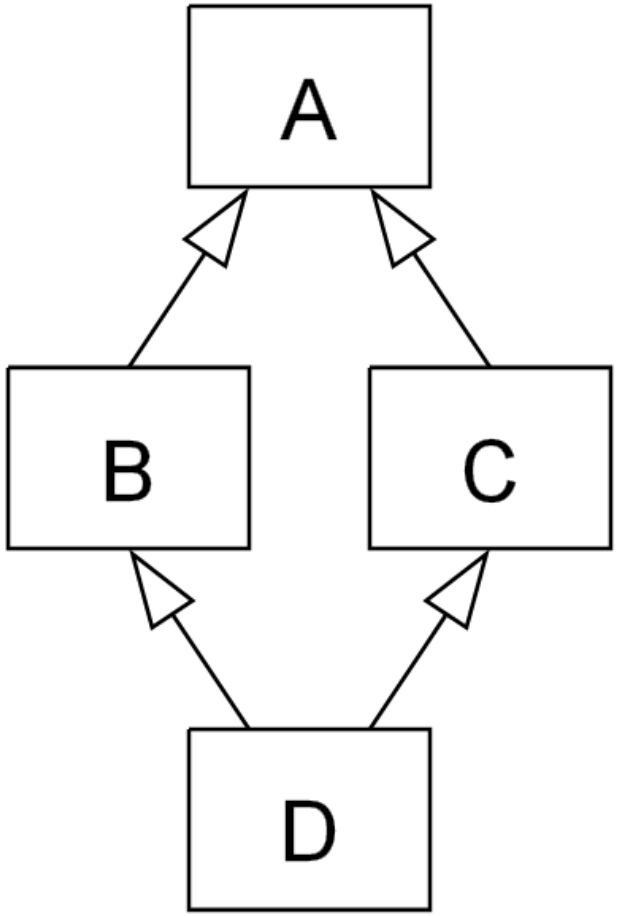
```
struct CommonBase {
    int foo();
};

struct BaseA : CommonBase {
    int foo() { return 1; }
};

struct BaseB : CommonBase {
    int foo() { return 2; }
};

struct Derived : BaseA, BaseB {
};
```

```
int main()
{
    Derived d;
    cout << d.foo(); // ERROR,
ambigous access.
    // Nem egyértelmű, hogy
melyik örökölt
    // függvényt szeretnénk hívni

    cout << d.BaseA::foo(); // 1
    cout << d.BaseB::foo(); // 2
}
```

Virtuális öröklés

- Több öröklési úton keresztül örököljük ugyanazt a tagot
- Hányszor tároljuk az ős adattagjait?

```
struct A {
    int foo();
};

struct B : public virtual A {
    int foo() { return 1; }
};
```



Többszörös öröklés, interfészek

- Interfészek: „pure virtual” osztályok
- pure virtual: nincs adattagja és implementációja
- pure virtual osztályokból nyugodtan örökölhetünk többszörösen
- Egy szerződést írnak le, amit a leszármazottnak meg kell valósítania

Többszörös öröklés, interfészek

```
class Drawable {
    virtual void Draw() = 0; // Pure virtual function
};
class SupportsInteraction {
    virtual void Click() = 0; // Pure virtual function
};
class Button : Drawable, SupportsInteraction {
    void Draw();
    void Click();
};
class Rectangle : Drawable {
    void Draw();
};
```

Generikus programozás

- Cél: hasznos algoritmusok és adatstruktúrák általánosítása paraméterezéssel
- Típusok más típusokkal

```
vector<int> intVect;
vector<Shape> shapeVect;
```

- Algoritmusok más algoritmusokkal

```
bool compareShapes(const Shape& s1, const Shape& s2)
{
   return s1.X > s2.X;
}
sort(shapeVect, compareShapes);
```

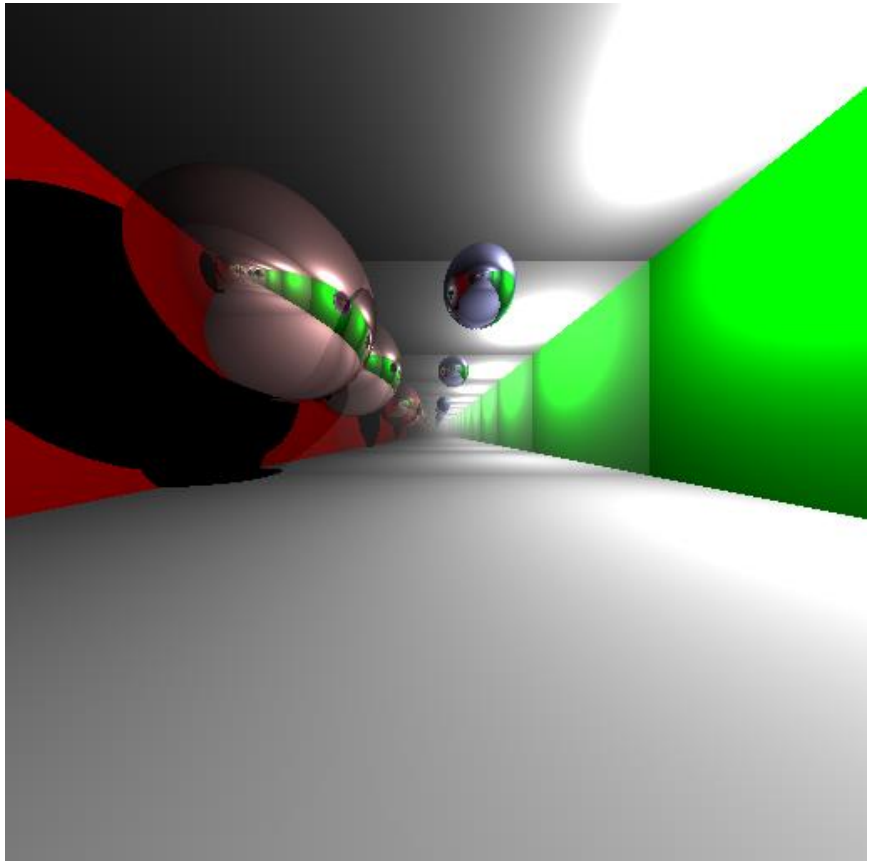
(Template metaprogramozás)

- Logika implementálása template-ekkel
- Kiértékelés **fordítási** időben

```
template <int n>
struct factorial {
    enum { value = n * factorial<n - 1>::value };
};

template <>
struct factorial<0> {
    enum { value = 1 };
};
```

metatrace

- Ray tracer fordítási időben



```
private:
        typedef typename whitted_mirror<
                intersection,
                ray,
                whitted_style::raytrace,
                depth_max
        >::color mirror_color;

        typedef typename intersection::material::reflection reflection;
        typedef typename intersection::material::color      surface_color;
public:
        typedef ift<
                intersection::does_intersect,
                color::add_rgbf<
                        color::mul_rgbf<
                                diffuse_color,
                                scalar::sub<scalar::c1,reflection>
                        >,
                        color::mul_rgbf<
                                color::filter_rgbf<
                                        surface_color,
                                        mirror_color
                                >,
                                reflection
                        >
                >,
                background_color
        > color;
```

C++ újdonságok

C++11, 14, 17, 20

Szabványosítás

- C++98: A már elterjedt programnyelv szabványosítása
- C++03: Hibajavítások
- C++11: Eredetileg C++0x volt a neve, mert évekig azt hitték, hogy 2010 előtt kijön
- C++14: Hibajavítások
- C++17: Fő cél az osztálykönyvtárak bővítése
- C++20: Rengeteg újdonság (nyelvi is)

C++ 11, a megkésett óriás

- Az első jelentős újítás 1989 (1998) óta.
- Számos új nyelvi és STL feature
- Sokat implementáltak már korábban a fordítók, de a szabványtól kezdve lehet megbízhatóan használni
- Összességében: koherensebb, könnyebben használható
- Jobb teljesítmény, biztosabb memóriakezelés
- Több feature, egyszerűbb használat

auto

- C++-ban sok esetben hosszú típusnevek

```
std::map<int, my_namespace::Gadget> gadgets;
std::map<int, my_namespace::Gadget>::const_iterator
        it = gadgets.begin();
```

- Ehelyett

```
auto it = gadgets.begin(); // Honnan tudja a típust?
```

Kollekción iterálás

```
for(auto it = shapes.begin(); it != shapes.end(); it++)
        it->Foo();

for(auto& shape : shapes)
        shape.Foo();
```

auto iterációs paraméter

```
vector<string> v; // Melyik a gyorsabb?
for (const auto& s1 : v) { }
for (const string& s2 : v) { }
// Egyforma.

map<string, int> m; // Melyik a gyorsabb?
for (const auto& p1 : m) { }
for (const pair<string, int>& p2 : m) { }
// const auto& a gyorsabb, mert m értékeinek típusa pair<const string, int>.
// Mivel a típusuk különbözik, ezért egy másolat jön létre.
```

enum problémák

- Az enumerátorok láthatók az egész scope-ban

```
// Fordítási hiba
enum Animals { Bear, Cat, Chicken };
enum Birds { Eagle, Duck, Chicken };
```

- Automatikus int-konverzió

```
int i = Chicken;
bool b = Eagle && Duck; // wat?
```

- A mögötte lévő típus nem szabályozható

enum class

```
// Működik
enum class Birds { Eagle, Duck, Chicken };
enum class Animals : unsigned char { Bear, Cat, Chicken };
auto i = Chicken; // Fordítási hiba
int i = Birds::Chicken; // Fordítási hiba
struct MyStruct
{
	Birds BirdsField; // 4 byte
	Animals AnimalsField; // 1 byte
};
```

Üres pointer, C++ 98

```
#define NULL 0
```

- Két jelentés: egész szám és pointer konstans

```
void f(char const *ptr);
void f(int v);
int n = NULL; // Lefordul.
f(NULL); // A második overload hívódik meg.
```

- Könnyen okozhat hibákat

nullptr

- nullptr: pointer literal, a típusa std::nullptr_t, az értéke 0.
- NULL konvertálható std::nullptr_t típusra
- nullptr nem kasztolható implicit módon integerré

```
void f(char const *ptr);
void f(int v);
int n = nullptr; // Fordítási hiba.
f(nullptr); // Az első overload hívódik meg.
```

override leszármazott osztályban

```
struct B {
	virtual void f();
	virtual void h(char);
	virtual void g() const;
	void k(); // Nem virtuális
};
struct D : B {
	void f(); // B::f()-et implementálja felül
	virtual void h(char); // B::h()-t implementálja felül
	void g(); // Nem implementál felül (rossz típus)
	void k(); // Nem implementál felül (B::k() nem virtuális)
};
```

Függvény-felüldefiniálást nem kötelező kulcsszóval jelölni

override kulcsszó

```
struct B {
	virtual void f();
	virtual void h(char);
	virtual void g() const;
	void k(); // Nem virtuális
};
struct D : B
{
	void f() override; // OK: B::f()-et implementálja felül
	void g() override; // Hiba: rossz típus
	virtual void h(char) override; // B::h()-t implementálja felül
	void k() override; // Hiba: B::k() nem virtuális
};
```

Függvény törlése

- Előfordul, hogy egy osztály másolását szeretnénk tiltani kívülről

```
class nonCopyable {
public:
    nonCopyable() { }
private:
    nonCopyable(const nonCopyable& other);
};
nonCopyable instance;
nonCopyable copy = instance; // ERROR
// C2248: 'nonCopyable::nonCopyable' : cannot access private member declared in class 'nonCopyable'
```

Függvény törlése

- C++11-ben explicit „törölhető” egy függvény

```
class nonCopyable
{
public:
    nonCopyable() { }
    nonCopyable(const nonCopyable& other) = delete;
};
nonCopyable instance;
nonCopyable copy = instance; // ERROR
// C2280: 'nonCopyable::nonCopyable(const nonCopyable &)' : attempting to
reference a deleted function
```

Függvény törlése

- Nem kívánt konverziók is letilthatók

```
struct Z
{
    // ...
    Z(long long);     // long longgal inicializálható
    Z(long) = delete; // de kisebb egésszel nem
};
```

Alapértelmezett implementáció

- Ha elveszítjük az alapértelmezett konstruktor implementációt
  - Mert például létrehozunk egy paraméteres konstruktort
- Hasonlító operátorhoz is C++20-tól

```
class A
{
    A(int);
    A() = default;
};
```

long long

- Eddig fordító specifikus megoldások
  - __int64
- Fontos a támogatása
  - 64 bites művelet atomi a megfelelő processzoron

```
long long a; // 64 bit
```

String literal

- Korábban volt
  - “hello” // const char* (ne használd)
  - L”hello” // const wchar_t*, 16/32 bit/char, Unicode 16/32, (ne használd)
- C++11
  - u8”hello” // UTF8, const char*
  - u”hello” // UTF16 , const char16_t* (ne használd)
  - U”hello” // UTF32, const char32_t*
  - R"(\w\\\w)"; // kombinálható mindegyikkel, raw string
- C++17
  - u8’h’ // UTF8 karakter, char (ne használd)
- Karaktereket általában ne használjunk
  - Például ß vs. SS, vagy i vs. İ

User defined literal

```
101010111000101b // binary
1.2i             // imaginary

constexpr complex<double> operator "" i(double d)
{
    return {0,d};
}
```

Thread local storage

```
thread_local int a;
```

- Minden szálban más az értéke
- Mire jó?
  - Globális változók szálbiztos tárolása
  - Temp változó, nem kell újrafoglalni, itt talán lehetne _alloca

Structured Binding (C++17)

```
auto [a, b] = f();
```

- Ha f függvény olyat ad vissza, amit szét lehet szedni több változóba
  - Tömb
  - Objektum mezőkkel
  - std::Tuple

Egyéb C++17

- Parallel STL:
  - `sort(par, vec.begin(), vec.end());`
- Hex floating point numbers
  - 0x1.2p3 // 9
- Fordításidejű if
  - if constexpr (true) { } // az else ág bele se fordul
- Inicializálás if-ben
  - if (auto p = f(); !p.second) { } // 2 tagja van az if-nek

Elágazás optimalizálás (C++20)

- Melyik ág valószínű

```
if (n > 1) [[likely]]
        return 1;
    else [[unlikely]]
        return 2;
```

Fordításidejű kiértékelés (C++20)

- C++17-hez képest bővült a kör, ahol lehet használni és van consteval kulcsszó

```
consteval int sqr(int n) {
  return n*n;
}
constexpr int r = sqr(100);  // OK

int x = 100;
int r2 = sqr(x);  // Error: Call does not produce a constant
```

C++ 20 range és view

- begin() + end() helyett
- Algoritmusok, amik elfogadtak iterátort, most elfogadnak range-et és view-t
- Hasonló a C# IEnumerable interfészhez
- Tipikusan input range minden – csak olvasható
  - Van output range, ami írható. Például vector input és output is
- | operátorral lehet fűzni őket

```
std::vector<int> ints{0,1,2,3,4,5};
for (int i : ints | std::views::filter([](int i){ return i >2; }))
    std::cout << i;
```

C++ 20 range és view

- Nem feltétlen csak iterálni lehet rajta – ezt mindegyik tudja
  - Címezni is lehet valamelyiket []

	forward_list	list	deque	array	vector
std::ranges::input_range	☑	☑	☑	☑	☑
std::ranges::forward_range	☑	☑	☑	☑	☑
std::ranges::bidirectional_range		☑	☑	☑	☑
std::ranges::random_access_range			☑	☑	☑
std::ranges::contiguous_range				☑	☑

C++ 20 range és view

- view egy lusta kiértékelésű algoritmus, nem kollekció
  - Csak akkor fut le, amikor iterálunk rajta
- Hasznos kollekciók transzformációjára
  - std::views::*
    - filter – csak azokat adja vissza, amiket a predicate igazra értékel
    - reverse – megfordít
    - drop – kihagy X elemet
    - take – első X elem
    - take_while – addig veszi az elemeket, amíg a predicate igaz
    - …

C++20

- Modulok
  - Mint #include, de a fordító izolálja a fordítást, így újra felhasználható a fordítás eredménye – gyors
- Coroutine: co_await a szokásos async-await minta
- <=> operátor sorrendezéshez
  - Ha alapértelmezett implementációt használunk (=default), akkor legenerálja az összes hasonlító operátort
- template kényszerek: lényegesen többet tud, mint pl. C#
- void f(auto x); // template<class T> void f(T x)

Fordítók

- Gyakorlatilag bármilyen platform
- Minden elterjedt fordító C++17 konform
  - GCC: teljes C++17 és 99% C++20
  - Visual C++: teljes C++17 és 99% C++20
  - Clang: közel teljes C++17 és 90% C++20

- Még sok másik: http://www.stroustrup.com/compilers.html

Fejlesztőeszközök

- Platform- és fordítófüggő
- Windows: Visual Studio
  - Esetleg Visual Assist extension
- Bárhol: Eclipse, Netbeans, Qt
- Csak szerkesztők: vim, emacs, sublime, VSCode, stb.

Kérdések?

Multiplatform szoftverfejlesztés

Új C++ nyelvi funkciók

Automatikus memóriakezelés

RAII – Resource Acquisition Is Initialization

Automatikus memóriakezelés

- Pointerek használata veszélyes
- Könnyű memory leaket okozni

```
void use_gadget(int x)
{
  Gadget* gadget = new Gadget();
  ...
  if(x > 100) throw std::runtime_error("error!");
  ...
  delete gadget;
}
```

Automatikus memóriakezelés

- Scoped variable
- Egyszerű, biztonságos
- Modern C++ *stílus*

```
void use_gadget(int x)
{
    Gadget gadget(x);
    ...
    if(x > 100) throw std::runtime_error("error!");
    ...
}
```

Automatikus memóriakezelés

- Smart pointer: automatikus felszabadítás
- Referencia számlált
- Shared ownership

```
void use_gadget(int x)
{
    shared_ptr<Gadget> gadget =
        make_shared<Gadget>(x);
    ...
    if(x > 100) throw runtime_error("error!");
    ...
}
```

Automatikus memóriakezelés

- unique_ptr – csak egy példány lehet
  - = operátor elpusztítja a régit
- weak_ptr – nem tartja életben az objektumot
  - Körkörös hivatkozásoknál
  - Például fa struktúra, ahol a szülő mutató weak

- allocator
  - Minden std osztály használja
  - Lehet saját

Erőforrások kezelése

- C++ fontos tulajdonsága: minden erőforrást kézzel kezelünk
- Nincs garbage collector
  - Memória felszabadítása kézzel
  - Gyors, vagy lassú?
  - Produktív, vagy nem?
  - Ez jó, vagy rossz?

Foglalás és felszabadítás

- Ezek a műveletek mindig párban vannak
- Szabványos fájl API:
  - fopen(): fájl megnyitása
  - fclose(): fájl handle bezárása
- POSIX:
  - socket(): TCP socket létrehozása
  - close(): socket lezárása
- Win32 Mutex API:
  - WaitForSingleObject(): mutex lockolása
  - ReleaseMutex(): mutex elengedése

Hibalehetőség

- Nehéz helyesen használni

```
void readFile()
{
    FILE* file = fopen("test.txt", "r");
    // feldolgozás...
    fclose(file);
}
```

- A fájl nem fog felszabadulni, ha
  - A feldolgozás során kivétel dobódik => try-catch
  - return, (goto) => át kell nézni a kódot, vagy __try-__finally (MS specifikus)
  - Lelőjük a szálat

Megoldás: RAII

- Resource Acquisition Is Initialization
- Az „erőforrást” egy lokális objektummal reprezentáljuk
  - A konstruktorban lefoglaljuk
  - A destruktorban felszabadítjuk
- A lokális objektum mindenképp felszabadul

Példa: fájl handle

- Fájlt reprezentáló osztály

```
class FileHandle {
public:
    FileHandle(const char* n, const char* rw) {
        f = fopen(n, rw);
        if (!f)
            throw "error";
    }
    ~FileHandle() { fclose(f); }
    // ...
private:
    FILE* f;
};
```

Példa: fájl handle

- Egyszerűen lokális objektumként hozzuk létre
- Blokk/függvény végén felszabadul

```
void readFile()
{
    FileHandle("test.txt", "r");
    // Feldolgozás...
    // Nem kell kézzel felszabadítani.
}
```

Memória, mint erőforrás

- Ebből a szempontból a memória nem más, mint egy erőforrás, csak ezt sokkal gyakrabban használjuk
  - malloc/free
  - new/delete
- Az erőforráskezeléshez hasonlóan a memóriakezelést se végezzük kézzel
  - A shared_ptr és unique_ptr típusok pont a RAII mintát valósítják meg a memóriára

Memória foglalás

- Nem RAII – nincs automatizmus
  - malloc, free
  - new, delete
  - new[], delete[]
  - Globális változók és TLS (thread local storage)
    - Ettől még ezek jók lehetnek, de nem szabadulnak fel
- RAII – automatikusan felszabadulnak
  - Globálisan: shared_ptr, unique_ptr, weak_ptr
  - Vermen változó és _alloca

Garbage collection

- Magasabb szintű nyelvekben van GC
  - Automatikusan kezeli a memóriát és szabadítja fel a már nem használt objektumokat
- C++-ban alapból nincs, RAII helyettesíti
  - Third-party GC-implementációk használhatók
- Vannak hibrid nyelvek
  - D
  - Managed C++

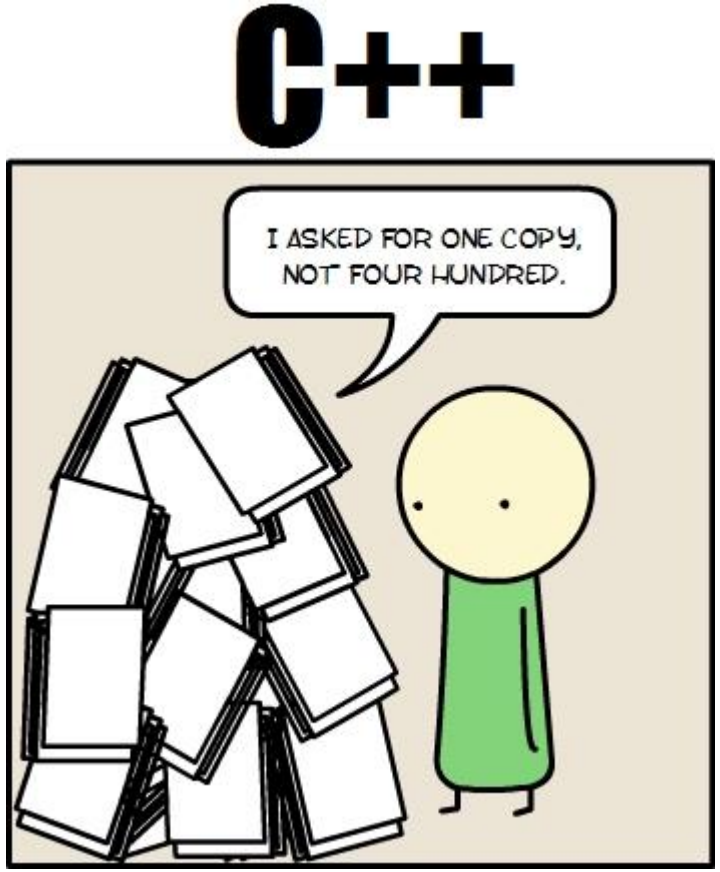
RAII vs. GC

- Mindkettőre igaz
  - Kényelmes – nem kell felszabadítani kézzel
  - Produktív – nehéz hibázni, ha követjük a szabályokat
- Különbségek
  - Hibázási lehetőség kevesebb GC-vel
    - Nem kell figyelni körkörös hivatkozásra
  - RAII determinisztikus, jobban kézben tartható
  - Sebesség más (GC gyorsabb lehet általános esetben)

Move konstruktor

Copy konstruktor elkerülése

- Alapprobléma
  - C++-ban könnyen készül másolat egy objektumról
  - Nagy objektumok vagy sok objektum esetén a másolás drága



Példa

- Hogyan adjunk vissza egy nagyméretű tömböt?

```
? createData();
```

Megoldás	Hátrány
std::vector<int> createData();	Másolás
std::vector<int>* createData();	Memóriakezelés
void createData(std::vector<int>& result);	Nehezen olvasható, operátorokra nem működik

- Az elsőt szeretnénk hátrány nélkül

lvalue, rvalue

- objektum: egy folytonos terület a memóriában
- lvalue: egy objektumra referáló kifejezés, ami tovább él a kifejezésnél (állhat egyenlőség bal oldalán)
  - int i;
- rvalue: minden más (ideiglenes)
  - i + 7;

lvalue, példák

```
int i;
i = 5;

const int ci = 5;
ci = 6; // ERROR

int a[5];
a[3] = 2;

std::string& getStr() { return globalStr; }
getStr() = "modifiy"; // Meglepő szintaktika
```

- a példákban ezek az lvalue-k
  - i
  - ci
  - getstr()
  - a[3]

rvalue

```
std::string createStr() { return "alma"; }

auto addr1 = &(createStr()); // ERROR
auto addr2 = &(std::string("alma")); // ERROR
int i = 2;
(i + 1) = 5; // ERROR
std::string("alma") = "narancs"; // OK (!)
createStr() = "narancs"; // OK (!)
```

- rvalue-k
  - createStr() – értékadás bal oldalán áll, mégsem lvalue
  - std::string("alma")
  - (i + 1)
- Az „Állhat értékadás baloldalán” nem teljeskörű definíció

lvalue, rvalue

```
std::vector<int> createData()
{
    std::vector<int> temp;
    // adatok előállítása...
    return temp;
}
```

- C++ 98: a fordító tudta, hogy mi lvalue, vagy rvalue
  - De nem volt rá mód a nyelvben, hogy megkülönböztessük azokat a kifejezéseket, amik olyan objektumot reprezentálnak, ami épp megszűnőben van

rvalue reference

- C++11: kiegészült a nyelv egy új referenciatípussal: **T&&**
- csak rvalue-hoz köt, módosítható
- Egy megszűnőben lévő objektumot reprezentál, „kilophatjuk” a tartalmát
- Ezzel elkerülhetjük a másolást

Példa: doubleVector pszeudokód

```
class doubleVector
{
    int count;
    int capacity;
    double* elements;
    doubleVector() : capacity(5), count(0), elements(new double[5]){ }
    ~doubleVector() { delete[] elements; }
    void add(double d);
    void remove(int index);
    double get(int index);
};
```

doubleVector COPY ctor

```
class doubleVector
{
    doubleVector(const doubleVector& other) :
        capacity(other.capacity),
        count(other.count),
        elements(new double[other.capacity])
    {
        // Klasszikus COPY ctor, lemásoljuk a teljes tömböt.
        for (int i = 0; i < count; i++)
            elements[i] = other.elements[i];
    }
}
```

doubleVector MOVE ctor

```
class doubleVector
{
    doubleVector(doubleVector&& other) :
        capacity(other.capacity), count(other.count)
    {
        // MOVE: kilopjuk a reprezentációt (memóriát).
        // Nincs másolás.
        elements = other.elements;

        // Töröljük a forrás objektumban a pointert.
        // Fontos, különben a destruktorban felszabadítanánk.
        other.elements = nullptr;
    }
}
```

Mikor hívódik MOVE?

```
doubleVector createVec() { /* ... */ }

void useVec(doubleVector vec) { /* ... */ }

void foo() {
    doubleVector vec;

    useVec(vec); // COPY, mert vec lvalue
    useVec(doubleVector()); // MOVE
    useVec(createVec()); // MOVE
}
```

move manuálisan

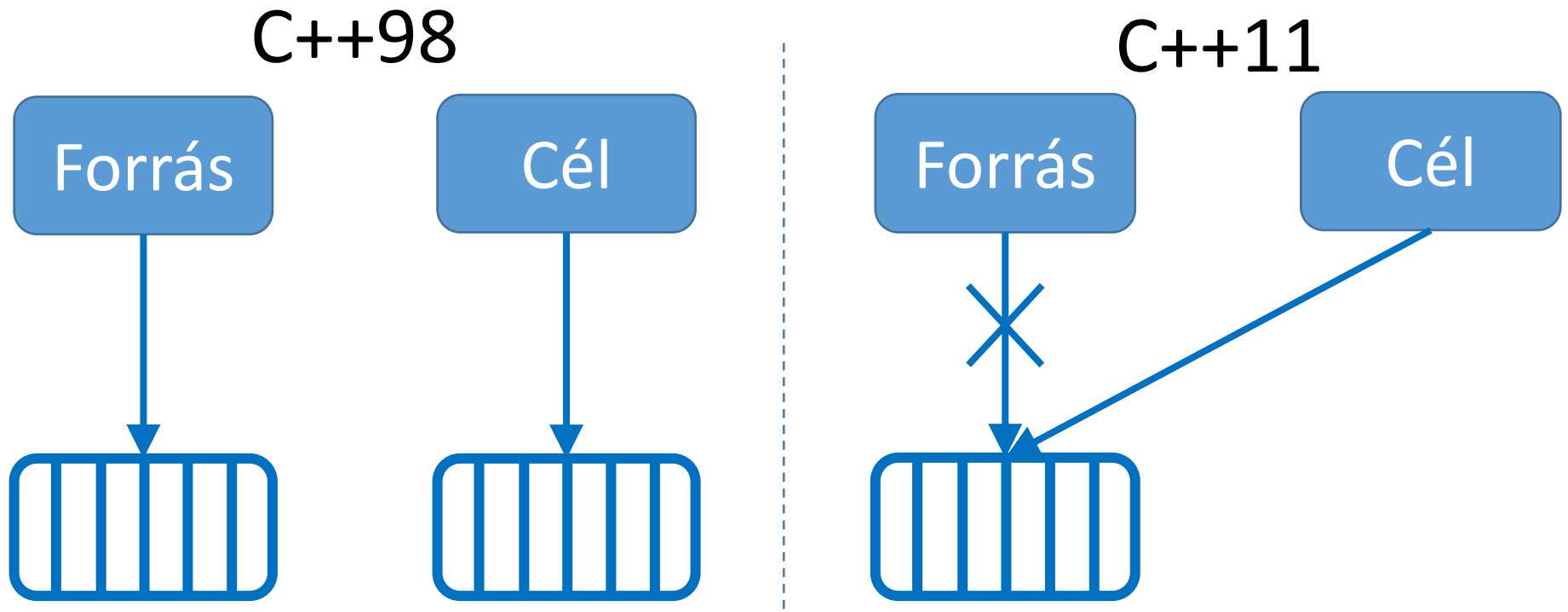
- Ha tudom, hogy egy egyébként lvalue kifejezés által reprezentált objektumot később már nem használok

```
doubleVector vec;
useVec(std::move(vec)); // MOVE
// Ezután már nem szabad vec-
et használni, lehet, hogy a useVec fv. kilopta a tartalmát.
```

- std::move: igazából nem több, mint kasztolás rvalue reference-re
- Így érték szemantikával átadhatunk nem másolható objektumokat is, pl. thread

Visszaadás érték szerint

- Nincs másolás, hatékony
- Az összes STL-konténer támogatja
- Jelentős gyorsulást jelenthet a fordító frissítése



Lambda kifejezések

Logika átadása

- Gyakran szükséges logikát paraméterként átadni, például:
  - algoritmusok
  - callback függvények
  - eseménykezelés
  - strategy pattern (futásidőben választható algoritmus, például plugin)

Logika átadása: függvénypointer

```
void execFunc(void (*func)()) { func(); }
void myFunc() { std::cout << "alma"; }
void foo3()
{
    execFunc(&myFunc);
}
```

- (Lehet referencia is)
- Specifikus típus
- Nehezen olvasható szintaktika
- Tagfüggvények típusa más, nehézkes használni

Példa: sort algoritmus

```
#include <algorithm>
bool cmpStrLength(const std::string& a, const std::string& b)
{
    return a.length() < b.length();
}
std::vector<std::string> strings=...;

std::sort(strings.begin(), strings.end(), cmpStrLength);
```

Functor objektum

```
struct strLengthComparer
{
    bool operator()(const std::string& a, const std::string& b)
    {
        return a.length() < b.length();
    }
};
std::vector<std::string> strings;
strLengthComparer cmp;

std::sort(strings.begin(), strings.end(), cmp);
```

Lambda expressions

- Logika definiálása egyetlen kifejezésben
- Functor objektumot generál

```
std::vector<std::string> strings;
// ...

std::sort(
    strings.begin(),
    strings.end(),
    [] (const std::string& a, const std::string& b)
    {
        return a.length() < b.length();
    });
```

Lambda expressions

- Az átadás helyén van az implementáció, könnyen olvasható
- Érdemes használni, ha
  - egyszerű logikát implementál
  - csak egyetlen helyen van rá szükség
- Egy functor objektum több helyen újra használható
  - Lambát is eltehetjük egy változóba és újra felhasználhatjuk – nem az igazi
- A functor objektum típusának a neve dokumentálja a funkcióját, a lambdának nincs neve

Lambda expressions: capturing

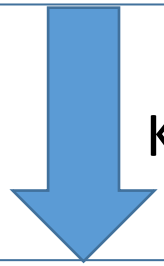
- Ha egy, a lambda scope-jában lévő változót szeretnénk használni a lambda kódjában

```
int count = 0;
std::generate(beg, end, [&count] (){ return count++; });
```

  - [&count]: capture referencia szerint
  - [=count]: capture érték szerint
  - [&]: minden hivatkozott változó referencia szerint
  - [=]: minden hivatkozott változó érték szerint
  - [=, &count]: minden hivatkozott változó érték szerint, kivétel count (ami referencia szerint)
  - []: nincsen capture

Lambda expressions: fordítás

```
std::vector<int> indices(10);
int count = 0;
std::generate(indices.begin(), indices.end(),
    [&count] (){ return count++; });
```



Kódgenerálás a fordítás során

```
struct generatedFuncClass {
    int& capturedInt; // capture by ref!

    generatedFuncClass(int& i) : capturedInt(i) { }

    int operator()() { return capturedInt++; }
};

std::vector<int> indices(10);
int count = 0;
generatedFuncClass gfc(count);
std::generate(indices.begin(), indices.end(), gfc);
```

Lambda expression részei

```
string name = "blue";

auto concatNameWithStr = [&] (string str) -> string
{
    string result = name;
    result.append(str);
    return result;
};

string result = concatNameWithStr("green");
cout << result; // prints "bluegreen"
```

capture list

paraméterlista (opcionális)

visszatérési érték (opcionális)

akció

Lambda expression típusa

```
auto x = [] (int a, int b) { return a < b; }
```

- Változó típusa nem specifikált
  - Gyakorlatban a típusa a fordító által generált functor osztály
- az auto helyére nem tudnánk konkrét típusnevet írni
- std::function: burkoló objektum egy függvényre, functor objektumra vagy lambdára

```
std::function<bool(int, int)> cmpIntsFunc =
    [] (int a, int b) { return a < b; };
```

Lambda expression típusa

- typeid operátorral kiírathatjuk egy kifejezés típusát (RTTI-nek engedélyezve kell lennie)

```
auto cmpInts = [] (int a, int b) { return a < b; };
std::function<bool(int, int)> cmpIntsFunc =
    [] (int a, int b) { return a < b; };
std::cout << typeid(cmpIntsFunc).name() << std::endl;
std::cout << typeid(cmpInts).name() << std::endl;
```

- Kimenet:

```
class std::function<bool __cdecl(int,int)>
class <lambda_b853351245db9a879f640980a0f46f1d>
```

std::sort

- Ezekkel hívtuk meg a sort-ot:
  - függvénypointer
  - functor objektum
  - lambda-kifejezés
  - std::function objektum
- Hogyan tudja a sort fv. bámelyiket fogadni?
  - template paraméter!

Függvény átadása template-tel

- template: kódgenerálás, bármi működik, ha a generált kód fordul

```
template <typename Comparer>
void doubleSort(vector<double>& vec, Comparer cmp)
{
    // sort algoritmus kódja
    int i;
    // ebben valahol használjuk a cmp-t összehasonlításra.
    if (cmp(vec[i], vec[i + 1]))
    {
        // ...
    }
    // ...
}
```

Függvény átadás template-tel

- Az előző példában így használtuk a cmp paramétert

```
if (cmp(vec[i], vec[i + 1]))
```

- A függvény template-nek bármilyen olyan cmp argumentumot átadhatunk, amire a fenti kódsor fordul
  - függvénypointer bool (double, double) függvényre
  - függvényreferencia bool (double, double) függvényre
  - funktor objektum
  - lambda
  - std::function (ez egy szabványos funktor objektum)
- A template típusú paraméter jó olyan esetben, ha nem tudjuk előre a paraméter típusát, vagy a típust nem lehet kifejezni a nyelvben

std::sort szabványos szignatúrája

```
template <class RandomAccessIterator, class Compare>
void sort(
    RandomAccessIterator first,
    RandomAccessIterator last,
    Compare comp);
```

Inicializálás

initializer lists

- C++11-től kezdve nem csak tömbök inicializálhatók {3, 5, 11} szintaktikával

```
vector<double> v = { 1, 2, 3.456, 99.99 };
list<pair<string, string>> languages = {
    { "Nygaard", "Simula" }, { "Richards", "BCPL" }, { "Ritchie", "C" }
};
map<vector<string>, vector<int>> years = {
    { { "Maurice", "Vincent", "Wilkes" }, { 1913, 1945, 1951, 1967, 2000 } },
    { { "Martin", "Ritchards" }, { 1982, 2003, 2007 } },
    { { "David", "John", "Wheeler" }, { 1927, 1947, 1951, 2004 } }
};
```

initializer list támogatása

- Új konstruktor

```
template<class E> class myVector {
    E* elements;
    int size;
public:
    // initializer-list constructor
    myVector(initializer_list<E> s)
    {
        elements = new E[s.size()];
        copy(s.begin(), s.end(), elements);
        size = s.size();
    }
};
```

uniform initialization

- C++ 98-ban több szintaktika van objektumok incializációjára:

```
string a[] = { "foo", " bar" }; // OK: tömb
vector<string> v = { "foo", " bar" }; // Hiba: nem tömb
void f12(string a[]);
f12({ "foo", " bar" }); // Hiba: blokk argumentumként

int a = 2;                  // assignment style
int aa[] = { 2, 3 };        // assignment style listával
complex z(1, 2);             // functional style
x = Ptr(y);                 // functional style konverzióhoz/kasztoláshoz

int a(1); // változó definíciója
int b(); // függvény deklarációja
int b(foo); // változódefiníció vagy függvénydeklaráció
```

uniform initialization

- C++11: egységes szintaktika kapcsos zárójelekkel

```
X x1 = X{1, 2};
X x2 = {1, 2}; // Opcionális egyenlőségjel
X x3{1, 2};
X* p = new X{1, 2};
struct D:X{
    D(int x, int y):X{x, y}{};
};
struct S{
    int a[3];
    // Régi problémára megoldás:
    S(int x, int y, int z):a{x, y, z}{};
};
```

Tag inicializáció (C++20)

- Mint C# var a=A{m=2};

```
struct A {
  string str;
  int n = 42;
  int m = -1;
};
auto a=A{.m=21}
```

uniform initialization

- Nem végez szűkítő converziót (narrow cast)

```
int x=2.5; // OK, truncates
int x{2.5}; // ERROR, narrowing conversion
```

- auto + kapcsos zárójelek: initializer_list!

```
auto x1=5; // int
auto x2(5); // int
auto x3{5}; // std::initializer_list<int>
auto x4={5}; // std::initializer_list<int>
```

Kérdések?

Multiplatform szoftverfejlesztés

Interop

Interop

- Fogalmak
  - Natív kód, natív réteg
    - (Natív) C++-ban megírt kód
  - Platform kód
    - C#, Java, Swift, … ami az adott platformon használt
  - Adapter
    - A két réteget összekötő kód, wrapper
- A platformszintű technológia és a natív C++ közötti együttműködés nem triviális
  - A két réteg között interop technológiára van szükség

Cél

- Miért akarunk C++-t hívni platform kódból?
  - Létező kódot kell bekapcsolni a rendszerbe
    - Algoritmusok
    - Integráció más rendszerrel
  - Adott funkcionalitás csak így érhető el
    - Például Unity natív plugin, .NET C#-ból elérhetetlen része
  - Multiplatform közös kód elérése
    - Rossz példa: Xamarin .NET Standard Library
      - Mert mindkét oldal C#
  - Optimalizáció
    - Legkevésbé gyakori

Interop, miért van rá szükség?

- Hívási konvenciók
  - Verem kezelése
- Típusrendszer
  - Ez jelentősen eltérő lehet, ami nagyon megnehezíti az áthívást
- Kivételkezelés
  - A hasonlóság általában kicsi a két réteg között
- Adatok bináris reprezentációja
  - Tipikus probléma a string
  - Igazítás (layout) is kérdéses

Interop, miért van rá szükség?

- Memóriamodell különbségei
  - C++-ban kézzel kezeljük a memóriát, memóriacímeket
  - Java-ban, .NET-ben szemétgyűjtő kezeli a heap-et, átmozgathatja az objektumokat, nem használhatunk natív pointereket (vagy csak nehezen)
- OO koncepciókat (objektum, adattagok, virtuális függvények) máshogy reprezentáljuk a memóriában

Interop megoldások

- A problémák elkerülése végett klasszikusan alacsony szintű
  - Nem kell foglalkozni a legtöbb eltéréssel
  - string és társai így is gond, azt kezelni kell
  - Csak primitív adatokat tudunk átadni
  - Kivételkezelés nincs – meg kell írni
- C-stílusú globális függvények meghívására van lehetőség
- Például: .NET P/Invoke

.NET – C interop, P/Invoke példa

- C++:

```
extern "C" {
    __declspec(dllexport) void nativeFunc(int i) {
        printf("Number passed is %d.\r\n", i);
    }
}
```

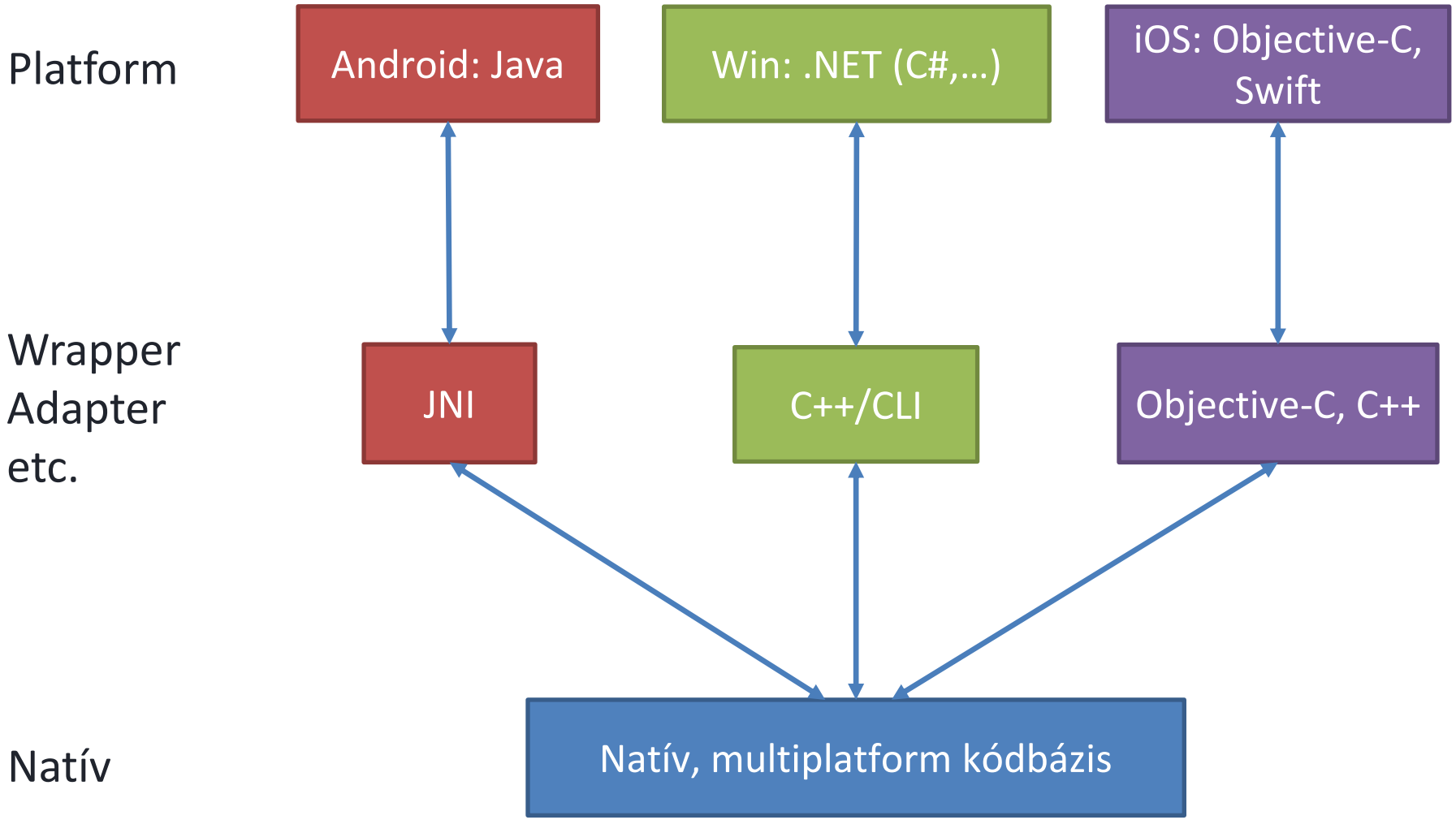
- .NET, C#:

```
[DllImport("NativeLib.dll")]
public static extern void nativeFunc(int i);

static void Main(string[] args) {
    nativeFunc(5);
}
```

Interop cél

- Bár működik, ennél többet szeretnénk
  - Minél kényelmesebben elérhető legyen a natív kódbázis a platform oldaláról
    - Bónusz, ha a natív oldal el tudja érni a platform kódot
  - Minél kisebb legyen az áthívás overheadje
    - Marshaling: adatok másolása/konvertálása
  - Minél kisebb munka legyen implementálni a köztes réteget
- A következő technológiák többet tudnak
  - JNI, Objective-C++, C++/CLI

Interop, technológia



Java Native Interfaces

JNI – Java Native Interface

- Infrastruktúra, ami ahhoz kell, hogy Javából elérjünk natív technológiákat (pl. C++), és fordítva
  - Többet tud egy sima C függvény meghívásánál
- Szükséges, ha
  - Platformszintű API-t nem lehet elérni Javából, csak mondjuk C++-ból
  - Már létező natív könyvtárat szeretnénk egy Java alkalmazásban használni
- https://developer.android.com/training/articles/perf-jni.html

JNI alapok

- Java kódban deklarálhatunk **natív** metódusokat
  - Ezek implementációja a natív rétegben van
- A natív metódus meghívására a JNI framework továbbítja a hívást a natív komponenshez
- Így lehet áthívni Javából egy C vagy C++-os osztálykönyvtárba
- Ezzel közvetlenül globális függvényeket hívhatunk meg

JNI Hello World!

Natív függvény

```
JNIEXPORT void JNICALL
Java_mypackage_HelloWorld_print
( JNIEnv *env, jobject obj )
{
    printf("Hello world\n");
}
```

Java osztály

```
package mypackage;

class HelloWorld {
    public native void print(); // Natív fv
    static // Statikus inicializáció.
    {
        // Natív library betöltése.
        // CLibHelloWorld.dll
        System.loadLibrary("CLibHelloWorld");
    }
    public static void main(String[] args) {
            HelloWorld hw = new HelloWorld();
            // Natív függvény meghívása.
            hw.print();
        }
    }
}
```

Natív függvény argumentumai

```
JNIEXPORT void JNICALL Java_mypackage_HelloWorld_print
(JNIEnv* env, jobject obj)
{
   printf("Hello world\n");
}
```

- Első paraméterben megkapjuk a JNI környezetre mutató pointert
- Második paraméterben azt a Java objektumot, amiből a hívás érkezett
- Ha definiáltunk volna még paramétert a metódusnak, az ezek után szerepelne

JVM elérése C++-ból

- JNIEnv* pointeren keresztül elérhetőek a Java futtatókörnyezet (JVM) szolgáltatásai
  - JVM típusok felderítése
  - Típusok mezőinek, metódusainak lekérése
  - Java objektumok létrehozása
  - Mezők beállítása
  - Metódusok meghívása
  - Egyéb: stringek létrehozása, tömbök kezelése, kivételkezelés, szálkezelés

Metaprogramozás

- A JNI használata egyfajta string alapú metaprogramozás
  - Mint .NET Reflection
- Objektumok fogják reprezentálni a Javás típusokat, a típusok mezőit, metódusait
- Ezeket a nevük és típusuk stringként való megadásával tudjuk lekérdezni

Java típusok elérése

- Egy osztály elérhető a neve alapján
- Az osztállyal kapcsolatos többi funkció az így visszakapott objektumon keresztül érhető el

```
package hu.jnitest;
public class Vector {
    public double X;
    public double Y;
    public double Length() { ... }
    public Vector Multiply(double d) { ... }
    public Vector Add(Vector v) { ... }
}
```

Java osztály lekérése

- Az osztály reprezentáló objektum lekérhető a környezettől a nevével azonosítva

```
void Foo(JNIEnv env*) {
    jclass vectorClass = env->FindClass("hu/jnitest/Vector");
}
```

- Működik saját és beépített típusokra is

Mezők, metódusok lekérése

```
jclass vectorClass = env->FindClass("hu/jnitest/Vector");
jfieldID xId = env->GetFieldID(vectorClass, "X", "D");
jmethodID ctorId = env->GetMethodID(vectorClass, "<init>", "()V");
jmethodID lengthId = env->GetMethodID(vectorClass, "Length",
    "()D");
jmethodID multiplyId = env->GetMethodID(vectorClass, "Multiply",
    "(D)Lhu/jnitest/Vector;");
```

- Mező lekérése név és típus megadásával
- Metódus lekérése név és szignatúra megadásával
- Konstruktor lekérése speciális <init> névvel

Metódusszignatúrák

- A név nem elég egy metódus azonosítására
  - Szükség van a szignatúrára is (polimorfizmus miatt)
- Formátum (…)…
  - (argumentum-típusok)visszatérési érték típusa
- Primitív típusok leírója egy karakter
  - Pl.: boolean – „Z”, char – „C”, void – „V”
- Osztályok leírója L karakterrel kezdődik, utána a teljes package és az osztály neve, a végén egy pontosvesszővel
  - “Ljava/lang/String;”, “Lhu/jnitest/Vector;”
- Tömbök leírója [karakterrel kezdődik
  - Pl. egy boolean tömb: „[Z”

Metódusszignatúrák – példa

```
Vector[] f (String s, int[] arr);
```

(Ljava/lang/String;[I)[Lhu/jnitest/Vector;

- Sok a hibalehetőség, könnyű elrontani
- Nincs fordításidejű ellenőrzés

Java objektum létrehozása, beállítása

- A korábban lekért jclass, jfieldid és jmethodid objektumokkal hozható létre új példány

```
jobject CreateJavaVector(double x, double y, JNIEnv* env)
{
    jobject javaObj = env->NewObject(vectorClass, ctorId);
    env->SetDoubleField(javaObj, xId, x);
    env->SetDoubleField(javaObj, yId, y);
    return javaObj;
}
```

- Mező értéke lekérdezhető a GetXField függvényekkel

Metódus meghívása

- Metódus meghívható a CallXMethod függvényekkel, ahol X a visszatérési érték típusára utal. Át kell adni
  - az objektumot, amin a metódust hívjuk
  - a metódus azonosítóját (jmethodid)
  - az argumentumokat

```
double AddAndMultiply(jobject vec1, jobject vec2, double x, JNIEnv* env) {
    jobject sum = env->CallObjectMethod(vec1, addId, vec2);
    jobject multiplied = env->CallObjectMethod(sum, multiplyId, x);
    jdouble result = env->CallDoubleMethod(multiplied, lengthId);
    return result;
}
```

Stringek kezelése

- A javás string egy objektum a managed heapen
  - Nem csak egy sima karakterlánc
- Nem kompatibilis a natív C++ stringekkel
  - Konvertálni kell a két réteg között
  - UTF-8 és UTF-16 enkódolást támogat a JVM
    - C++ is támogatja ezeket, így elképzelhető, hogy maga a kódolás megegyezik
- Új string objektum létrehozása

```
jstring str = env->NewStringUTF("test string");
```

Stringek kezelése

- String hosszának lekérdezése

```
jsize len = env->GetStringLength(str);
```

- String nyers tartalmának lekérdezése (az így megkapott tömb használható megszokott módon natív C++-ban)

```
jchar* charsUTF16 = env->GetStringChars(str, nullptr);
char* charsUTF8 = env->GetStringUTFChars(str, nullptr);
```

- Használat után a karaktereket fel kell szabadítani a ReleaseStringChars vagy a ReleaseStringUTFChars függvényekkel

Tömbök kezelése

- Tömböket saját típusok reprezentálják: jintArray, jdoubleArray, jobjectArray, stb.
- Új tömb létrehozása

```
jintArray arr = env->NewIntArray( 5 );
```

- Primitív-tömb elemeinek beállítása
  - Egyszerre több elemet másol át

```
env->SetIntArrayRegion( arr, startIndex, length, nativeArray );
```

Tömbök kezelése

- Elemek lekérdezése
  - GetIntArrayElements: összes elem
  - GetIntArrayRegion: egy része
- Objektum-tömbök elemeinek beállítása egyenként történik
  - GC-vel együtt kell működni

```
jobject string = env->NewString(text.data(), text.length());
env->SetObjectArrayElement(array, index, string);
```

Lokális referenciák

- GC-vel együtt kell működni
- Amikor kapunk/létrehozunk egy referenciát, akkor a GC nem mozgathatja (pin/lock)
  - Amíg fut a kódunk
- Csak az adott natív hívás kontextusában használható
  - Visszatéréskor automatikusan felszabadul
  - Memory leak biztos
- Kézzel is fel lehet szabadítani hamarabb

```
env->DeleteLocalRef(javaObject);
```

Globális referenciák

- Ha később is szükségünk van egy referenciára
  - Eltároljuk
  - Háttérszálat indítunk, amiben használjuk
- Akkor globális referenciát kell rá létrehoznunk

```
jclass vectorClass = env->FindClass("hu/jnitest/Vector");
vectorClass = env->NewGlobalRef(vectorClass );
```

- Ezt kézzel fel kell szabadítani, magától nem szabadul fel

```
env->DeleteGlobalRef(vectorClass );
```

Szálkezelés, JNIEnv

- Minden natív függvény megkapja a JNIEnv* pointert
- Ha nem egy natív fv. kontextusában vagyunk, elkérhetjük a JVM-től. Erre egyszer el kell tárolnunk egy referenciát, ami sosem változik:

```
JavaVM* jvm;
int ret = env->GetJavaVM(&jvm);
```

- A JavaVM-től bármikor elkérhetjük a környezetet

```
JNIEnv * env;
int ret = jvm->GetEnv((void **)&env, JNI_VERSION_1_6);
```

Szálkezelés, JNIEnv

- Natív oldalról indított szálból
  - A szálat be kell regisztrálnunk

```
ret = jvm->AttachCurrentThread( &env, nullptr);
```

  - Ez nem lassít, csak létrehoz a JVM-ben egy gyűjteményt, ahova tárolni lehet a referenciákat

Referencia natív objektumra

- A natív objektumok nem a felügyelt heap-en vannak, nem tud róluk a JVM
  - Java referenciával nem tudunk natív objektumot kezelni
- Viszont egy natív objektum címe létrehozás után nem változik a memóriában
  - Címét egy longként a Java oldalnak átadva eltárolhatjuk
  - Visszaküldhetjük a natív rétegnek, ahol mutatóvá kell kasztolni

Referencia natív objektumra

- Natív objektum címének felküldése
  - 32 biten

```
MyNativeObj* obj = new MyNativeObj();
long address = reinterpret_cast<long>(obj);
env->CallVoidMethod(javaClass, storeAddressMethod, address);
```

  - 64 biten long long kell
- Cím visszaküldése egy natív metódusnak

```
MyNativeObj* obj =
    reinterpret_cast<MyNativeObj*>(nativeEnginePointer);
obj->Foo();
```

C++/CLI

C++/CLI

- Nyelvi kiegészítések a C++ nyelvhez
  - Elérhetőek/létrehozhatóak .NET-es típusok
- CLI: Common Language Infrastructure
  - Nyílt szabvány, aminek egyik megvalósítása a .NET Framework
- Ezekkel a nyelvi kiegészítésekkel használhatók a .NET platform konstrukciói és szolgáltatásai, így Windows-ra írt alkalmazásokkal lehet C++-ból együttműködni
  - Xamarin nem támogatja

C++/CLI céljai

- A .NET és a natív réteg közötti különbségek hasonlóak, mint a Java esetén
  - Más típusrendszer
  - Objektumok máshogy vannak reprezentálva a memóriában
  - .NET oldalon felügyelt heap szemétgyűjtéssel, C++ oldalon natív heap manuális memóriakezeléssel
- Ami Javában nincs: .NET oldalon két alapvetően eltérő user-defined típus van: értéktípus és referenciatípus, a kettőt máshogy kell tárolni, átadni

Windows interop, példa

Natív (C++/CLI)

```
namespace NativeLib
{
    public ref class MyClass sealed
    {
    public:
        Platform::String^ CreateString()
        {
            return L"alma";
        }
    };
}
```

Nyelvi kiegészítések
C++/CLI

Platform (C#)

```
var c =
    new NativeLib.MyClass();

string s = c.CreateString();

Console.WriteLine(s);
```

C++/CLI assembly

- Speciális osztálykönyvtár, amiben keveredhet a C++/CLI kód és a natív C++ kód
- C++/CLI-ben a .NET osztálykönyvtárának (BCL), és más .NET-es szerelvényeknek bármelyik osztálya elérhető és használható
- Lehet include-olni natív headeröket
- Lehet hozzá linkelni natív libraryket
- .NET-ből elérhető
- A benne definiált C++/CLI típusok .NET-ből ugyanúgy használhatók, mintha bármilyen más .NET-es nyelvben lettek volna implementálva (pl. C#-ban)

Nyelvi kiegészítések: tracking handle

- Tracking handle: speciális referencia, ami egy managed heapen lévő objektumra mutat

```
String^ str = "Ez egy .NET-es string";
auto di = gcnew DirectoryInfo("C:/Temp");
auto strArr = gcnew array<String^>(10);
```

- Tagok elérése: pointerhez hasonlóan -> operátorral

```
Console::WriteLine(di->FullName);
```

Nyelvi kiegészítések: tracking handle

- gcnew: Speciális operátor a new helyett, ami a natív heap helyett a felügyelt heapen hoz létre egy .NET-es objektumot
- Az így létrehozott objektumokat nem kell manuálisan felszabadítani
  - Nincs gcdelete, delete nem működik rajta
  - Hasonló a használatuk egy okos pointerhez, de nem referenciaszámlálással működnek, hanem a .NET megszokott szemétgyűjtésével
- Ez a nyelv az egyik ritka esete az opcionális GC-nek

Nyelvi kiegészítések: .NET típusok

- Natív osztályok létrehozása
  - class
  - struct
- .NET-es típusok létrehozása
  - public ref class: .NET-es referencia típus (class)
  - public value struct: .NET-es érték típus (struct)
  - public interface class: .NET-es interfész
  - public enum class: .NET-es enum
    - public nélkül C++11 nyelvi elem

ref class példa

```
public ref class ManagedPerson {
    int age;
    System::String^ name;
public:
    ManagedPerson(int age, System::String^ name) : age(age), name(name) { }
    property int Age {
        int get() { return age; }
        void set(int value) {
            if (value < 0)
                throw gcnew System::ArgumentException("Age must be 0+");
        }
    }
    property System::String^ Name {
        System::String^ get() { return name; }
        void set(System::String^ value) { name = value; }
    }
};
```

ref class példa

- A public kulcsszó azt jelenti, hogy elérhető legyen ez a típus más szerelvényekben is
- A .NET-es nyelvek (pl. C#) funkciói elérhetők C++/CLI-ben és fordítva
- Ez magas szintű, mindkét oldalon egyszerűen kezelhető interopot tesz lehetővé

Referenciatípusok használata

- Létrehozhatjuk a felügyelt heapen is, ilyenkor a szemétgyűjtő kezeli és szabadítja fel

```
auto personOnHeap = gcnew ManagedPerson(25, "Peter");
```

- Létrehozhatjuk a stacken is, ilyenkor automatikusan felszabadul, mint a natív C++ objektumok

```
ManagedPerson personOnStack(32, "John");
```

  - C#-ban erre nincs lehetőség, hogy referenciatípusokat a stacken hozzunk létre

Value struct példa

```
public value struct Color
{
    unsigned char A;
    unsigned char R;
    unsigned char G;
    unsigned char B;
};
```

- .NET-es érték típus lesz – nem terheli a GC-t
- Tömbben nagy mennyiségben hatékonyan létrehozható
- Tipikusan a stacken hozzuk őket létre és érték szerint adjuk át
  - A felügyelt heapen is létrehozhatjuk, de akkor nem közvetlenül az érték fog tárolódni, hanem egy csomagoló objektum (boxing)

Natív és felügyelt típusok együtt

- Elég szabadon keverhető, néhány megkötéssel

```
class NativeClass
{
public:
    NativePerson NativeMethod(NativePerson p); // OK
    ManagedPerson^ ManagedWithHandle(ManagedPerson^ mp); // OK
    ManagedPerson ManagedByValue(ManagedPerson mp); // OK
    NativePerson nativePerson; // OK
    ManagedPerson^ managedPerson; // ERROR: natív típusban nem
lehet ref
    ManagedPerson managedPersonByValue; // ERROR: natív típusban
nem lehet ref class érték szerint
    gcroot<ManagedPerson^> ManagedPerson; // OK
    Color valueTypeField; // OK
};
```

gcroot<T>

- A natív kód nem tud a felügyelt referenciákról és a GC-ről
- A szemétgyűjtés miatt a felügyelt heapen lévő objektumok pozíciója a memóriában megváltozhat
  - Nem tárolhatunk felügyelt objektumra mutatót
- gcroot<T>: segédosztály, amivel egy felügyelt objektumra tárolhatunk referenciát natív típusban
  - A System::Runtime::InteropServices::GCHandle segédosztályt használja

Natív és felügyelt típusok együtt

```
public ref class ManagedClass {
public:
    NativePerson NativeType(NativePerson p); // OK, de csak C++-ból lesz hívható, C#-ból nem.
    ManagedPerson^ ManagedWithHandle(ManagedPerson^ mp); // OK
    ManagedPerson ManagedByValue(ManagedPerson mp); // OK, de szokatlan, C#-ban nem is így jelenik meg a visszatérési érték, hanem kimeneti paraméterként, mert referenciatípusokat C#-ban nem lehet érték szerint átadni.
    NativePerson nativePerson; // ERROR: felügyelt típusban nem lehet natív mező.
    NativePerson& npRef; // OK: Referencia lehet.
    NativePerson* npPtr; // OK: És pointer is.
    ManagedPerson^ managedPerson; // OK.
    ManagedPerson managedPersonByValue; // OK.
    Color valueTypeField; // OK
};
```

Natív típusok a felügyelt osztályban

- Az előző példából:

```
public ref class ManagedClass {
    ...
    NativePerson& npRef; // OK: Referencia lehet.
    NativePerson* npPtr; // OK: És pointer is.
    ...
};
```

- Ezek gond nélkül tárolhatók
- Egy natív referencia vagy pointer gyakorlatilag egy memóriacím, mintha csak egy long típusú mező lenne
- C#-ban ezek nem jelennek meg
  - A natív típusokat a .NET típusrendszere nem ismeri

Generikus típusok

- .NET generikus típusok funkciója hasonló, mint a C++ template-eké, de a működési elvük más. A szintaktikájuk is hasonló:

```
generic<typename T> where T : Shape
ref class GeometryManager {
    void Store(T^ shape);
    void Draw(T^ shape);
};
```

where T : Shape: Megkötés a típusparaméterre (C++20-ban van template-re is)

Generikus típusok vs. template-ek

- Generikus típusok: .NET-es runtime mechanizmus
  - Benne marad az IL kódban
- C++ template-ek: fordításidejű kódgenerálás
  - Fordítás közben eltűnik
- A kettő tud egyszerre működni
  - Template generikus osztály
  - Fordítás után marad a generikus típus minden olyan változatban, amilyen típussal használtuk a template-et

Egyéb

- .NET-es tömb: `array<int>^ ints;`
- Többdimenziós tömb: `array<String^, 2>^ strs;`
- Delegate-ek
- Minden meg van valósítva

C# != .NET

```
public ref class ClassWithIndexer
{
public:
    property int default[int] // Fordul, elérhető C#-ból (indexer)
    {
        int get(int index) { return 0; }
        void set(int index, int value) {}
    }
    property int prop2[int] // Fordul, de nem érhető el C#-ból, reflectionnel igen
    {
        int get(int index) { return 0; }
        void set(int index, int value) {}
    }
};
```

C++/CLI, összefoglalás

- Magas szintű megoldás
  - Egy új nyelvet kellett hozzá létrehozni
- A .NET-es, C#-ban elérhető funkciók leképezése C++-ra nyelvi kiegészítésekkel
- C# oldalról megszokott .NET API-t látunk
- Fontos: A fordító függvényenként más
  - .NET-es függvényekre a .NET-es + JIT
  - Natív C++-os függvényekre az optimalizáló fordító – ami gyorsabb kódot eredményez

Objective-C++, Swift

Objective-C – C++ interop

- Tud C-stílusú függvényeket hívni
  - Ugyanaz a probléma, mint az összes többi platformnál
- A fordító (Clang) mindkét kódot képes fordítani!
  - Egy fájlban is
- Könnyű az átjárás, Objective-C és C++ szabadon keverhető
- Az objektummodellek eltérései miatt van néhány megkötés, például:
  - Objective-C osztály nem származhat C++ osztályból és fordítva
  - Ha C++ típusú mezőt tartalmaz egy Objective-C osztály, annak csak a default konstruktorát tudja meghívni

Objective-C és Objective-C++

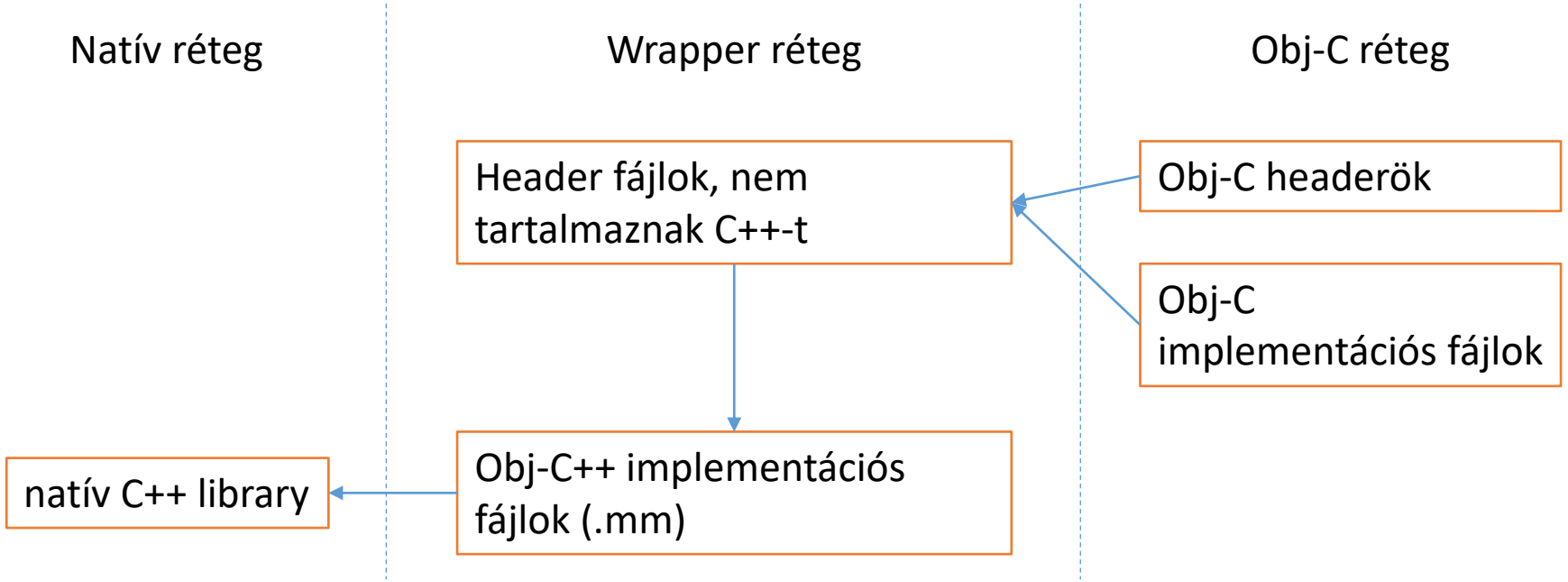
- C++ és Objective-C együttes használatához egy külön fordítási mód (Objective-C++) kell
- Obj-C fájlok kiterjesztése .m, Obj-C++ fájloké .mm
- Ha egy Obj-C headerben include-olunk C++-kódot

```
#include <map>
@interface MyClass : NSObject {
    @private
    std::map<int, id> lookupTable;
}
// ...
@end
```

  - Innentől kezdve ezt a headert csak Obj-C++ fájlokban lehet include-olni
  - Láncreakció, lehet, hogy a teljes projektet Obj-C++ fordításra kell átalakítani, ez egy nagy projektnél problémás lehet

Wrapper réteg

- Bár keverhető a C++ és az Objective-C, itt is érdemes tudatosan szétválasztani a kettőt
- Csak a kettő közötti interop réteg legyen Objective-C++ módban fordítva



Wrapper réteg, megoldás 1: PIMPL

- Pointer to Implementation
- A headerben csak egy struct forward declaration és egy pointer, ezt Obj-C-ben is szabad
- Header fájl (.h):

```
struct MyClassImpl;

@interface MyClass : NSObject {
  @private
  struct MyClassImpl* impl;
}
// public method declarations...
- (id)lookup:(int)num;
@end
```

PIMPL

- C++ implementáció használata csak az .mm forrásfájlban
- Itt már lehet C++ típusokat használni

```
#import "MyClass.h"
#include <map>
struct MyClassImpl {
  std::map<int, id> lookupTable;
};
@implementation MyClass
- (id)init {
  self = [super init];
  if (self)  {
    impl = new MyClassImpl;
  }
  return self;
}
- (void)dealloc {
  delete impl;
}
- (id)lookup:(int)num {
  std::map<int, id>::const_iterator found =
    impl->lookupTable.find(num);
  if (found == impl->lookupTable.end()) return nil;
  return found->second;
}
@end
```

Wrapper réteg, megoldás 2: class extensions

- Obj-C 2.0 óta közvetlenül az implementációs fájlban deklarálhatók a tagváltozók (instance variable, ivar). Ebben az esetben ezek privátok lesznek, és a headerben nem is szerepelnek. Így itt használhatunk natív C++ típusokat, a headert pedig Obj-C-ben is include-olhatjuk.
- Header:

```
#import <Foundation/Foundation.h>

@interface ObjcObject : NSObject
- (void)exampleMethodWithString:(NSString*)str;
// other wrapped methods and properties
@end
```

Wrapper réteg, megoldás 2: class extensions

- Implementációs fájl (.mm), itt deklaráljuk a C++-ban implementált típussal rendelkező mezőt:

```
#import "ObjcObject.h"
#import "CppObject.h"

@implementation ObjcObject {
  CppObject wrapped; // Ez az objektum C++-ban van implementálva, akik a headert látják, azok erről nem tudnak.
}

- (void)exampleMethodWithString:(NSString*)str
{
  std::string cpp_str([str UTF8String], [str lengthOfBytesUsingEncoding:NSUTF8StringEncoding]);
  wrapped.ExampleMethod(cpp_str);
}:
```

Swift – C++ interop

- Tud hívni C-stílusú függvényeket
  - Közvetlenül hívjuk, ahol szükséges
    - Hibalehetőség, az interop kód szét van szórva, nehéz kézben tartani
  - Készíthetünk csomagoló osztályt minden függvényhez
    - C++ oldalon C-stílusú függvények hívnak tovább
    - Swift oldalon Swift csomagoló osztályt készítünk
- Nincs más interop lehetőség C++ hívásra
- Viszont tud hívni Objective-C-t!
  - Dupla csomagolás, Swift – Objective-C – C++
  - Ez macerás, de robosztus megoldás
  - Fordításidőben látjuk a hibákat

Kérdések?

Multiplatform szoftverfejlesztés

Qt alapok

Mi a QT?

- Teljes alkalmazás fejlesztői keretrendszer
  - C++ osztálykönyvtár
  - GUI
    - De nem csak az – hálózat, SQL, unit test, multimédia, stb.
  - QT Creator
    - Kód szerkesztő
    - Fordító választható (VS, GCC, Clang, stb.)
    - Debugger (QML debugger is)
    - Source Control integrált
  - Multiplatform támogatás
  - Dokumentáció, tutorials, példakódok

Felhasználása

- Mindenen fut, ami programozható
  - Desktop (Windows, Linux, Mac OS) és szerverek
  - Beágyazott rendszerek
  - Mobil eszközök (iOS, Android, …)
- Elterjedt
  - Skype, Spotify, VLC, Maya, Google Earth
- Licence
  - Ingyenes, LGPL
  - Fizetős/open source, ha a teljes csomag kell (pl. Qt Charts)
  - Qt módosítása csak akkor lehetséges, ha open source a módosítás

Qt C++ Hello World

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel label("Hello World!");
    label.show();
    return app.exec();
}
```

Qt Quick

- QML – deklaratív felület leírás, ami kódot is tartalmazhat
- Vezérlők gyűjteménye
- Model-View támogatás
- Animációs keretrendszer
- JS futtató motor
- Minden platformon natív kinézet
  - De meg is adható a stílus

QML betöltő

- QQmlApplicationEngine
  - Akárhogyan létre lehet hozni (vermen is)
  - Destruktor töröl mindent
- Load, vagy konstruktor létrehozza az objektumokat
- QUrl a resource fájlba tud hivatkozni
  - Sima fájlt is be lehet tölteni

```
#include <QApplication>
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QQmlApplicationEngine
engine(QUrl("qrc:/main.qml"));
    return app.exec();
}
```

QML Hello World

```
import QtQuick 2.4
import QtQuick.Controls 1.3
import QtQuick.Window 2.2
import QtQuick.Dialogs 1.2

ApplicationWindow {
    title: "Hello World"
    width: 640
    height: 480
    visible: true
}
```

Qt Quick

Vezérlők

QML nyelv

- Felület deklarálására
  - Miért jó deklaratívan megadni felületet?
- JavaScript Object szerű (nem JSON)
  - Egyszerű
  - Picit kényelmesebb, mint XML
- Struktúra megegyezik a hierarchiával
  - Szinte mindig
- Tartalmazhat kódot, JavaScript

QML vezérlők

- Típus (Button)
- Tulajdonságok (text)
- Értékek (“hello”)
- Bármibe tehetünk bármit
  - Buttonban Rectangle
  - A Rectangle itt rárajzol a Button szélére
- Sok hasonlóság XAML-lel
  - Egyszerűbb (jó és rossz is)
  - Gyorsabb/kevesebb memóriát eszik

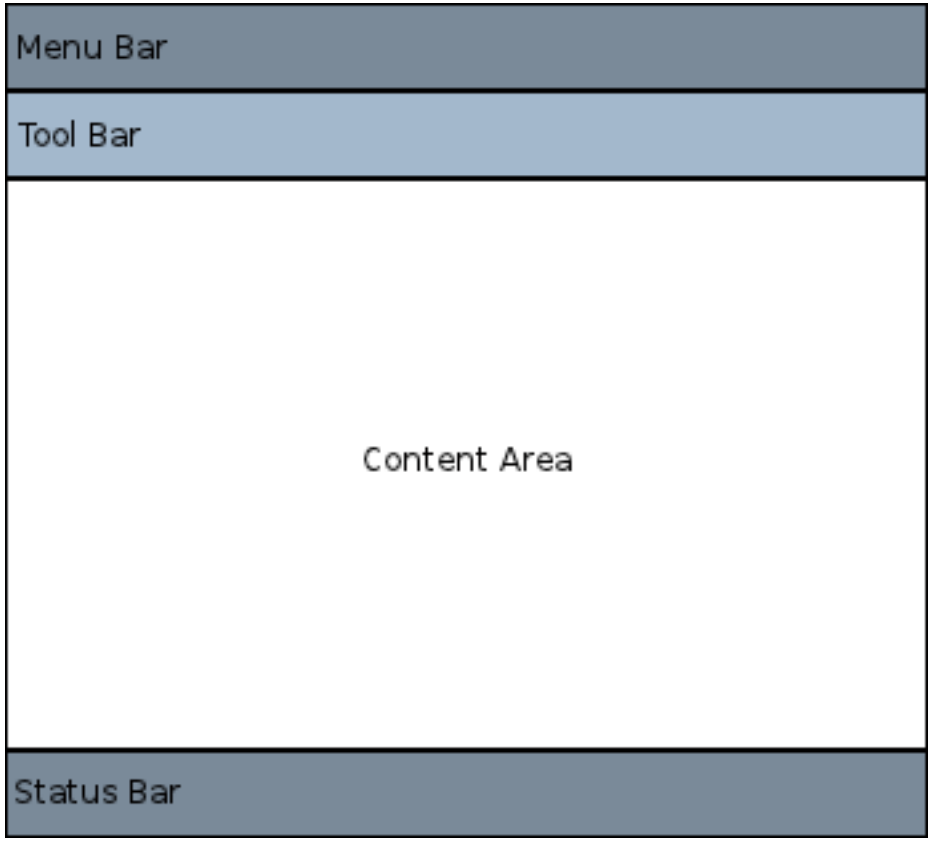
```
Button{
    text: "hello"
    Rectangle{
        width: 10
        height : 10
        color : "red"
    }
}
```

QML alaptípusok

- Fontosabbak
  - bool, int, double, string, list, url
  - color, date, font
  - point, rect, size,
  - vector2d, vector3d, vector4d, matrix4x4
- Vannak ezeknek is tulajdonságai, de nem generálnak eseményt, ha változnak
  - vector4d.x nem generál, de vector4d igen

QML vezérlők - ablak

- ApplicationWindow – felső szintű ablak
  - Fontosabb tulajdonságai: title, color, width, height, visible
- MenuBar - menü
- ToolBar
  - ToolButton, stb.
- StatusBar – státusz sor



QML vezérlők - Label

- Szöveg kiírása
- Text-ből származik, ami mindent tud
  - Label annyiban más, hogy a operációs rendszer alapértékeit használja
- Text
  - Fontosabb tulajdonságok: text, font, color, elide, horizontalAlignment, verticalAlignment, wrapMode
  - textFormat
    - PlainText – azt írja ki, ami benne van
    - StyledText – HTML 3.2
    - RichText – HTML 4, lassú

QML vezérlők - Button

- Fontosabb tulajdonságok: text, isDefault, pressed
  - tooltip: ha a platform támogatja
  - style: stílus megadása
- Benyomva maradó gomb: checkable, checked
- Dropdown: menu
- Benyomásra kikapcsol más gombot: exclusiveGroup
  - Ez a RadioButtonnál lényeges
- Signal: clicked()

QML vezérlők – RadioButton

- Buttonnal is meg lehet csinálni
  - A platform style elvész

```
ColumnLayout {
    ExclusiveGroup { id: tabPositionGroup }
    RadioButton {
        text: "Top"
        checked: true
        exclusiveGroup: tabPositionGroup
    }
    RadioButton {
        text: "Bottom"
        exclusiveGroup: tabPositionGroup
    }
}
```

QML vezérlők

- CheckBox
  - text, pressed, checked, style
  - checkedState, partiallyCheckedEnabled, exclusiveGroup
- ProgressBar
  - value, maximumValue, minimumValue
  - indeterminate, orientation, style
- Slider
  - value, maximumValue, minimumValue, stepSize
  - tickmarksEnabled, style, orientation, pressed

QML vezérlők

- SpinBox (numeric up down)
  - value, maximumValue, minimumValue, stepSize
  - decimals, font, horizontalAlignment, style
  - prefix, suffix
  - editingFinished()
- Switch (toggle button)
  - checked, exclusiveGroup, pressed, style, clicked()
- Calendar
- BusyIndicator

QML vezérlők – TextField (input)

- text, placeholderText, inputMask, readOnly
- Jelszó támogatás: echoMode, displayText
- Szerkesztés
  - canPaste, canUndo, canRedo
  - length, maxLength
  - selectedText, selectionStart, selectionEnd
- Kinézet
  - font, textColor, style
  - horizontalAlignment, verticalAlignment
  - cursorPosition, cursorRectangle

QML vezérlők – TextField (input)

- Szótár támogatás – mobil eszközökön
- Több billentyűzet lehetséges
- Van automata korrekció is
- inputMethodHints
  - Qt.ImhSensitiveData – ne tárolja a szótárban
  - Qt.ImhNoAutoUppercase – első karakter ne legyen nagy
  - Qt.ImhPreferNumbers – más billentyűzet jöhet elő
  - Qt.ImhPreferUppercase, Qt.ImhPreferLowercase
  - Qt.ImhDate, Qt.ImhTime
  - Qt.ImhMultiLine – enterre ne záródjon be a billentyűzet
  - Qt.ImhEmailCharactersOnly, Qt.ImhUrlCharactersOnly
  - Qt.ImhDigitsOnly – csak számok
  - Qt.ImhFormattedNumbersOnly – számok, -, .

QML vezérlők – TextField (input)

- Validálás
  - validator: Int, Double, RegExp
  - editingFinished() signal enter után mindig jön
  - accepted() signal csak akkor jön, ha valid

```
TextField {
    validator: IntValidator {bottom: 1; top: 10;}
}
```

QML vezérlők – TextField (input)

- Függvények
  - copy(), paste(), undo(), redo()
  - selectAll(), selectWord(), select(start, end), deselect()
  - insert(pos, text), getText(start, end)

QML vezérlők - TextArea

- Többsoros szövegszerkesztő
- Sima szöveg, vagy Rich Text
- Tab és Enter
- Tördelés: NoWrap, WordWrap, WrapAnywhere, Wrap
- Syntax highlighting
  - textDocument tulajdonság
  - C++-ban lehet csak megírni
  - Van kiindulási alap: QSyntaxHighlighter

QML vezérlők – GroupBox

- Keret egy szöveggel
- Lehet CheckBox is a szöveg, letiltja a belső vezérlőket, ha nincs pipálva
- Flat stílusú is lehet
  - Minden eszközön más
  - Mobilon bal-jobb-alsó keret eltűnik (stílus függő)

Qt Quick

Alapok

QML Item

- Felület alapegysége
- Szinte minden ebből származik
- Hierarchia itt van implementálva
  - children: list<Item> - látszó gyerekek
  - resources: list<Object> - nem látszó gyerekek
  - data: list<Object> - összes gyerek, default property
- Ide kerül, amit minden vezérlő tud
  - Pl. enabled, parent

QML Item

- Megjelenítés
  - Opacity, antialiasing, clip, smooth
- Input fókusz
  - Focus, activeFocus, activeFocusOnTab
- Méret és pozíció
  - visible, x, y, z, width, height, rotation, scale, transform, transformOrigin
  - Nem 3D-s (nincs depth tulajdonsága)
  - De 3D térben van (x, y, z, és forgatni is lehet térben)

QML Item - anchors

- Automata pozícionálás
- Lehet horgonyozni
  - Nem csak szülőhöz, hanem testvérhez is
  - top, bottom, left, right, fill
  - topMargin, bottomMargin, leftMargin, rightMargin
  - A Margin hozzáadódik a kiszámolt értékhez

```
Item {
    Image {
        id: pic
    }
    Text {
        anchors.top: pic.bottom
        anchors.topMargin: 5
    }
}
```

QML Item - anchors

- A fill csak egy egyszerűsítés, hogy ne kelljen 4-et beállítani
- alignWhenCentered: fél pixelre nem teszi
  - Amikor kell: statikus elrendezés, éles kép
  - Amikor nem kell: Animált, folytonos mozgás
- BaseLine igazítás
  - baslineOffset: Hol van a baseline ebben a vezérlőben
  - achors.baseLine: A másik vezérlő baseline-ja
  - achors.baseLineOffset: Igazítás finomhangolása

QML Item - anchors

- Középre igazítás
  - anchors.centerIn: kényelmi beállítás a kettő helyett
  - anchors.horizontalCenter
  - anchors.verticalCenter
  - anchors.horizontalCenterOffset
  - anchors.verticalCenterOffset

QML Item

- Szintén Item-ben implementálták
  - Állapotok és átmenetek
  - Animációs alrendszer
  - Post process effectek

Saját vezérlő

- Létrehozunk egy .qml fájlt
- Nincs megkötés a gyökérelemre
  - Célszerű Itemet használni minimum (layout logikát tudja)
- A vezérlő neve a fájl/erőforrás neve lesz
  - Vagy ha külső fájl, akkor importálni kell

```
import QtQuick 2.0
Item {
    id: root
    Rectangle{
        anchors.fill: parent
        color: "white"
    }
}
```

Qt Quick

Layout

QML layout

- Column és Row
  - Egymás alá/mellé teszi a gyerekeit
  - spacing: megadható mennyi helyet hagyjanak ki
  - Nincs hátterük
- Grid
  - column tulajdonsága van csak (nincs row)
  - Előbb egymás mellé teszi őket, majd következő sorba
  - Ha üres cellára van szükség, akkor oda kell tenni valamit, ami nem rajzol

QML layout

- Flow
  - Hasonló Gridhez
    - De nincs oszlopszám megadva
    - Nincsenek igazítva egymás alatt az elemek
  - Addig teszi a vezérlőket egy sorba, amíg van hely, utána új sort nyit – hasonló a HTML layout logikához

QML layout

- GridLayout
  - Méretezi az elemeket (nem úgy, mint Grid)
  - XAML Grid, CSS grid szerű működése van
  - Automata cellakiosztás
    - Flow tulajdonság adja meg, hogy milyen sorrendben használja fel az elemeket, ha nincs megadva sor/oszlop
    - GridLayout.LeftToRight az alap, ekkor columns tulajdonság adja meg az oszlopok számát
    - GridLayout.TopToBottom esetén rows-t kell beállítani

QML layout

- GridLayout
  - Cella megadható kézzel is attached property-kel
    - Layout.row, Layout.column
    - Layout.rowSpan, Layout.columnSpan
  - Méret megkötéseket adhatunk
    - Min, max, prefered – width, height
  - Kitöltés: Layout.fillWidth és Layout.fillHeight
  - Igazítás cellán belül: Layout.alignment

QML layout

- RowLayout és ColumnLayout
  - Az algoritmus ugyanaz, mint a GridLayout
  - Picit kényelmesebb megadni, ha csak egy sor/oszlop van

QML layout

- Ablakhoz kötés
  - Ha az ablak méretezhető (pl. PC)
  - Tipikusan egy layout konténer az ApplicationWindow alatti első elem
    - Állítsuk be, hogy töltse ki az ablakot: achors.fill: parent
  - Az ablakon
    - width: layout.implicitWidth
    - height: layout.implicitHeight
    - minimumWidth: layout.Layout.minimumWidth
    - minimumHeight: layout.Layout.minimumHeight
    - Ha van max a layouton, akkor azokat is be kell állítani

Qt Quick

Események

Események – Signal

- Komponens definiálja
  - Sajátot is létrehozhatunk
  - Lehet C++ és QML is
- Fel lehet rá iratkozni
  - Bárhonnan, ha az esemény láthatósága megengedi
  - A feliratkozás helyén nem látszik, hogy mik a paraméterei, de a deklarálásnál igen (és a doksi is megadja a beépítettekhez)

Események – Signal Handler

- Az objektumban deklarálni kell egy Signal Handlert
  - on<Signal> a neve, ahol <Signal> az esemény neve (nagybetűvel)
  - Értéke a JS kód
  - Kívülről és C++-ból is fel lehet iratkozni rá, de ez a szintaktika a legkényelmesebb

```
Button{
    text: "hello"
    onClicked : { rect.color = "blue"}
}
```

Események – Property Change

- Minden tulajdonsághoz automatikusan tartozik egy Property Change Signal
  - Az alaptípusok tulajdonságaihoz nem
  - A doksiban ezek külön nincsenek benne, de mindenhez van
  - Változás után hívódik meg, az értékét simán a tulajdonság lekérdezésével tudhatjuk meg
- Feliratkozni az on<Property>Changed deklarációval lehet
  - <Property> a tulajdonság neve (nagybetűvel)

Események – Attached Signal Handler

- Más forrásból is kaphatunk eseményt, ha az definiál Attached Signalt
- Példa a Component objektum, aminek a completed eseménye mindenhonnan elérhető

```
Rectangle{
    width: 200; height: 200
    Component.onCompleted : {
        console.log("The rectangle's color is", color)
    }
}
```

Események – Timer

- Időzítő
- Triggered signal
- Animációt nem ezzel csinálunk

```
Timer{
    interval: 500; running: true; repeat: true
    onTriggered : time.text = Date().toString()
}
Text{ id: time }
```

Események – saját Signal

- Saját vezérlőben signal kulcsszó
- Eseményt generálni a meghívásával lehet, mintha függvény lenne (nem az)

```
Item {
    id: root
    signal clicked ( )
    Rectangle{
        anchors.fill: parent
        color: "white"
    }
    MouseArea{ anchors.fill: parent; onClicked: root.clicked()}
}
```

Események – saját Signal

- Feliratkozni a szokásos módon lehet
- Tegyük fel, hogy a saját vezérlő típusneve ClickRect

```
ClickRect {
    onClicked: console.log("Hello")
}
```

Események – connect()

```
Item {
    id: root
    signal clicked ( )
    Rectangle {
        anchors.fill: parent
        color: "white"
        Component.onCompleted: {
            root.clicked.connect(clickThisToo)
        }
    }
    function clickThisToo(){ }
    MouseArea{ anchors.fill: parent; onClicked: root.clicked() }
}
```

- Több függvényt is rá lehet kötni egy signalra
  - A Handler mellett
  - Másik signalt is

Kérdések?

Multiplatform szoftverfejlesztés

Qt Quick 2

Qt Quick

JavaScript

QML JavaScript

- Jelenlegi JavaScript verzió: ES2016 (Qt 6.5-ben)
- JS kódot lehet írni
  - Property Binding
  - Eseménykezelő (Signal handler)
  - Bármilyen objektumhoz új függvényt
  - Importálható .js fájl
    - Ezek a változók és függvények felhasználhatók bárhol

QML JavaScript

- Nincs window, document, …
- Van Qt globális objektum
  - Számos Qt specifikus függvény
    - binding, exit, …
  - Segítő függvények
  - application object
    - Alaptulajdonságok mint name, version, …
  - platform (android, ios, windows, …)

QML import

- import <Module> <Version> as <Name>
  - <Module> lehet
    - Telepített modul, pl. QtQuick (az alapkontrollokhoz)
    - JS fájl
  - <Version> meg kell egyezzen, ha megadjuk
- Ha nincs név, akkor globális névtérbe kerül
  - import QtQuick 2.0

JavaScript - Property Binding

```
Rectangle {
    id: colorbutton
    width: 200; height: 80;
    color: mousearea.pressed ? "steelblue" : "lightsteelblue"
    MouseArea {
        id: mousearea
        anchors.fill: parent
    }
}
```

JavaScript - Eseménykezelő

```
Rectangle {
    width : 200; height: 80; color: "lightsteelblue"
    MouseArea {
        anchors.fill : parent
        onPressed : {label.text = "I am Pressed!"}
    }
    Text {
        id: label
        anchors.centerIn : parent
        text : "Press Me!"
    }
}
```

JavaScript – saját függvény

```
Item{
    function pi2() {
        return Math.PI * Math.PI;
    }

    MouseArea{
        anchors.fill: parent
        onClicked : console.log(pi2())
    }
}
```

JavaScript – külső .js fájl

- import kulcsszó
- as libname
  - Így lehet hivatkozni rá

```
import "hello.js" as MyFunctions
Item{
    MouseArea{
        anchors.fill: parent
        onClicked : console.log(MyFunctions.hello())
    }
}
```

JS, vagy C++

- Betöltés után szinte mindent meg lehet írni JS-ben
- És C++-ban is
  - QObject-ből származtatunk – ez látszik QML-ben
- Ezek után el is felejthetjük a C++-t?
- Válasz 1
  - Nem mindent lehet megírni JS-ben
  - JS lassabb, de ez vajon számít-e
  - Ha minden JS lenne, akkor miért nem HTML+JS?
    - QT többet tud, mint HTML
    - Van lehetőségünk átlépni C++-ba, ha szükséges

JS, vagy C++

- JS-ben célszerű megírni, ami a felülethez tartozik
  - Ha nem túl bonyolult
    - Nincsenek olyan szintű eszközök, mint C++-hoz
  - Ha nem számít a sebesség
    - C++ hozzáállás: mindig számít
    - Finomítunk: Ha nem kritikus a sebesség

Qt Quick

Alaposztályok

QtObject

- Egyszerű objektum
  - Csak egy neve van
  - C++ kódból név alapján meg lehet találni
  - Csoportosíthatunk mezőket (jobb oldal)
- Hierarchia gyökere
  - Szinte minden ebből származik
  - Item ebből származik

```
QtObject{
    id: attributes
    property string name
    property int size
    property variant attributes
}

Text{ text: attributes.name }
```

Component – sablon

```
Component{
    id: redSquare
    Rectangle{
        color: "red"
        width : 10
        height : 10
    }
}

Loader{ sourceComponent: redSquare; x: 20 } // QML-ben

redSquare.createObject(parent, {"x":20}); // JS függvényben
```

- Újrafelhasználható
- Létrehozza minden példányhoz
- ItemTemplate (XAML-ben) hasonló

Component

- Betöltés állapota
  - progress: 0..1
    - 1-nél nincs kész, status==Component.Ready-nél van kész
  - status: Null, Ready, Loading, Error
  - url: a qml fájl, amiben van
- Minden így töltődik be, az első qml fájlunk is
- Component.onCompleted
  - attached signal – mindennek van
  - (Lehet használni globális JS függvényt is – saját dolgait tudja inicializálni)
- Component.onDestruction()

Qt Quick

Adatkötés

QML Item – Adatkötés

- Minden köthető mindenhez
  - Alaptípusok tulajdonságai nem triggerelnek
- Tetszőleges JS kifejezés, vagy teljes kód írható be
  - Kifejezés esetén nem kell return, sem kapcsos zárójel

```
Rectangle {
        width: 100
        height: parent.height
        color: "blue"
}
```

QML Item – Adatkötés

- Property Binding csak egy kényelmi szintaktika
  - Valóságban egy binding objektum jön létre
    - Tetszőleges JS függvényt átadhatunk
- Qt.binding: JS szintaktika
- Függőségi gráf

```
Rectangle{
    width: parent.width
    Component.onCompleted: {
        height = Qt.binding(function() { return parent.height });
    }
}
```

QML Item – Adatkötés

- Adatkötés csak akkor történik, ha a fenti két szintaktikát használjuk
- Például itt space-re nem
  - Helyette simán kiszámolja az értéket, beleírja, majd nem változik többé

```
Rectangle {
    width: 100
    height: width * 2
    Keys.onSpacePressed: {
        height = width * 3
    }
}
```

QML Item – Adatkötés

- Két irányú adatkötés – nincs külön támogatás
  - Binding-loop kialakulhat (itt nem, mert property csak akkor süti el a changed signalt, ha tényleg változik az értéke)

```
Button {
    id: button1
    property int count: 0
    onClicked: count += 1
    text: count
}
Button {
    id: button2
    property int count: 0
    onClicked: count += 1
    text: count
}
```

```
Binding {
        target: button1
        property: 'count'
        value: button2.count
}
Binding {
    target: button2
    property: 'count'
    value: button1.count
}
```

Qt Quick

Állapotok

QML Item – Állapotok

- Aktuális állapot: state
- Összes állapot: states: list<State>
- Váltani a state=<state neve> módon lehet
- Mindenképpen van alapállapot
  - A neve üres string
  - Mindent visszaállít a kezdeti állapotba
  - Definiálni is lehet saját alapállapotot

QML Item – Állapotok

- State QML elem
  - name: string
  - changes: list<Change>
    - PropertyChanges, AnchorChange, ParentChange, …
  - extend: string
    - Átveszi a megnevezett state összes változását
    - Felül lehet írni, hozzá lehet adni
  - when: bool
    - JS kifejezést adunk meg, ami kiértékelődik
    - Ha igaz, akkor automatikusan állapotot vált
    - Ha több is igaz lesz egyszerre, akkor az elsőre vált

QML Item - Állapotok

```
Rectangle {
    id: rect
    width: 100; height: 100
    color: "black"
    MouseArea {
        id: mouseArea
        anchors.fill: parent
        onClicked: rect.state == 'clicked' ? rect.state = '' : rect.state = 'clicked';
    }
    states: [
        State {
            name: "clicked"
            when: width < 100
            PropertyChanges { target: rect; color: "red" }
        }
    ]
}
```

QML Item - Állapotok

- PropertyChanges
  - Leggyakrabban használt
  - Több tulajdonságot is állíthat
    - PropertyChanges { target: rect; color: "blue"; height: 5 }
    - undefined visszaállítja alapértékre (nem kezdeti értékre)
  - Lehet bindingot létrehozni
    - PropertyChanges { target: rect; height: parent.width }
  - Vagy az éppen aktuális értékét beírni binding nélkül
    - PropertyChanges { target: rect; explicit: true; height: parent.width }

Qt Quick

Átmenetek

QML Item - Átmenetek

- Ha nem azonnali váltás kell – sosem az kell
- transitions: list<Transition>
- Átmenet lehet
  - Állapotok között
    - Transition { from: "s1"; to: "s2"; ColorAnimation { … }}
    - Minden állapotban – from: "*", ez az alapértéke
  - Tulajdonság változásra – Behavior
    - Ez kicsi prioritású – ha van állapot animáció, akkor az érvényesül ütközés esetén
    - Behavior on width {NumberAnimation{ duration: 10 }}

QML Item - Átmenetek

- Minden animációban
  - Tulajdonságok: loops, paused, running
  - Események: started(), stopped()
  - Függvények: start(), stop(), restart(), pause(), resume(), complete()
- Animációk hierarchiába szervezése
  - Alapban párhuzamosan futnak a Transition-be felvett animációk, ha több van
  - SequentialAnimation
  - ParallelAnimation

QML Item - Átmenetek

- Nem animáló animációk
  - PropertyAction – nem animál, azonnal átállít
  - PauseAnimation – vár
  - ScriptAction – kód futtatása
- PropertyAnimation
  - NumberAnimation – lineáris
    - SpringAnimation – rugó
    - SmoothedAnimation – Easing function
  - ColorAnimation
  - RotationAnimation
  - Vector3dAnimation

QML Item - Átmenetek

```
Rectangle {
    id: rect; width: 100; height: 100; color: "red"
    MouseArea {
        id: mouseArea; anchors.fill: parent
    }
    states: State {
        name: "moved"; when: mouseArea.pressed
        PropertyChanges { target: rect; x: 50; y: 50 }
    }
    transitions: Transition {
        NumberAnimation { properties: "x,y"; duration: 300 }
    }
}
```

QML Item - Átmenetek

- Easing
  - PropertyAnimation tulajdonsága
  - Type
    - Egyenlet: Linear, Quad (x^2), Cubic (x^3), Quart (x^4), Quint (x^5), Sine(sin), Expo (2^x), Circ ($sqrt(1-x^2)$), Elastic, Back, Bounce, Bezier
    - Belépési/kilépési pont: In, Out, InOut, OutIn
  - Paraméterek: amplitude, overshoot, period

```
NumberAnimation {
    to: 100; easing.type: Easing.OutBounce; duration: 1000
}
```

QML Item - Átmenetek

- Speciális animációk
  - AnchorAnimation – változik az objektum
  - ParentAnimation – változik a szülő
  - PathAnimation – x,y animáció
- on szintaktika – <Animation> on <property>
  - Nem kell megadni: target, properties
    - PropertyAnimation on x { to: 100 }
  - Automatikusan indul, ha nem átmenetben van

Qt Quick

Style

QML Style

- Minden vezérlőnek testre szabható a kinézete
  - Amit mi írunk, annak csak akkor, ha megírjuk
- style tulajdonságnak kell megadni a saját stílust
- Minden vezérlőnek mást kell megadni
  - Pl. Button-nak ButtonStyle
  - Ezek egyszerű struktúrák, összefogják a részeket, amit megadhatunk
  - Nem kell minden részt felüldefiniálni

QML Style

```
Button{
    text: "Push"; width: 100; height: 30
    style : ButtonStyle{
        background: Rectangle{
            border.width : control.activeFocus ? 2 : 1
            border.color : "red"
            radius : 10
            gradient : Gradient{
                GradientStop{ position: 0; color: "white" }
                GradientStop{ position: 1; color: "blue" }
            }
        }
    }
}
```

QML Style

- A Style struktúrában vannak Component-ek az egyes részekhez, és egyéb adatok
- Például
  - ButtonStyle: background, label
  - CheckBox: background, label, indicator, spacing: int
  - Slider: groove, handle, panel, tickmarks

Qt Quick

Magas szintű szolgáltatások

Dialógus ablakok

- ColorDialog
- FileDialog
- FontDialog
- MessageDialog
- Dialog
  - Általános célú ablak, platform gombokkal
  - OK, Cancel, Open, Save, Close, Apply…

Dialógus ablakok

```
Dialog {
    id: dateDialog
    visible: true
    title: "Choose a date"
    standardButtons: StandardButton.Save | StandardButton.Cancel
    onAccepted: console.log("Saving the date " +
        calendar.selectedDate.toLocaleDateString())
    Calendar {
        id: calendar
        onDoubleClicked: dateDialog.click(StandardButton.Save)
    }
}
```

HTML megjelenítése

- Text vezérlő szöveget tud
- WebView teljes weboldalt
  - Url-t is meg lehet adni, ahonnan letölti
  - Böngészőt lehet írni vele
  - Chromium 108 motor

Szenzorok

- Szokásos: Accelometer, Altimeter, AbmientLightSensor, Compass, Gyroscope…
- Speciális eszközökben vannak: AmbientTemperatureSensor, DistanceSensor…
- Összesen 17 szenzor
  - Vannak átfedések (Pl. Light és AmbientLight)
- Nem csak mobil eszközökre van kitalálva, de azokkal is jól megy
  - Robotok vezérlése

Multimédia

- MediaPlayer
  - Audio
  - Video
- Camera
  - Az összes beállítása ki van vezetve (vaku, fókusz is)
- Radio
- Torch

```
Torch {
    power: 75          // 75% of full power
    enabled: true      // On
}
```

ShaderEffect

- Vertex és pixel shader
- Rectangle az objektum, ebben lehet más is
- Alapban 4 vertex
  - mesh megadásával bármennyi lehet (torzításhoz)
- cullMode állítható (backface culling)
- blending: alpha blending bekapcsolásához
  - Csak a source alpha és az összeadást tudja

ShaderEffect

- Vertex shader
  - Standard bemenő paraméterek
    - Állandó minden vertexre (uniform): qt_Matrix, qt_Opacity
    - Vertexenként (attribute): qt_Vertex, qt_MultiTexCoord0
  - Ezen kívül fel lehet venni bármi mást is, amit átadunk
  - Kimenetet mi definiáljuk
    - varying kulcsszó, ez megy a pixel shadernek

ShaderEffect

```
vertexShader: "
    uniform highp mat4 qt_Matrix;
    attribute highp vec4 qt_Vertex;
    attribute highp vec2 qt_MultiTexCoord0;
    varying highp vec2 coord;
    void main() {
        coord = qt_MultiTexCoord0;
        gl_Position = qt_Matrix * qt_Vertex;
    }"
```

ShaderEffect

- Pixel shader
  - Vertex shader kimenete jön interpolálva
  - Sampler a mintavételezéshez
    - Image-re kötve
  - Állandókat felvehetünk (uniform)

```
fragmentShader: "
    varying highp vec2 coord;
    uniform sampler2D src;
    void main() {
        gl_FragColor = texture2D(src, coord);
    }"
```

Kérdések?

Multiplatform szoftverfejlesztés

TypeScript

JavaScript

JavaScript támogatás

- Van szabvány, és megkésve követjük is
- Fordítót használunk
  - Babel
  - TypeScript
  - Tipikusan ES6-ra fordítunk
- Ez meglepően jól működik
  - Ellentétben a HTML és CSS problémákkal

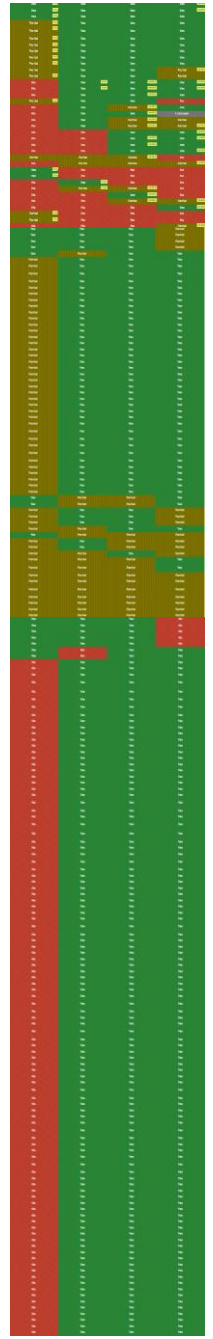
JavaScript támogatás

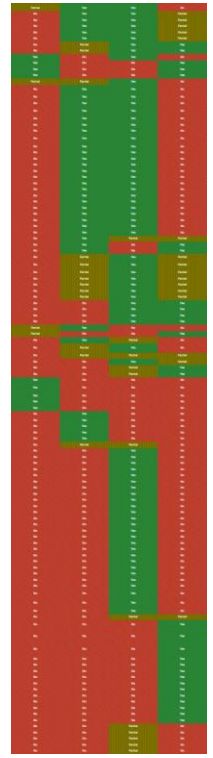
- Polyfillt használunk a nem implementált funkcionalitáshoz

```
if ( !String.prototype.startsWith )
{
    Object.defineProperty( String.prototype, 'startsWith', {
        value: function ( search, rawPos )
        {
            var pos = rawPos > 0 ? rawPos | 0 : 0;
            return this.substring( pos, pos + search.length ) === search;
        }
    } );
}
```

Jelenlegi támogatás

- Report: caniuse 2022 április
  - Csak JS
- Oszlopok
  - IE 11
  - Firefox
  - Chrome
  - iOS Safari
- Az jobb alsó részben JS API-k vannak
  - Web Bluetooth, WebUSB, stb.
- Firefox és Chrome (Edge) elöl





Gyengén típusos

- Vannak típusok
  - number, string, boolean, Object, Symbol, function, bigint, null, undefined
  - Csak futásidőben kerülnek ellenőrzésre
  - A runtime megpróbál mindenhol konvertálni
    - Csak végső esetben dob hibát
- De nem kell/lehet kiírni őket
  - Rövid, jól átlátható kódot eredményez
    - Amíg kicsi a program

Gyengén típusos

- Közepes és nagy szoftvereknél probléma
  - Nincs fordításidejű ellenőrzés, sokkal többet kell tesztelni
  - Problémás a tooling
    - Kódkiegészítés: nehéz javaslatokat adni a fejlesztőnek, hogy mit tud beírni
    - Ellenőrzés: kevesebb hibát lehet kielemezni
  - Nem látni a kódból, hogyan kell használni
    - Ez megoldható dokumentációval – /**…*/ széleskörben támogatott
- Mi csak a közepes és nagy szoftverekre koncentrálunk

TypeScript

Általában

Mi a TypeScript?

- Típusos JavaScript, a típusok opcionálisak
  - Minden JS egyben TS is
  - Amint beírunk típust valahova, az már csak TS
- Típust a változó neve után írjuk

```
function A( a, b ){ // .js, vagy .ts fájl is lehet
{
  return a + b;
}
function A( a: number, b: number ) // csak .ts fájlban lehet
{
  return a + b;
}
```

Tudásban TypeScript=JavaScript

- TS nem tud többet
- Ha kivesszük a típusokat, akkor JS-t kapunk
  - Ugyanúgy fog viselkedni a kód futásidőben
  - typeof és társai is csak a JS szintet hozzák
- Az egyetlen különbség, hogy van egy fordítási lépés
  - Ez nagyon fontos, akkor is használjuk, ha JS-t írunk (babeljs segítségével)
  - A cél, hogy átkódoljuk az új szabvány szerint megírt kódot a célplatformra (pl. ES5 IE11 miatt)
  - TS esetében ez a lépés kiveszi a típusokat is

Miért fontosak a típusok

- Kezdetleges dokumentáció
  - Sokszor lehet következtetni, hogy mit csinál
    - Névből – JS/TS
    - Paraméterek nevéből – JS/TS
    - Paraméterek típusából – csak TS
- Tooling
  - Kódkiegészítés
    - Kontextusfüggő: típus korlátozza a listát
  - Linter
- Fordításidejű kódellenőrzés
  - Hasonló linterhez, de sokkal hatékonyabb

Miért fontosak a típusok

- Tesztelés segítése
  - Típusok esetén a tesztelés költsége jelentősen csökken (akár felére)
    - A hibák jelentős része nem kerül bele a kódba
- Összességében a típusok csökkentik a költséget

OO paradigma

- Fontos: a típusok használata nem segít az objektum orientáltságon
  - Függetlenek egymástól
- JS támogatja az OO irányelveket
  - Vannak osztályok, egységbe zárás (encapsulation)
  - Belső működés elrejtése – absztrakció
    - Private (#) csak ES2019-től, TS-ben volt/van
  - Öröklés
  - Polimorfizmus – ez nincs JS-ben, sem TS-ben
    - Egy függvény viszont több típussal is tud működni
    - Az eredeti célt el tudja érni

OOP TS-ben

- Támogatottak
  - Osztályok
  - Interfészek (explicit- és implicit megvalósítás)
  - Absztrakt osztályok
  - Öröklés
  - Láthatósági módosítók (public, private, protected)
  - Osztályszintű változók és függvények
  - Enum típusok, string literálok, unió- és metszettípusok
- Nem támogatottak
  - Valódi metódus overloading
  - Valódi többszörös öröklés
  - Típusonként több konstruktor/azonos nevű függvény

Osztályok és öröklés

- A legtöbb keretrendszer nem osztály alapú
  - Régen nem voltak osztályok
  - Kezdő programozóknak egy akadály
  - this probléma nem segít
  - Komponens alapú fejlesztés
    - Kompozíciót használunk, nem öröklést
  - Egységbe zárást a komponens valósítja meg
    - Ami vagy osztály, vagy nem
- TS-től a típusosságot kérjük
  - Osztályokkal külön nem foglalkozunk
  - Ettől még sok kód osztályt fog használni

TypeScript

Típusok

Alaptípusok

- Az alaptípusok a JS alaptípusok, plusz

```
let a: number[]; // tömb
let b: [ number, string ]; // tuple
enum Color { Red, Green, Blue };
let c: Color; // enum
let d: any; // nincs ellenőrzés
let e: "red" | "green" | "blue"; // string literal
```

Összetett típusok

- Unió: string | number
  - Vagy egyik, vagy másik
  - Nagyon sokat használjuk
    - Mert azonos neve nem lehet függvényeknek
      - Például polimorfizmus megoldására
    - Nem a fordító dönti el, hogy melyiket kell hívni
      - Függvényen belül if-elünk
- Metszet: ObjA & ObjB
  - Minden A-ból és B-ből

Függvények, röviden

- Default és opcionális paraméterek

```
function fd( a: string = "hello", b?: string )
{
}
```

- undefined-ot kapunk, ha nincs megadva
  - Vagy kézzel azt adtak át
  - Tehát a default paraméter is lehet undefined
- Azonos a működés JS-sel
  - Nem fordul le, ha nem adunk meg egy kötelező paramétert – minden az, ami nem opcionális/default

Osztályok, röviden

- Konstruktorban tulajdonság

```
class C
{
  constructor( public name: string ) { }
}
```

- public, protected, private működik
  - De csak fordítás időben
  - #field is működik, ez futásidőben is
- readonly, static, abstract kulcsszavak
- Accessors: get és set

this

- TS nem oldja meg teljesen a this problémát
  - De segít rajta
- Nekünk kell megoldani
  - Minden callback-nél használjunk arrow function szintaktikát

```
setInterval( () =>
{
  // itt a this azonos a külsővel
}, 1000 );
```

Type guards

- Fordító követi a kód logikáját

```
function toS( x: string | number )
{
  if ( typeof x === "string" )
    return x;
  else
    return x.toFixed();
}
```

- Működik instanceof esetén is

Paraméteres típusok – Generics

- Használhatunk előre nem ismert típusokat

```
function concat<T>( a: T, b: T )
{
  return a.toString() + b.toString();
}
concat( 1, "2" );  // Error
```

- Osztályok és függvények is
- Több paraméter is lehetséges
- A fordító látja, hogy mivel hívjuk, nem kell megadni – mint C#, vagy C++

Paraméteres típusok – Generics

- Kényszerekkel

```
interface HasLength
{
  length: number;
}
function getTotalLength<T extends HasLength>( a: T, b: T )
{
  return a.length + b.length;
}
```

Interfészek – interface kulcsszó

- Objektum tulajdonság
- Objektum függvény
- Objektum konstruktor függvény
- Függvény
- Indexer

- Ezeket mind meg lehet adni interface nélkül is

```
interface HasLength<T>
{
  new(): T;
  length: number;
  getLength(): number;
}
interface Indexable
{
  [ key: string ]: string;
}
interface Action<T>
{
  ( param1: T );
}
var x: Action<string> =
  s => console.log( s );
```

Struktúrálisan típusos

- Két változó akkor azonos típusú, ha struktúrálisan azonos a típusuk
- Például

```
type SoN = string | number;
function FA()
{
  let a: SoN = 1;
  let b: number | string = a;
}
```

- A típus neve nem számít

Struktúrálisan típusos

- Ez igaz interfészekre és osztályokra is

```
interface IA
{
  a: string;
}
interface IB
{
  a: string;
}
```

- És minden más típusra
  - Ha kompatibilis, akkor fordul

Struktúrálisan típusos

- Függvények is követik a kompatibilitás elvét
- Még trükkös esetekben is

```
let x = ( a: string ) => { };
let y = ( a: string, b: string ) => { };
y = x; // OK
x = y; // Error
```

  - JS-ben mindenhol átadhatok kevesebb paraméterrel rendelkező függvényt

Modulok

És egyéb nyelvi elemek

Névterek (ritkán használt)

- Egy fordítási egységen belül

```
namespace NS
{
   export class C
   {
   }
}
```

- Használata: /// <reference path="x.ts"/>
- Kód darabolása a cél
  - Nagyon hasonló az osztály egységbe záráshoz

Modulok

```
module M
{
  export class C{}
}
```

- Ezt csak modul betöltővel lehet használni
  - import { MyClass } from 'my-class';
- Fordításnál állíthatjuk, hogy milyen kódot generáljon
  - CommonJS (Node.js)
  - RequireJS (AMD)
  - ...

Típusdeklarációs fájlok .d.ts

- Külső könyvtárakhoz van típusokat leíró fájl
- Fel kell tenni
  - npm i -D @types/jquery
  - Vagy letölteni kézzel
- Majd megmondani a fordítónak
  - /// <reference path="jquery/jquery.d.ts" />
  - Ezt a .ts fájlunkban, ahol használjuk
- Óriási gyűjtemény
  - https://definitelytyped.org/

Típusdeklarációs fájlok .d.ts

- Ezek sima .ts fájlok
  - De tipikusan nincs bennük olyan kód, ami benne marad fordítás után
  - Csak típusok leírása van bennük
- Tipikusan: type, declare, interface

```
type StringOrNumber = string | number;
declare class A
{
  private name: string;
}
```

Típusdeklarációs fájlok .d.ts

- Mi magunk is írhatunk .d.ts fájlt
- Célok
  - Könyvtárat írunk
    - Más fel fogja használni
    - JS-ként adjuk át, így a típusok eltűnnek belőle
  - Más nyelven készítettük a szervert
    - C# szerver típusait célszerű deklarálni .d.ts fájlban
    - Lehet automatikus folyamat
    - Protocol Buffer megoldás .d.ts fájlt is generálhat

Aszinkron programozás

async, await

Promise

- Egyre több API használ Promise-t
  - Ez egy osztály, ami támogatja
    - Több feliratkozót
    - Hívás-válasz mintát – mint egy függvényhívás
      - De például ismétlődő eseményekre nem alkalmas – nem egy esemény
    - Egységes hibakezelést
      - Van benne try-catch, nem kell kézzel beletenni
    - Láncolást: .then(valami).then(más)
- Felhasználása .then(callback)
  - Vagy .catch(callback)

Promise

- A sima callbackhez képest kényelmesebb
  - Mindennek azonos az interfésze
    - Nem kell tudni, hogy melyik paraméter is a callback
  - Azonos a hibakezelés is
- Nem tökéletes
  - A kód még mindig callbackekben van
- ES6-tól van
  - ES5-re fordításkor belefordítja a kódját

Promise – delay

- Egy példa a setTimeout Promise-ra alakítására

```
function delay( ms: number )
{
  return new Promise( ( resolve, reject ) =>
    setTimeout( resolve, ms ) );
}
```

- Visszadunk egy Promise-t
- Elindítunk egy timert
- Amikor lejár, meghívjuk a resolve-ot
  - Ami meghív minden .then-t, ami rá van téve

async, await

- Ha egy függvény Promise-t ad vissza
  - Beírhatunk elé egy awaitet
  - Feltéve, hogy async függvényben vagyunk

```
async function fa()
{
  await delay( 500 );
  console.log( "hello" );
}
```

- Az await utáni kód a .then-be kerül fordításkor
  - Minden await ponton elvágja a kódot a fordító

async

- Promise-t ad vissza (csak nem látszik)
- Akkor hívja meg a resolve-ot, amikor az utolsó sor is lefutott
- Meghívja a reject-et, ha kivétel keletkezik

```
async function fa()
{
  await delay( 500 );
  console.log( "hello" );
}
```

```
function fa()
{
  return new Promise( ( resolve, reject ) =>
  {
    delay( 500 ).then( () =>
    {
      console.log( "hello" );
      resolve();
    } );
  } );
}
```

Szálak

- JS-ben csak egy szál van, azon megy az egész
  - Ha szinkron minden, akkor megszakítás nélkül
  - Ha olyan API-t hívunk, ami később hív vissza, akkor kiütemezi a szálat
    - És folytatja ugyanazon a szálon, amikor visszatér
- Más nyelvekben (pl. C#) kontextus van
  - Azonos kontextusban kapjuk vissza a vezérlést
  - A fő szál, ami a UI-t futtatja egy külön kontextusban van egyedül
    - A fő szálon való aszinkron programozás egyszálú – nem kell szinkronizálni
  - Háttérszálak – például szerver kódban – egy kontextusban vannak együtt
    - Alapban többszálú, az aszinkron programozás nem változtat ezen

Kérdések?

Multiplatform szoftverfejlesztés

Web alapú alkalmazások

Web alapú alkalmazás

- Webes technológiákat használ
  - HTML, CSS, JS
- Asztali és/vagy mobil alkalmazások
  - Nem weboldalak, nem web alkalmazások
- Multiplatform
  - A lehető legtöbb eszközön menjen
  - 5% felett: Windows, macOS, iOS, Android
  - 5% alatt: Linux, ChromeOS, …

Web alapú alkalmazás

- Alkalmazásként működik
  - Új néz ki, mint egy alkalmazás – design
  - Új viselkedik, mint egy alkalmazás
    - Nem linkel ki, …
    - Együttműködik a többi alkalmazással
    - OS integráció (share, drag&drop, …)

Webes technológia

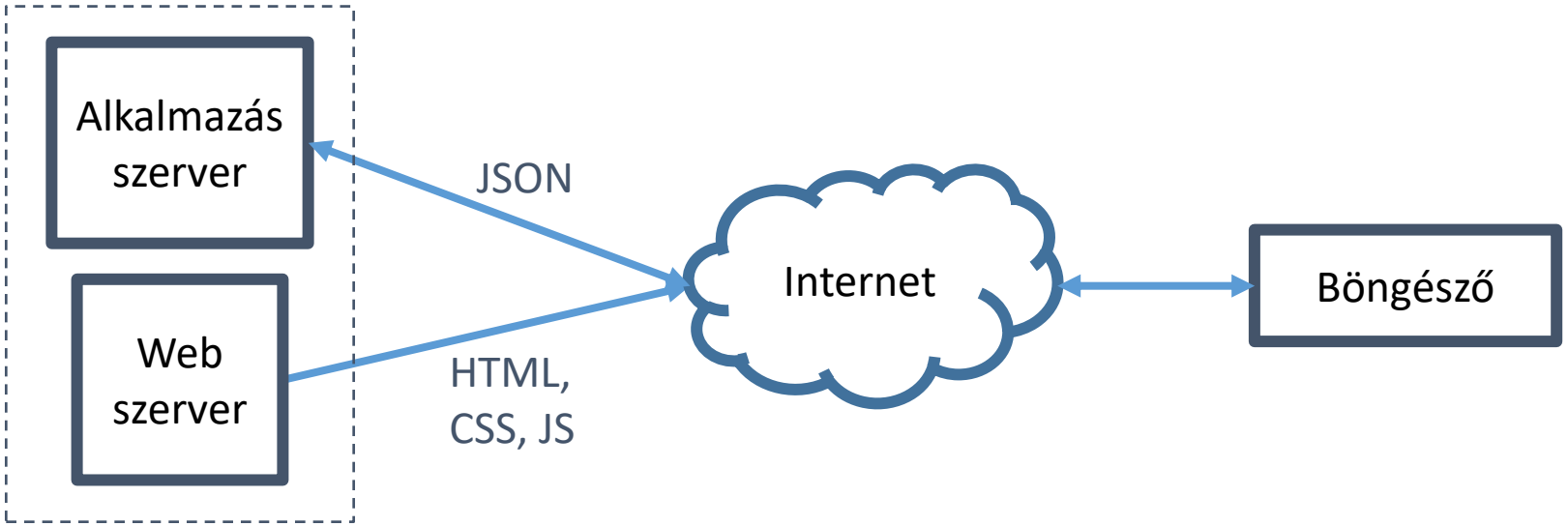
- Webes technológiák használata felhasználói felület készítésére
  - HTML elemek + CSS
    - Tartalomfogyasztó alkalmazások (Twitter, …)
      - Ezek lehetnek sima webalkalmazások is
    - Utility és productivity alkalmazások
    - Egyszerűbb játékok
  - Canvas
    - Tipikusan játékok (legnagyobb bevétel mobilon)
    - Esetleg multimédia alkalmazások
- Hogyan lehet hatékonyan fejleszteni HTML-en és CSS-en alapuló alkalmazást?
  - Canvas-ra visszatérünk

Kliens, vagy szerver oldali keretrendszer

Architektúra

- Szerver oldali keretrendszer

- Kliens oldali keretrendszer



Szerver oldal

- Példa: ASP.NET, PHP, JSP
- Minden kérést a webszerver kezel
- Bemenet: HTTP kérés
- Kimenet: HTTP válaszban HTML [+JS+CSS]
  - Teljes oldal, vagy csak része
  - Ha nem teljes oldal, akkor kliensen kell JS a feldolgozáshoz
- Működik JS nélkül is
  - Ha van JS, akkor lehet validálást, felhasználót segítő apróságokat végezni

Kliens oldali keretrendszer

- Példa: Angular, React, Vue
- Webszerver statikus oldalakat ad vissza
- Alkalmazás szerver JSON-t ad vissza
  - Kliens készíti el a HTML oldalt az adatokból
- Nem működik JS nélkül – ma már nem gond
- Szervert nem terheli a render

Kliens oldal – előnyök

- Jobban skálázódik
  - Szerver oldalról egy nagy tétel eltűnik (render)
- Gyorsabb
  - Kliens nem felejt navigálások között
    - Nem kell újra letölteni az oldal részeit
  - Maga a lekérés is kisebb
    - JSON kisebb, mint az abból készített HTML
- Gyorsabbnak tűnik
  - Kliens oldalon lehet animációkkal, stb. úgy tenni, mintha az adat már itt is lenne

Kliens oldal – előnyök

- Hozzáfér csak kliens oldalon lévő szolgáltatásokhoz (share target, notification)
- Aktív kapcsolatot tarthat fenn a szerverrel
  - Frissítheti magát
- Több szerverrel is kommunikálhat
  - Nem kell minden kommunikációnak átmenni saját szerveren
- PWA (Progressive Web Apps)
- A hibrid megoldások (szerver render, de van kliens oldali része is) ezek egy részét tudják

Botok (crawler) – kliens oldal

- Indexelő crawlerek
  - Google, Bing, DuckDuckGo, Yandex, …
  - Linkeket követik
  - Legtöbb nem hajt végre JS-t
  - Néhány igen
    - Google és Bing még XHR-t is végrehajt
    - Websocketet egyik sem tud
  - Komplex oldalon hibáznak
    - Lehet/kell segítséget adni, mindegyiknek van leírása
    - Sajnos több száz crawler van

Botok (share link) – kliens oldal

- Share link botok
  - Facebook, Twitter, Skype, Viber, Telegram, …
  - Egy link miatt elemzik az oldalt, keresnek
    - Képet
    - Címet
    - Leírást
  - Nem keresnek linkeket, nem követik őket
  - Kliens oldali kódot általában nem hajtanak végre
  - Bonyolult oldal esetén hibáznak
    - Segíteni kell nekik
    - meta tagek formájában (pl. Open Graph tagek)

Botok – szerver oldal

- Szerver oldalon renderelt oldal hátrányai nem problémák botok esetén
  - Nem kell validálni
  - Nem kell interakció: animáció, egyéb hatások
- Minden botnak más a fontos
  - Például képméret függő, hogy facebook hogyan teszi ki a linket
  - Több különböző oldalt/variációt kell gyártani
    - Akár 3-4
  - Szerver oldali render kell

Botok – amikor nem számít

- Ha az oldal nem indexelhető
  - Intranet site
    - Csak a site felhasználóinak érdekes adat
  - Védett tartalom
    - Bizonyos felhasználók férhetnek csak hozzá
  - Alkalmazás
    - Játékok
    - Utility appok
    - Egyéb alkalmazások
  - Vagy ezek kombinációja
- Akkor nem kell foglalkozni a botokkal

SSR – Server Side Rendering

- Szerver oldalra átvitt kliens kód
  - Tipikusan Node.js szerverrel
  - Nem megy minden – nyilván
- Minden általunk vizsgált technológia támogatja
- Hibrid megoldás: szerver és kliens render
  - Első körben a szerveren elkészül a HTML
  - Böngésző megjeleníti
  - Majd letöltődik a teljes app
  - Átveszi a kész HTML-t, működik minden

Kliens oldali render

A probléma

- Miért kell keretrendszer?
  - HTML-t előállítani nem nehéz

```
let name = "Leo";
div.innerHTML = `<span>Kedves, ${ name }!</span>`;
```

  - Eseményekre cserélni is tudjuk
  - Bonyolult HTML-t is tudunk készíteni
    - Bár az olvashatósága egy stringben nem jó
    - Nincs szintaktikai színezés és kódkiegészítés
    - De szét tudom bontani egyszerűbb komponensekre
      - Ez amúgy is jó gyakorlat

A probléma

- A teljes HTML cserélése nem jó
  - Villog
  - Lassú
    - 10-500 ms függően az elemek számától
    - Egyszer nem gond
    - Minden bemenetre nem életképes
  - Elveszti az állapotot
    - Fókusz
    - input típusú elemek

A probléma

- Megkereshetjük a kérdéses elemet

```
let span = container.querySelector( "div span.sum" );
span.textContent = "Összeg: " + total;
```

  - Sajnos a kód függ a HTML felépítésétől
    - Karbantarthatóság problémás
  - Ha nem akarunk használni semmit, akkor ez a megoldás

Struktúra szinkronizáció

- A HTML fa felépítésű, egy ágon sok levél lehet
- Fa/lista szinkronizációs probléma (set/tree reconciliation)
  - Van két listánk A és B, generáljunk módosító utasításokat (insert, remove)
    - A-ból B legyen
    - Legkevesebb utasítás generálódjon
  - Fára ez $O(n^3)$ - lassú
- Naiv módszer: oldjuk meg csak beszúrásra és törlésre
  - Átrendezésre előről kezdjük

Megoldás

- Készítsünk egy keretrendszert, ami
  - Az általunk megadott HTML-t legyártja
  - Képes frissíteni
    - Nem villog
    - Nem lassú
    - Nem veszti el az állapotot
  - Opcionálisan
    - Adatkötést támogat, akár kétirányú adatkötést
    - Jó a tooling
      - Segít a HTML szerkesztésében
      - Ellenőrzi a kódot szerkesztés/fordítás időben
      - Segít debuggolni, ha gond van

Vizsgált keretrendszerek

- React
  - Csak UI keretrendszer
  - Legnépszerűbb
  - Nagy ökoszisztéma
- Vue
  - Szintén csak UI keretrendszer
- Angular
  - Teljes keretrendszer
- Összehasonlítás a végén

Kompozíció (összetétel)

Strukturális tervezési minta általában (Composite)

A probléma

- Szeretnénk felépíteni egy struktúrát, ami a felhasználói felületet jól leírja és kezeli
- Lehetne típusonként eltérő interfész
  - Konténereket máshogyan kezelni, mint az elemeket
  - Ez kényelmetlen a fejlesztőnek
    - Minden típusra eltérő kódot kell írni
    - Tesztelni, újra felhasználni (pl. konténerben konténer)
- Lehetne egzotikus struktúra
  - Nem fa, hanem például gráf
  - Meg akarjuk oldani, hogy egy elemnek több szülője is legyen, stb.

A megoldás (kompozíció minta)

- Egy elemet (komponens) attól függetlenül akarunk kezelni, hogy az egy konténer, vagy csak egy pici rész/elem
  - Egymásba ágyazhatóság miatt legyen minden komponensnek egy olyan interfésze, ami támogatja a minimumot függetlenül attól, hogy levél, vagy ág
- Fa struktúrában szeretnénk tárolni
  - Egy szülő és 0, vagy több gyerek
  - Adjuk fel az egyéb struktúrákat, mert nehéz kezelni őket

A megoldás

- Meg lehet oldani objektum orientált módon
  - Örökléssel: Levél és Konténer származik komponensből
- Vagy simán minden komponens
  - Ebben az esetben futásidőben kell megoldani, hogy ha valaminek nem lehet gyereke
    - Ez nem feltétlen gond, mert a kivételeket amúgy is kezelni kell

Kivételek

- Készíthetünk olyan elemet, aminek
  - Nem lehet gyereke
  - Megadott számú gyereke lehet csak
    - Például 1
- Ezeket nehéz örökléssel kezelni
  - Főleg, hogy adatfüggő is lehet
- A fa felépítésénél kell hibát dobni
  - Ez lehet futásidőben
  - Vagy a szerkesztő eszköz által
    - Ez utóbbi gyakori

Felhasználása

- Nagyon gyakran használt minta felhasználói felület kialakítására
  - Adja magát
- Minden általunk tárgyalt keretrendszer ezt használja
- A komponensek célja
  - Dekompozíció: részekre bontani a bonyolult felületet
  - Felelősség: csak saját magán belül felelős, de ott mindenért
  - Újrafelhasználás
    - Jól körülhatárolt, ezért jó eséllyel működik máshol is

Komponens

- A komponensben van
  - Nézet (HTML) leírása: sablon, vagy kód
  - Nézettel való interakció: események, adatkötés
  - Állapotkezelés
- A komponensek egymásba ágyazhatók
  - Így épül fel a logikai fa
  - Vannak komponensek, amik csak levelek lehetnek

Eszközök

VS Code, webpack, ...

Történelem

- 2007: Steve Jobs vízionálja a webet, mint appot telefonon (iPhone bemutatása)
  - De nem ez valósult meg, 2008-ban kijött az AppStore
- 2010 körül váltak a böngészők futtatókörnyezetté
  - Előtte lassú volt minden, nem volt általános a web alapú alkalmazás
- 2010: NPM – Node Package Manager
- 2010: AngularJS, ez zsákutca lett
- 2013: React
  - 2019: Hooks

Történelem

- 2013: Electron
  - Böngésző elég gyors
- 2014-től jelennek meg a csomagolók
  - Webpack (2014), Rollup (2015), Parcel (2017) és társai
  - Fejlesztés közben is képesek csomagolni, akár hot reload képességekkel
  - A végső csomag optimalizált
    - Tree-shaking (DCE: Dead Code Elmination)
    - Lazy loading, modulokra bontott
    - Minimalizált
    - Verzionált, hogy pl. ne ragadjon be régi verzió a cache-ben

Történelem

- 2014: Vue
- 2015 PWA
  - Az ötlet, hogy átvegyük a vezérlést a cache felett
  - De ekkor még használhatatlan
- 2015 Visual Studio Code, Webstorm, …
  - Webstorm (Jetbrains) eredetileg 2010-es, de alapvető funkciókat később kap csak
  - Visual Studio nem jól használható ebben a modellben
- 2016: Angular
- 2020 ESM csomagolók (Vite, WMR, …)
  - Nem csomagolnak fejlesztés közben

Történelem

- Jelen
  - Lassan változnak szokások
  - Korábban elkezdett projektek nem kerülnek konvertálásra általában
  - Egyre nagyobbak az alkalmazások
    - Egy 5-10 évvel ezelőtti projekt webpackben 1-10 másodperc alatt frissül
    - Mai projekt több perc is lehet
  - Állandóan változó eszközpark a mai napig
    - Nincs integrált környezet, ellentétben sok más technológiával

Böngésző motor

- Jelenlegi böngészők a leggyorsabb UI-t biztosítják
- JS motorok lehetővé tették a szerver oldali programozást
- PWA-kat lehet csomagolni AppStore-ba és PlayStore-ba
- JS és TS alapú multiplatform fejlesztés egyre gyakoribb
  - JavaScript talán a leghasználtabb nyelv jelenleg

Kérdések?

Multiplatform szoftverfejlesztés

React

React

Hello, World

Hello, World

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
```

- Létrehozunk egy komponenst (Greeter)
  - Ami egy függvény
  - Kap egy p paramétert
    - Amiben egy name tulajdonság van
  - Meghívja a React.createElement függvényt
    - Létrehoz egy h1 objektumot (JS objektum, nem HTML)
    - Hello+p.name+"!" tartalommal
  - Visszaadja a kapott React fát

Hello, World

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
ReactDOM.render(React.createElement(Greeter, { name: "Leo" }),
    document.body);
```

- Meghívjuk a ReactDOM.render függvényt
  - Paraméterek: tartalom és konténer
  - A tartalmat a React.createElement állítja elő
  - A mi komponensünket használva
    - És átadva neki a paramétert

JSX – babeljs

- Egyszerűsített szintaktika
- A fordító átalakítja a kódot az előző formára

```
let Greeter =  function(p) {
    return <h1>Hello, { p.name }!</h1>;
}
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

```
var Greeter = function (p) {
    return React.createElement("h1", null, "Hello, ",p.name,"!");
};
ReactDOM.render(React.createElement(Greeter, { name: "Leo" }),
    document.body);
```

TSX – TypeScript

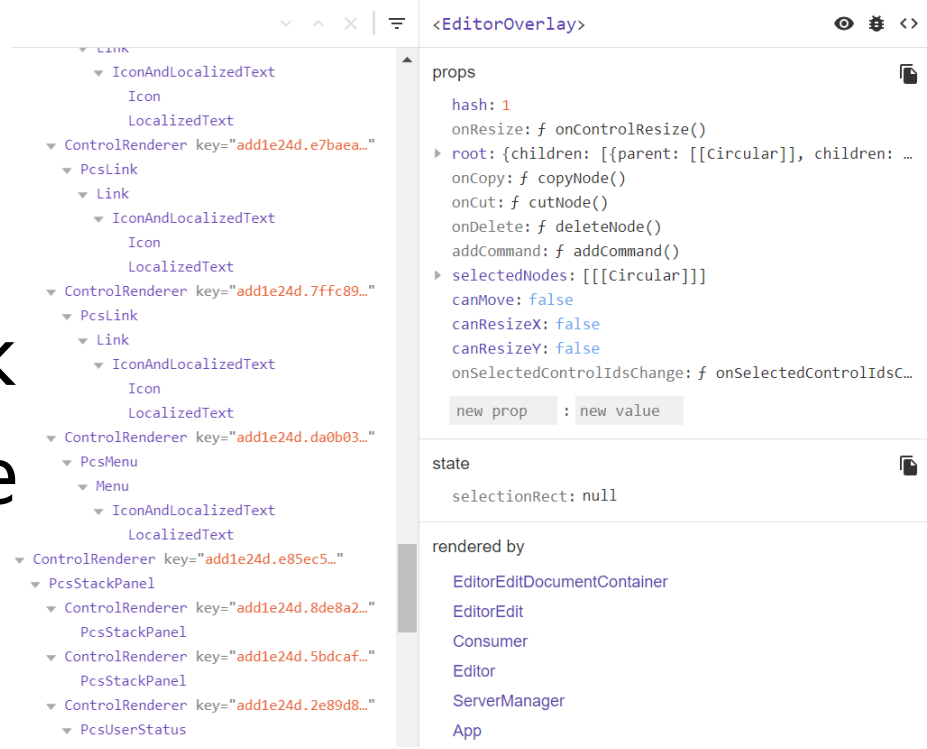
- Típusos JSX

```
let Greeter =  function(p) {
    return <h1>Hello, { p.name }!</h1>;
}
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

```
let Greeter = ( p: {name: string} ) => <h1>Hello, {p.name}!</h1>;
ReactDOM.render( <Greeter name="Leo" />, document.body );
```

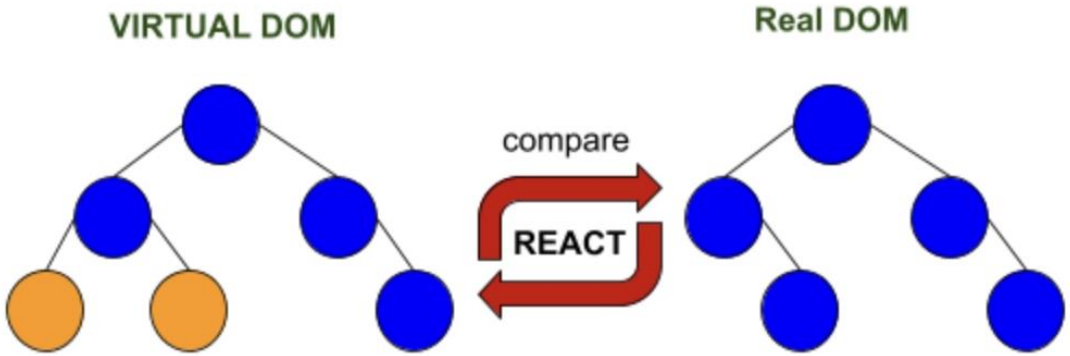
- Kapcsos zárójelek között saját kód
  - Csak kifejezést lehet beírni, ami visszaad valamit

React fa



- A létrehozott elemek nem HTML elemek
- Ugyanúgy hierarchiába vannak rendezve
  - Ez adja majd a HTML fát
- A render hívás végzi el a HTML-re konvertálást
  - Szinkronizációs folyamat, amit csak JS objektumokon végez el
    - Keresi a változásokat (tree reconciliation)
  - Az eredmény egy változás lista
    - Első futáskor az eredmény a teljes fa
    - Utána csak a különbség

React fa



- A különbségből előáll a parancslista
  - Tényleges HTML változtató parancsok
  - Ezt végrehajtja
- Lehetne vizsgálni a tényleges HTML fát is
  - Lassú
- Kézzel beleírni a HTML-be nem lehet
  - Nem észleli
  - Nem írja vissza
  - De ha változik a React fában, akkor felülírja
  - Megbízhatatlan megoldás

React

Összetett komponensek

Osztály komponens

- Osztály, mint komponens (függvény helyett)
  - Egységbe zárás
    - Összegyűjthetjük a komponenshez tartozó függvényeket
  - render függvény kötelező
  - Első típus paraméter a props típusa
  - Lehetnek belső változói

```
class GreeterC extends React.Component<{name: string}>{
    render(){
        return <h1>Hello, {this.props.name}!</h1>;
    }
}
```

Osztály komponens előnyei/hátrányai

- OO mintát jobban kövei, egyszerűbb megérteni
  - Bonyolult komponenseknél jobban látható, hogy mit történik
- Elvileg többet tud – pl. belső változók
  - Ritka az az eset, amikor szükség van erre
  - És akkor is meg lehet oldani máshogy
- Hosszabb kódot eredményez
  - Pár sorral több
- this probléma
  - Mindenhol arrow functiont kell használni, vagy bindolni (később)

Tulajdonságok (props)

- Publikus interfész
  - Elérhető kívülről
  - JSX/TSX támogatja a beállítást
    - Mintha egy sima HTML attribútum lenne
- Nem változtathatjuk belülről – paraméterként viselkedik
- Minden rajzoláskor újra megkapjuk a szülő által adott tulajdonságokat
  - Az előző rajzoláskori tulajdonságok elvesznek
  - Nem alkalmas állapot tárolására

Állapot kezelés (state)

- Belső állapot
  - Ez sok komponensnek nem lesz – állapotmentes
- Megmarad az értéke rajzolások között
  - Ezért alkalmas állapot kezelésre
- Inicializálni kell konstruktor időben

```
state = { name: "" };
```

- Típusa, amit beállítunk kezdő értéknek

Állapot kezelés (state)

```
class Counter extends React.Component<{}, {c: number}>
{
    state = {c: 0};
    inc() { this.setState( { c: this.state.c + 1 } ); }
    render() { return <p>Counter: { this.state.c }</p>; }
}
```

- Állapot kezdeti értéke `{c: 0}`, típusa `{c: number}`
- setState állítja
  - Ez egy rajzolást is kivált
  - Máshogyan nem lehet állítani az állapotot

Állapot kezelés (state) – aszinkron

- setState aszinkron
  - Nem akkor állítja be, amikor meghívjuk
  - Ez optimalizáció miatt van
    - Előbb végigmegy a teljes fán, és csak a végén állít be mindent
  - Számunkra ez nem tűnik fel
    - Kivétel, ha az állapot előző értékét felhasználjuk az új érték állításához

```
inc() {
    this.setState( { c: this.state.c + 1 } ); // nem mindig lesz jó
}
```

Állapot kezelés (state) – aszinkron

- Az aszinkron állapot állítás problémára van megoldás
  - setState tud kezelni függvényt is

```
inc() {
    this.setState( state => {c: state.c + 1} );
}
```

- Csak akkor kell használni, ha az aszinkron működés problémás lehet
  - Nem gyakori

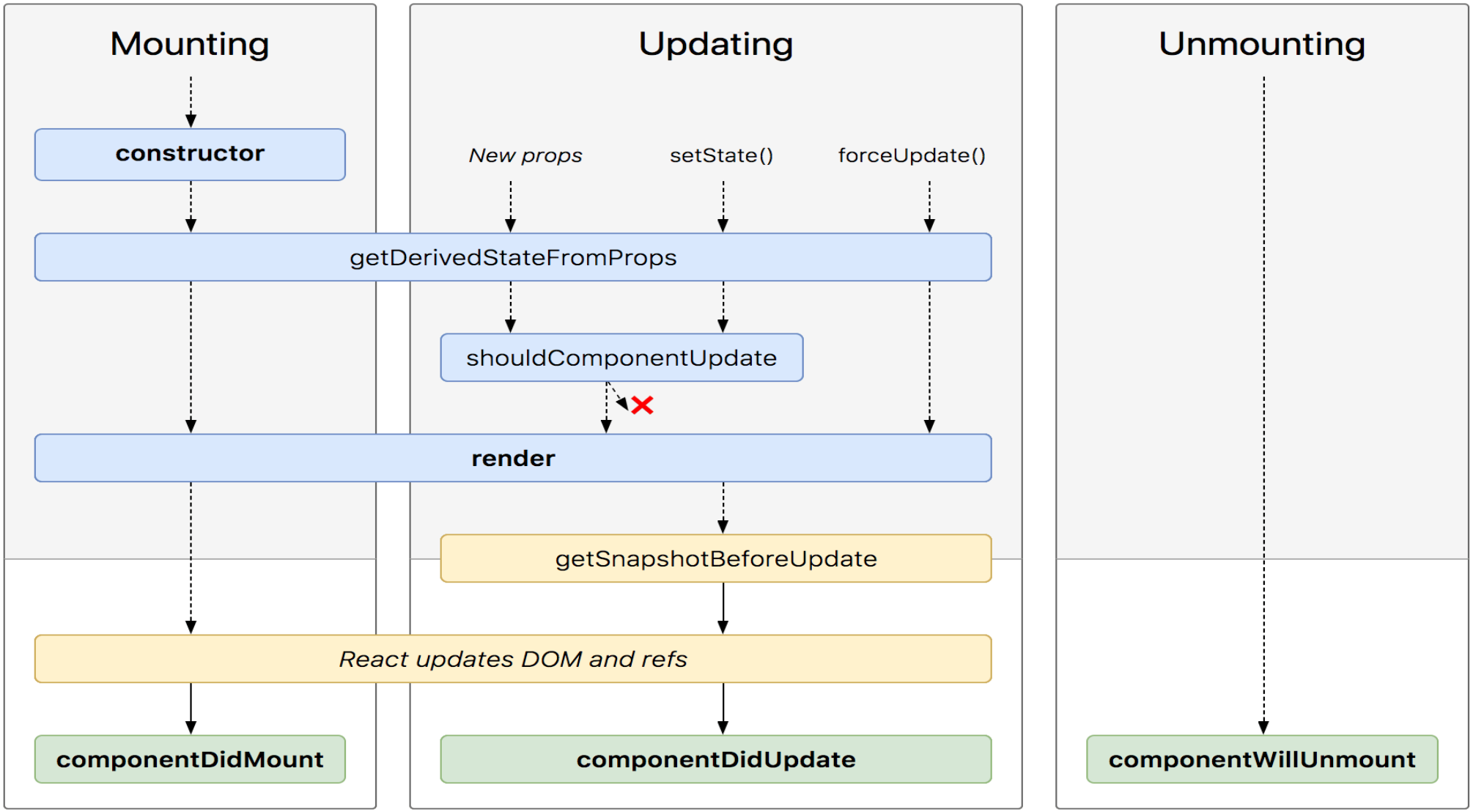
Állapot kezelés (state) – merge

- Állapot állításkor az egyes állapotok külön állíthatók

```
state = { a: 1, b: 2, c: 3 };
inc() { this.setState( { a: 10 } ); }
```

- A kód csak az `a` állapotot állítja át
  - `b` és `c` marad 2 és 3
- Csak a felső szintű állapotokra vonatkozik – shallow

Komponens életciklusa



Életciklus kezelés

- Tipikus beavatkozási helyek, amikor a keretrendszer meghívja a komponensünket
  - constructor – létrehozáskor
  - componentDidMount – első rajzolás után
  - componentDidUpdate – többi rajzolás után, ritkán használt
  - componentWillUnmount – eldobás előtt
- Létrehozhatunk saját tulajdonságokat
  - Nem a props-ban és nem a state-ben
  - Ha nincs köze a rajzoláshoz, vagy kézzel akarjuk kezelni
  - Életciklusa azonos a state-tel

Életciklus kezelés

```
class Counter extends React.Component
{
    private timer: number;
    componentDidMount()
    {
        this.timer = setInterval( () => this.forceUpdate(), 1000 );
    }
    componentWillUnmount()
    {
        clearInterval( this.timer );
    }
    render()
    {
        return <span>{ new Date().toLocaleTimeString() }</span>
    }
}
```

Kompozíció

React megoldása

render

- A felület leírása kóddal történik
  - Hibrid megoldás, nem tisztán deklaratív
  - Egyre népszerűbb, több keretrendszer megy ebben az irányban
- A kódba (render) szinte bármit beírhatunk
- A lényeg, hogy egy fát adjon vissza
  - Ami leírja a felületet
  - Továbbra sem kézzel hozzuk létre a HTML-t
- A teljes szinkronizációt a keretrendszer végzi

Egymásba ágyazás

- A komponensek egy fát alkotnak
  - Statikus gyerekek (fixen beírva)
  - Feltételes gyerekek (if-ben)
  - Generált gyerekek (tömbben)
- Fából több lehet
  - Több konténerbe is tehetünk React fát
  - Egy alkalmazás pici része is lehet React-es
    - Vagy több része egymástól függetlenül
  - Nagyon sok ne legyen
    - Például generálunk egy 1000 cellából álló táblázatot és minden cella egy React fa…

Statikus és feltételes gyerekek

- Statikus elemek esetén nincs kód a renderben
- Feltételes esetben használhatunk
  - && operátor: JS-ben a második tagot adja vissza, ha az első tag igaz
    - Hamis esetben hamis értéket ad vissza, amit React úgy értékel, hogy nincs ott semmi

```
return <header>
  <h1>Hello, Leo!</h1>
  {props.C && <Comp2/>}
</header>;
```

  - ?: operator – feltétel függően csak az egyik

Lista létrehozása

- Tömböt kell visszaadni

```
return <ul>
        { ar.map( (x,idx) => <li key={ar[idx]}><Dot/></li>) }
    </ul>;
```

- Kell key attribútum
  - Ez azonosítja az elemet
  - Innen tudja a React fa szinkronizáció
    - Melyik új elem
    - Melyik törölt
    - Melyik változott

Lista létrehozása – key

- A key egyedi kell legyen a tömbön belül
  - De nem globálisan
- Tipikusan a key egy ID az adatbázisból
- Ha nem tudunk jó key-t adni, akkor használjuk az indexet
  - Ez általában megoldja a problémát
  - Ha az elemeket átrendezzük, akkor lassú
- Új elem létrehozása, aminek még nincs ID-je
  - Adjuk neki olyat, ami amúgy nem lehetséges (pl. -1, vagy "boo")

Több gyökérelem

- A komponens több elemet is visszaadhat
  - Tömbként, vagy <Fragment> virtuális elemmel, vagy röviden <>
- Ez akkor fontos, ha egy wrapper HTML elem (pl. div) elrontaná a formázást/layoutot
  - Ha nem rontja el, akkor betehetünk gyökérnek egy div-et
  - Kerülendő – általában ne tegyünk felesleg plusz elemeket a HTML fába
- Tipikus példa a flexbox
  - Nem lehet plusz elemeket betenni, mert elromlik

Feltételes attribútumok

- Ha feltétel függően akarunk betenni egy HTML attribútumot
  - Hamis értéket adunk neki

```
autoFocus={this.props.autofocus}
```

    - Ez működik érték nélküli attribútumokra
      - Pl. disabled, required, stb.
  - Vagy használjuk a spread operátort

```
let attrs: any = {};
if ( condition )
    attrs.disabled = true;

<input { ...attrs } />
```

class, for, classList

- Az egyes attribútumok elnevezése a JS szintaktikát követi
  - Nem class=“…”, hanem className="„
  - Nem for=“…”, hanem htmlFor=“…”
  - (preact-ben nincs ez a megkötés, ott lehet class-t használni)
- Picit zavaros, mert úgy tűnik, mintha HTML-t írnánk (JSX, TSX miatt)
  - De ez átfordul kódra, ahol a class és for kulcsszavak
- Nincs classList
  - De amúgy is kódból állítjuk elő a class listát
    - Van segédkönyvtár, ha bonyolult: classcat

Komponens gyerekei

- Ha a komponensünkbe beletesznek tartalmat

```
<MyComp>
    <button>Push</button>
</MyComp>
```

- Azt a props.children-en keresztül érjük el
- Bárhogyan felhasználhatjuk segédfüggvényeken keresztül
  - React.Children.map
  - React.Children.count
  - React.Children.toArray
  - …

Öröklés

- Kerülendő (React ajánlás)
  - Az általánosabb komponensből ne származtassunk, hanem a props-on keresztük specializáljuk

Felhasználói bemenet

Esemény kezelés

- HTML elemek eseményeire feliratkozhatunk

```
render(){
    return <p>Counter: { this.state.c }<br />
        <button onClick={ () => this.inc() }>Inc</button></p>;
}
```

- Vagy arrow function, vagy bind
- Hívhatunk bármit
  - Saját függvényt
  - props-ban kapott szülő/külső függvényt
  - Globális függvényt

input és társai

- Felhasználó bemenet kiolvasása
  - Gombnyomásra (pl. elküld gomb)
  - Változásra, például validáláshoz
- Típusok
  - <input type="text"> és társai: textbox
  - <input type="file">: fájlválasztó
  - <textarea>: multiline
  - <select>: combobox
- Nem ide tartozik
  - <input type="button">, <input type="checkbox">, …

state-ben tárolt állapot

```
class TextInput extends React.Component<{}, { value: string }>
{
  state = { value: "" };
  render()
  {
    return <input type="text" value={ this.state.value }
      onChange={ e => this.setState({value: e.target.value})} />
  }
}
```

- setState hívás helyett lehet validálni, stb.

DOM-ban tárolt állapot

- Elsődleges állapot a DOM-ban van
  - El sem tároljuk a state-ben – felesleges
- Amikor szükségünk van rá, kiolvassuk
- Ehhez kell egy referencia
  - Nem triviális, mert a generált objektum elérhetetlen
  - Tudunk referenciát adni objektumokra (ref)

DOM-ban tárolt állapot

```
class TextInputU extends React.Component<{}, {}>
{
    input = React.createRef<HTMLInputElement>();
    push() { alert( this.input.current.value ) }
    render()
    {
        return <div>
            <input type="text" ref={ this.input } />
            <button onClick={ () => this.push() }>Push</button>
        </div>
    }
}
```

Hol legyen az állapot?

- <input type="file"> esetén DOM lehet csak
  - Nem lehet állítani a value-ját
- Kezdeti értéket mindkettő támogatja
  - value a state esetben
  - defaultValue a DOM esetben
  - Azért van különbség, mert a value beállítására a vezérlő írhatatlan lesz a felhasználó számára
    - onChange hívódik így is, ezért működik
    - null-t adva mégis írható lesz

Típusok

- Mit hogyan kell használni

Típus	**Érték**	**Change callback**	**Érték a callbackben**
<input type="text" />	value="string"	onChange	event.target.value
<input type="checkbox" />	checked={boolean}	onChange	event.target.checked
<input type="radio" />	checked={boolean}	onChange	event.target.checked
<textarea />	value="string"	onChange	event.target.value
<select />	value="option value"	onChange	event.target.value

Validáció

- PropTypes modul
- Típus ellenőrzés
  - Ezt TypeScript miatt automatikusan kapjuk
  - Csak a fordítás időben ellenőrizhető adatra
    - Ez szinte minden, ha nem használunk any-t
- Saját ellenőrző
  - Error-t kell visszaadni, ha hibás
- Megkövetelhető, hogy csak 1 gyerek legyen

Hooks

Osztály helyett függvény

Hook – függvény

- Nem kell osztályt írni
- Állapot useState hívással megszerezhető

```
function TextInputHook(props)
{
    let [value, setValue] = React.useState(props.def);
    return <input type="text" value={value}
        onChange={e => setValue(e.target.value)} />
}
```

- Hooks nem tud többet az osztálynál
  - Tömörebb szintaktika

Állapot kezelés

- useState visszaadja az állapotban tárolt értéket és az állapot beállító függvényt
- Állapot már nem egy objektum, hanem lista
  - Egyesével lekérdezhető, sorrend fontos
    - Például a második useState hívás a 2. állapotot adja vissza
    - Nem hagyhatunk ki useState hívást (nem lehet if-ben)

Életciklus kezelés

- Osztálynak voltak életciklus függvényei
  - Fel tudott iratkozni külső eseményekre
- useEffect az életciklus kezelő
  - Függvényt adunk át
  - Meghívja minden rajzolás után
  - Ha visszaadunk egy függvényt, akkor azt meghívja a componentWillUnmount idejében

Életciklus kezelés

```
class Time extends React.Component
{
    private timer: number;
    componentDidMount()
    {
        this.timer = setInterval( () => this.forceUpdate(), 1000 );
    }
    componentWillUnmount()
    {
        clearInterval( this.timer );
    }
    render()
    {
        return <span>{ new Date().toLocaleTimeString() }</span>
    }
}
```

Életciklus kezelés

- Át kell álljunk state-re, csak az vált ki rajzolást

```
function TimeHook()
{
  let [ time, setTime ] = React.useState( "" );
  React.useEffect( () =>
  {
    let timer = setInterval(()=>setTime(new Date().toLocaleTimeString()),1000);
    return () => clearInterval( timer );
  }, [] );
  return <span>{ time }</span>
}
```

- useEffect 2. paramétere a függőség
  - Mikor futtassa a megadott függvényt – itt soha

Sebesség

Alapok

- A render folyamat alapban minden komponenst érint
  - Azokat is, amik nem változtak
  - A rendszer nem tudja, hogy változnak-e, meg kell hívni a render-t, hogy ez kiderüljön
  - Onnan indul, ahol változás történt
    - Tehát csak a részfán megy végig
- Ez a működés optimalizálható
  - A nem változott komponenseket nem kell vizsgálni
  - De valahogyan tudni kell, hogy melyek ezek

PureComponent

- Nem a Component-ből, hanem a PureComponent-ből származtatunk
  - Csak akkor hívódik render, ha a props, vagy state változott
  - Egyéb belső állapota nem lehet (state-en kívül)
- Ha nem ennyire tiszta a rendszer
  - shouldComponentUpdate függvény
  - Meghívódik, hogy kiderüljön, kell-e rendert hívni
  - A PureComponent ezt használja
    - Megnézi, hogy a props és state változott-e

React.memo

- A PureComponent függvény (hook) verziója
  - Csak akkor hívja meg a függvényt, ha a props, vagy state változott
- Különben az eltárolt fát használja – innen a név

```
var comp = React.memo( props =>
{
  // ...
} );
```

Sebesség

- A React fa generálása és hasonlítása gyors
- Komplex felület esetén (10000+ elem) nem gyors
  - Mikor van ilyen sok elem?
    - Táblázatot kell megjeleníteni bonyolult cellákkal, details sorral, stb.
- Teljes komponensek (és gyerekeik) kihagyása jelentősen gyorsít
- Maga a HTML render, amit a böngésző végez, ma már gyors

Kérdések?

Multiplatform szoftverfejlesztés

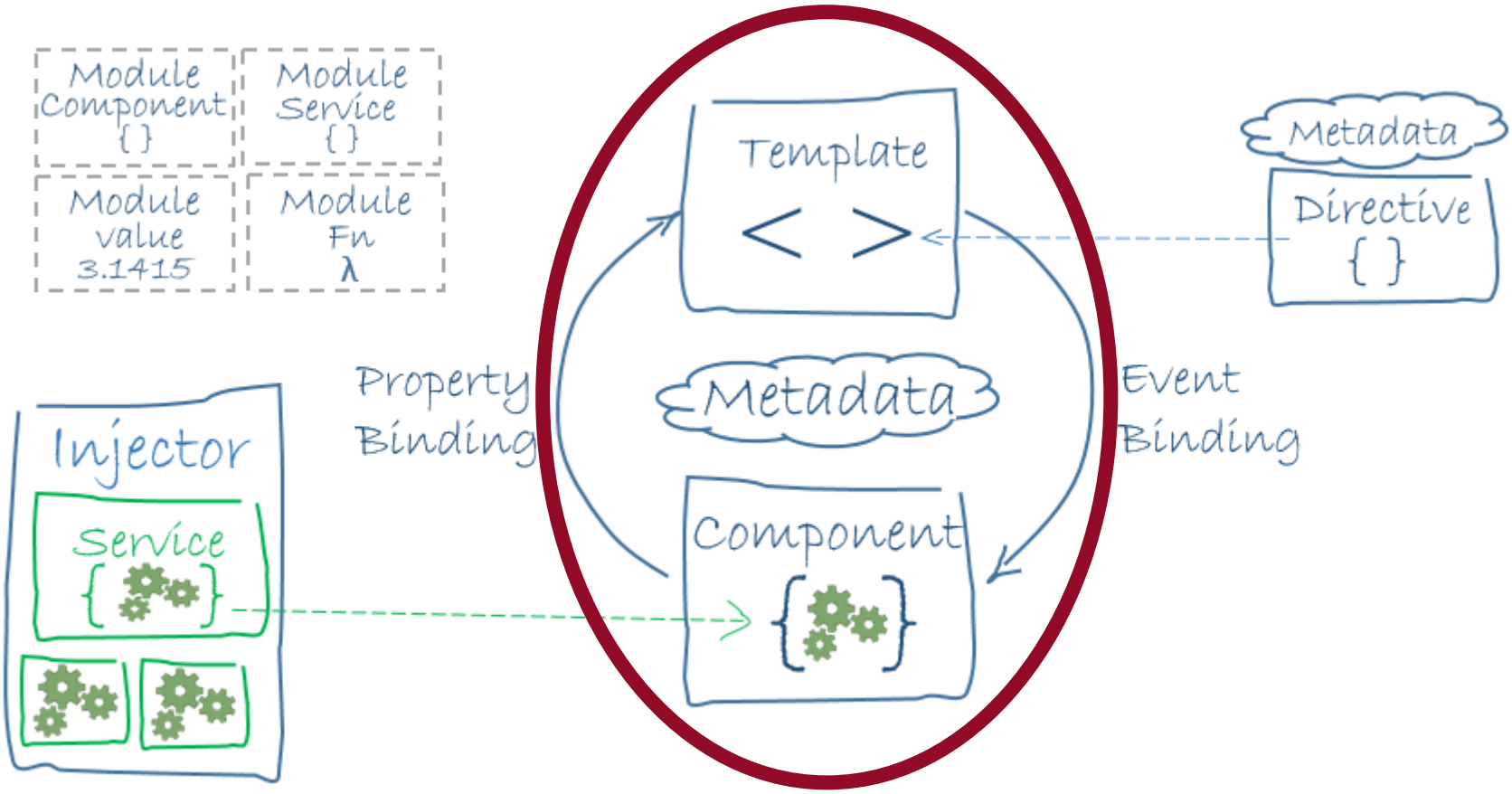
Angular

Emlékeztető

- Mit is akarunk megoldani?
  - Automatizált HTML frissítés
    - Listákkal, feltételekkel
  - Input kezelés
  - Adatkötés (kétirányú, ha lehetséges)
  - Kompozíció
    - Felület komponensenkénti kezelése
  - Tooling
    - Debug, test

Angular

Angular architektúra



Komponens

- X.component.ts
  - A komponens kódja (TypeScript)
- X.component.html
  - A megjelenítésért felelős HTML sablon
- X.component.css
  - A komponenshez (HTML sablonhoz) tartozó CSS
- X.component.spec.ts
  - Teszt kód, opcionális
  - A komponenssel együtt írhatjuk a unit tesztet

Hello World (component)

```
<p>Hello, {{name}}!</p>
```

```
import { Component } from '@angular/core';
@Component( {
  selector: 'welcome',
  templateUrl: './welcome.component.html',
  styleUrls: [ './welcome.component.css' ]
} )
export class WelcomeComponent
{
  name = "Leo";
}
```

@Component decorator

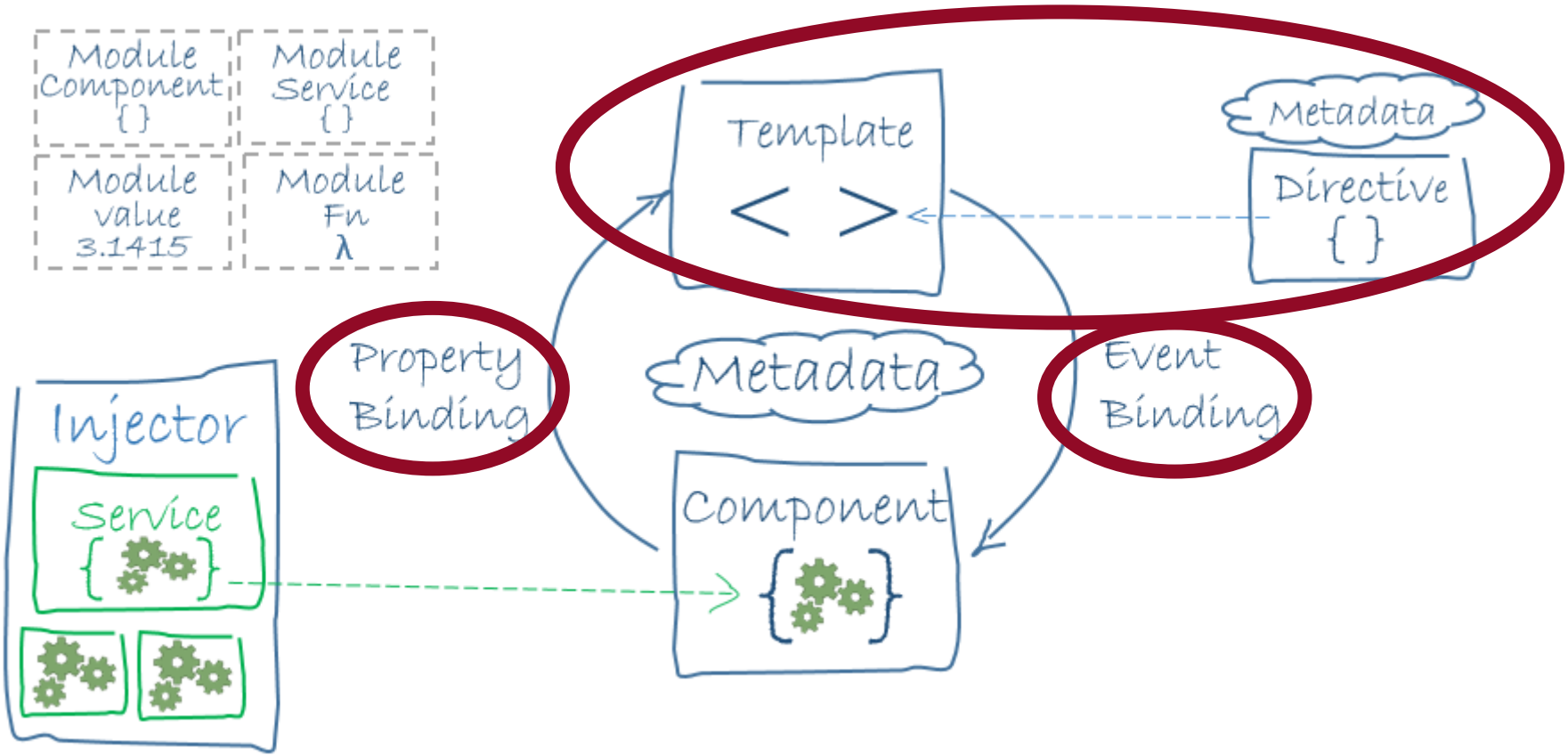
- Importálni kell
- Mit tud egy decorator?
  - Kb. mint attribútum C#-ban, itt mindig függvény
  - Írhatunk saját decoratort is
- Plusz információt ad – metaadat
  - selector: milyen HTML tag-re tegye magát
  - template, vagy templateUrl
    - Közvetlenül a HTML sablon, vagy az elérhetősége
  - styles, vagy styleUrls
    - Maga a CSS, vagy elérhetősége

export class

- A komponens kódja osztály formában
- Kívülről is látható (export kulcsszó miatt)
- Normális osztály, bármit beleírhatunk
  - Bizonyos függvények speciális funkcionalitást valósítanak meg (ld. életciklus)
- Adatkötés forrása
  - Az osztály minden tulajdonságához tudunk kötni

Sablon szintaktika

Angular architektúra



HTML sablon

- Mint Vue-nál: normál HTML + attribútumok
  - Emlékeztető: React-nél kód van
  - Szinte minden HTML tag megengedett
    - script, html, body, base nem
- Adatkötés
  - Egyirányú (komponens ⇨ sablon) : {{}}, [] = "..."
  - Egyirányú (sablon ⇨ komponens): ()
  - Kétirányú: [()]
- Strukturális direktívák
  - Az adott HTML elem megjelenjen-e vagy sem
  - *ngIf, *nfGor, *ngSwitch stb.
  - Sajátot is készíthetünk

0, vagy 1 elem: *ngIf

- Kifejezést fogad el

```
<p *ngIf="numbers.length>3">Sok szám</p>
```

- Teljesen kiveszi az elemet
  - Az elem tartalmát nem értékeli ki
  - Teljes részfát törli, komponenseket is
- Ha csak rejteni akarjuk
  - display-t kell none-ra állítani kötéssel
  - Nincs külön megoldás, mint pl. Vue-ban
- Van ngSwitch, mintha sok ngIf lenne
  - Kényelmesebb

0, vagy több elem: *ngFor

- "let x of c" szintaktika

```
<li *ngFor="let n of numbers">{{n}}</li>
```

  - x az elem
  - c a gyűjtemény
- Megszerezhetjük az indexet is

```
<li *ngFor="let n of numbers; let i=index">{{i+1}}: {{n}}</li>
```

- Azonosító megadása: trackBy (React: key)
  - Függvény, ami visszaadja a kulcsot

```
<li *ngFor="let n of numbers; trackBy: numbersKey">{{n}}</li>
```

Adatkötés {{}}

- Betehetjük önállóan, vagy szöveg mellé

```
<p>Hello, {{name}}!</p>
```

- Egyirányú adatkötés (komponens ⇨ sablon)
- Tetszőleges TS kifejezés lehet benne
  - Kivétel olyanok, amiknek mellékhatása van
    - Például értékadás
  - Cél, hogy ne legyen bonyolult
    - Deklaratív megoldásokat nehéz tesztelni
- Attribútumban is működik

Adatkötés [attrib]=“expr”

- Azonos hatása van, egyirányú
- De ez a DOM elem tulajdonságához köt

```
<p [id]="numbers[0]">ID</p>
```

  - Esetünkben azonos az attribútummal (id)
  - De a legtöbb esetben más (className vs. class)
  - Vagy nincs is olyan tulajdonság (aria-label)
- Ha az elem egy komponens (és nem DOM elem), akkor annak a tulajdonságához köt
- Alternatív szintaktika
  - [prop] helyett bind-prop

Adatkötés [class]

- A class attribútum egy lista, a kötése egyedi
- Egyesével ki-be kapcsolni osztályt

```
<p [class.centered]="isCentered">Közép</p>
```

- Egyszerre többet kezelni
  - ngClass direktíva

```
<p [ngClass]="{'centered': numbers.length==2}">Közép</p>
<p [ngClass]="n==3? 'centered' : 'normal'">Közép</p>
```

  - Van még pár megadási lehetőség

Adatkötés [style]

- A style attribútum egy objektum
- Egyesével

```
<p [style.display]="n==2 ? 'block' : 'none'">Hello</p>
```

  - Mértékegység megadással is lehet

```
<p [style.font-size.em]="numbers.length">Big</p>
```

- Egyszerre többet kezelni
  - ngStyle direktíva, hasonló az ngClass-hoz

Adatkötés (esemény)

- (esemény) szintaktika
  - Kódot futtat
  - Nem elég megadni a fv. nevét, meg is kell hívni

```
<button (click)="onSave()">Mentés</button>
```

- Ha szükségünk van az esemény objektumra
  - $event változóban van
- Alternatív szintaktika: on-esemény
- Egyirányú adatkötés (sablon ⇨ komponens)

Kétirányú adatkötés: [()]

- [()] szintaktika adja a kétirányú adatkötést

```
<comp [(prop)]="myprop">
<comp [prop]="myprop" (propChange)="myprop=$event">
```

  - []: egyirányú adatkötés a tulajdonságra
    - comp.prop kötve this.myprop-hoz
  - (): eseményre feliratkozás
    - comp.propChange-re
- Ez meg is oldaná a kétirányú adatkötést, ha a DOM-ban egységesen ez lenne az elnevezés
  - De nem ez, és nem egységes
- Komponenseinkben ezt érdemes használni

Kétirányú adatkötés: ngModel

```
<input [(ngModel)]="text">
```

- ngModel direktíva szükséges
  - Ismeri az összes beépített HTML elemet
    - Mindegyik eseménykezelőjére fel tud iratkozni
    - És tudja állítani az értékét
- Kell hozzá importálni a FormsModule-t
  - Form elemekre alkalmazható
- Kiterjeszthető, ha külső könyvtárhoz kéne igazítani

HTML elem azonosítása

- Megkereshetjük a fában, pl. querySelector
  - Ha átírjuk a sablont, akkor a kód hibás lesz
  - Ezt el kell kerülni
    - Szerepkörök szétválasztása fontos paradigma (separation of concerns)
- Jelöljük meg és hivatkozzunk rá

```
<p #subtitle>Felirat</p>
<p>{{subtitle.textContent}}</p>
```

- Kódban is elérhetjük: @ViewChild

Formátum konvertálók – Pipes

- Ha az adat formátuma nem megfelelő
- Használhatunk beépített konvertálót
  - date: dátum

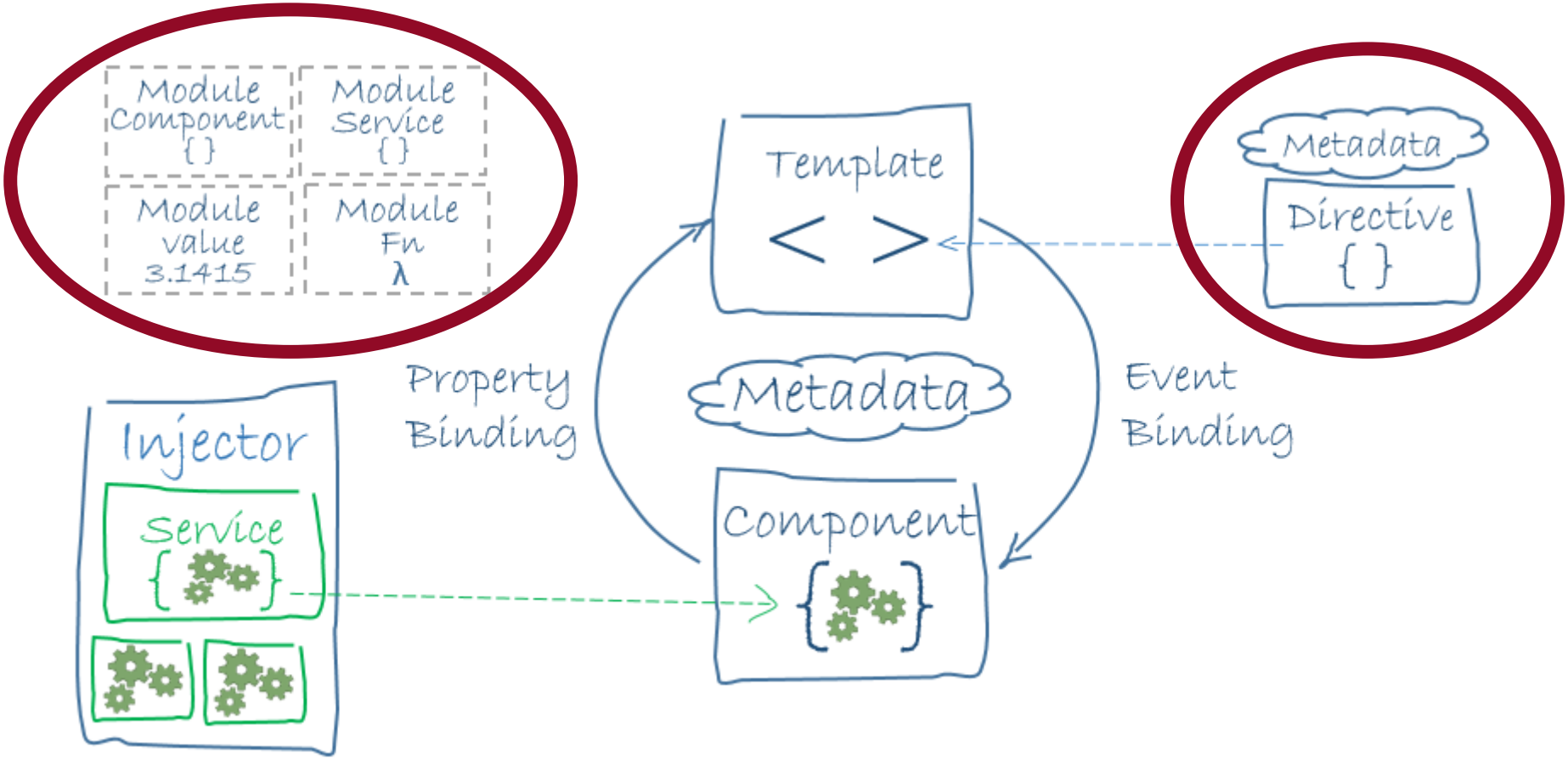
```
<p>Ma: {{ today | date }}</p>
```

  - uppercase, lowercase, titlecase
  - number: szám formázás
  - async: amint megjön az adat, frissít és kijelzi
  - json: JSON.stringify()
- Egymás után láncolhatjuk őket
- Írhatunk sajátot

Felépítés

Életciklus, modulok, direktívák

Angular architektúra



Életciklus kezelése

- Angular esetén ritkán van szükségünk rá
  - A legtöbb feladatra van kész megoldás
- ngOnInit: konstruktor helyett inicializáló logika ide
- ngOnDestroy: végső takarításra használjuk
- ngOnChanges: adatkötött tulajdonság változásra
- ngDoCheck: ha valamiért nem lehet egy változást észlelni, akkor itt beavatkozhatunk
- stb.

Modul

- A modul komponensek halmaza
  - Amik összetartoznak
  - Kifelé egységesen tudnak fellépni
  - Egy adott feladatot oldanak meg együtt
  - Komponensen kívüli kód is tartozhat hozzá
- Csak egyben lehet őket importálni/használni
- Nem JS modul, annál tipikusan kisebb
- Egy alkalmazás több modulból áll általában
- A keretrendszer is ilyen modulokból áll
  - És külső könyvtárak is

@NgModul

- Modul dekorátor

```
@NgModule({
  declarations: [AppComponent, WelcomeComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

  - declarations: modul tartalma (pl. komponensek)
  - imports: függőségek
  - providers: szolgáltatások (ld. később)
  - bootstrap: belépési pont

Modul indítása

- Main.ts-ben indul a fő modulunk

```
platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

- A fő modul pedig indítja, amit a bootstrap-ben megadtunk

Klterjeszthetőség

- Angular funkcionalitása kiterjeszthető
  - Alapvető funkciókat is kiterjesztés old meg (ngIf)
  - A keretrendszer nagy része kiterjesztés
- Direktívák
  - A HTML generálásba és komponensek működésébe beavatkozás
- Szolgáltatások
  - Funkciókat valósítanak meg komponensen kívül
  - Például adatlekérés a szerverről
- Pipe, serviceWorker, webWorker, …

Direktívák

- Beavatkoznak az elemek működésébe
  - HTML elemeken és komponenseken is működnek
  - Akár megváltoztatják a HTML fa felépítését is
- Attribútum direktíva az adott elemen működik
  - Például ngClass, ami osztályokat tesz rá/vesz le
- Strukturális direktíva a fát változtatja meg
  - Például *ngFor, ami HTML fát generál
  - * és mikroszintaktika, hogy keveset kelljen írni
    - E nélkül <template>-et kéne definiáljunk
- Sajátot is írhatunk

Direktívák

- A direktíva egy osztály
  - Nem komponens, mert nincs felülete
  - Ezen kívül szinte ugyanazt tudja, mint egy komponens
- @Input és @Output tulajdonságokkal vesz részt adatkötésben
  - De ez nem kötelező
  - @Input és @Output = dekorátorok

Attribútum direktíva

- 2 fájlból áll: az osztály és a teszt
- Egy példa, ami bold-ra állít

```
import { Directive, ElementRef } from '@angular/core';
@Directive( { selector: '[appBold]' } )
export class BoldDirective
{
  constructor( el: ElementRef )
  {
    el.nativeElement.style.fontWeight = 'bold';
  }
}
```

Attribútum direktíva – HostListener

- Eseményekre tudunk figyelni

```
@HostListener( 'mouseenter' ) onMouseEnter()
{
  this.el.nativeElement.style.fontWeight = 'bold';
}
@HostListener( 'mouseleave' ) onMouseLeave()
{
  this.el.nativeElement.style.fontWeight = 'normal';
}
```

  - A HostListener megoldja a feliratkozást és a leiratkozást

Attribútum direktíva – adatkötés

- @Input lehetővé teszi az adatkötést
- Ha azonos a neve a direktívával, akkor default
  - Eltérő név esetén a komponensben meg kell adni a nevet

```
@Input( 'appBold' ) bold: boolean;
@HostListener( 'mouseenter' ) onMouseEnter()
{
  this.el.nativeElement.style.fontWeight = this.bold ? 600 : 300;
}
```

- Használata

```
<p [appBold]="numbers.length>3">Közép</p>
```

Attribútum direktíva – adatkötés

- Van @Output is a kétirányú adatkötéshez

```
@Output( 'appBoldChange' ) boldChange = new EventEmitter<void>();
```

- emit függvény tüzeli az eseményt

```
@HostListener( 'mouseenter' ) onMouseEnter()
{
  if ( this.bold )
  {
    this.el.nativeElement.style.fontWeight = 'bold';
    this.bold = false;
    this.boldChange.emit();
  }
}
```

Attribútum direktíva – adatkötés

- Felhasználása

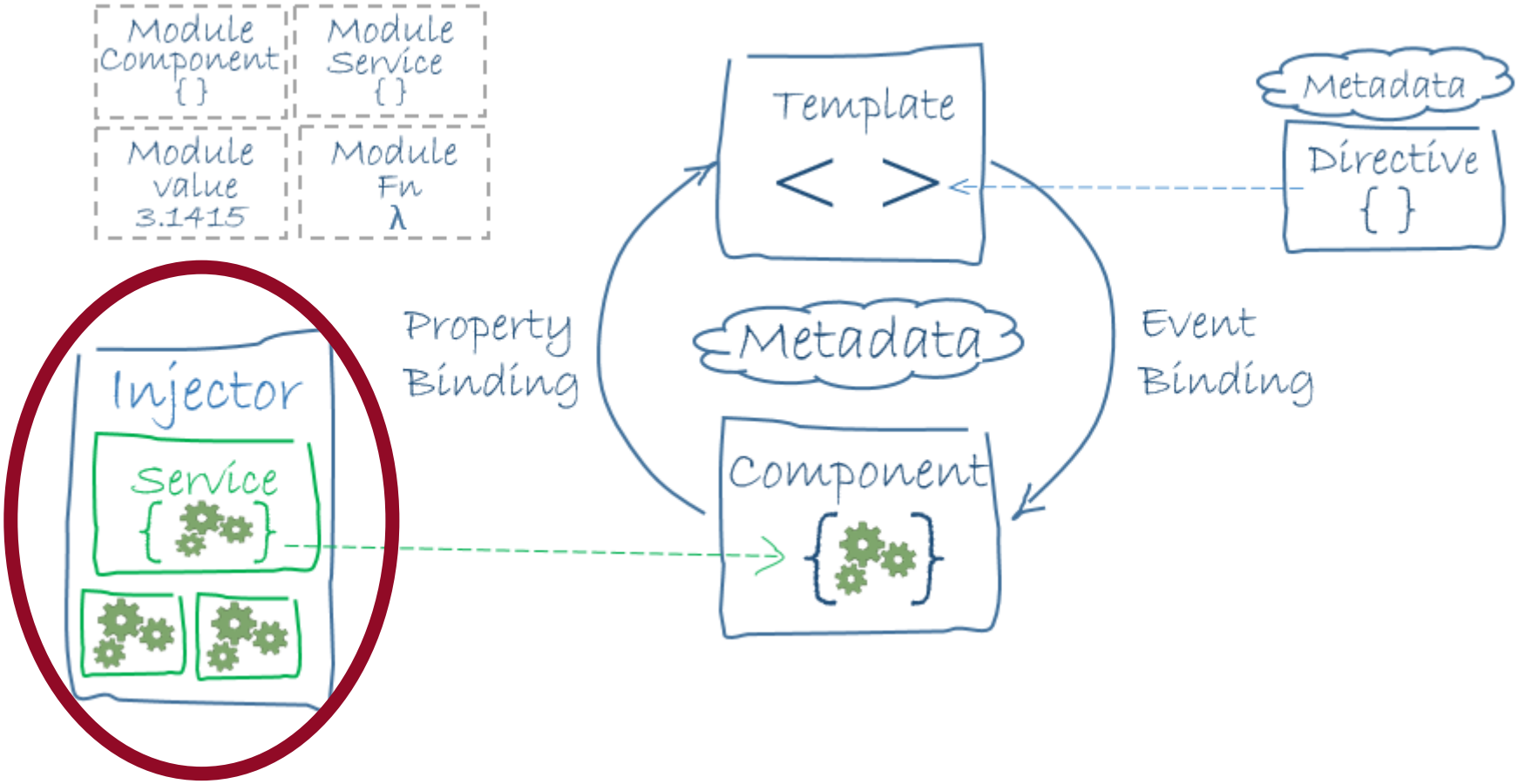
```
<p [appBold]="isBold" (appBoldChange)="isBold=$event">Bold</p>
```

- A kétirányú adatkötés szintaktikával

```
<p [(appBold)]="isBold">Bold</p>
```

  - Ez csak azért megy, mert van
    - appBold
    - appBoldChange
  - Más elnevezésnél nem működik
    - Külön ki kell írni az eseménykezelőt
- Törekedjünk erre az elnevezésre

Angular architektúra



Szolgáltatások (service)

- Tipikusan nem globálisan hozzuk létre a komponensen kívüli objektumokat
  - Szolgáltatások formájában
  - De nem kódoljuk bele egyik komponensbe sem
  - Felsoroljuk, hogy mik vannak
  - Mi hivatkozik mire
  - És melyik komponensnek mi kell
- A keretrendszer automatikusan adja
- Függőség injektálás (Dependency Injection)

Függőség injektálás (DI)

- Általánosan: Az objektum által használt függőséget nem az objektum kezeli, hanem csak kapja. Az injektor felelős a függőségek kezeléséért (életciklus, kiosztás).
- Esetünkben a komponens nem felelős a szolgáltatásokért, csak kapja őket használatra
- A komponensek jönnek-mennek, míg a szolgáltatások életciklusa teljesen eltérő
- A komponensnek nem kell tudnia, hogyan hozzon létre szolgáltatásokat

Függőség injektálás (DI)

- Szerepkörök szétválasztása miatt fontos
  - Tesztelhetőség nő
  - Dekompozíció erősödik
  - Újrafelhasználhatóság javul
  - Komponens kódja kisebb, olvashatóbb marad
- Nem csak Angular-ban van
  - Így ismerhetjük fel a DI-t: használunk egy szolgáltatást (külső funkcionalitást), ami nem globális, és nem is mi hoztuk létre

Példa szolgáltatás

- Attól szolgáltatás, hogy injektálható

```
import { Injectable } from '@angular/core';
@Injectable( { providedIn: 'root' } )
export class RandomDataService
{
  constructor() { }
  get() { return [ 1, 2, 5, 6, 7 ]; }
}
```

- Amúgy csak egy sima osztály
- providedIn: mely komponensek számára elérhető (root mindenkinek)

Szolgáltatás felhasználása

- Simán átvesszük a konstruktorban, és használjuk

```
constructor( rng: RandomDataService )
{
  this.numbers = rng.get();
}
```

- Az injektor gondoskodik arról, hogy
  - Létre legyen hozva
  - Átadja, amikor létrejön a komponens
  - Megszüntesse valamikor

Szolgáltatások

- Szolgáltatások
  - HttpClient: XHR+json kommunikáció
  - Location: address bar
  - FormBuilder: űrlap kezelő
  - Router: navigáció
  - …
- Tree shaking működik szolgáltatásokra (DI-re)
  - Csak azok kerülnek bele a végső kódba, amire hivatkozunk

Kérdések?

Multiplatform szoftverfejlesztés

Vue

Emlékeztető

- Mit is akarunk megoldani?
  - Automatizált HTML frissítés
    - Listákkal, feltételekkel
  - Input kezelés
  - Adatkötés (kétirányú, ha lehetséges)
  - Kompozíció
    - Felület komponensenkénti kezelése
  - Tooling
    - Debug, test

Vue

Hello, World

```
<template>
  <h1>Hello, {{ name }}!</h1>
</template>
<style scoped>
h1 {
  font: 48px "Segoe Print";
}
</style>
```

```
<script lang="ts">
import Vue from "vue";
export default Vue.extend({
  name: "HelloWorld",
  props: {
    name: String
  }
});
</script>
```

- .vue fájl
  - SFC: Single File Component
  - Egyben van minden: HTML, CSS, JS (TS)

HTML template

- Saját HTML template szintaktikája van
- Deklaratív felület megadás
- {{}} létrehozza az adatkötés
- v-bind szintén adatkötés, de attribútumra
  - v-bind:title=“name”
  - Leggyakrabban használt direktíva, van rövidítése
    - :title=“name”

Feltételes elem – v-if

- 0, vagy 1 elem generálása
- Tetszőleges kód, amire lehet hívni if-et
  - Hívhatunk függvényeket is

```
<p v-if="numbers.length>3">Sok elem</p>
```

  - Az adatnak léteznie kell a komponensben
    - data, props, …

```
  data(){ return { numbers: [1, 2, 4, 5] }; }
```

- Van v-else és v-else-if is, illetve v-show, ami csak display:none-t állít, nem törli

Ciklus – v-for és v-key

- 0, vagy több elem generálása
- x in c szintaktika
  - c a lista
  - x az elem, amit lehet kötni

```
<p v-for="n in numbers">item {{n}}</p>
```

- Opcionálisan kulcsot adhatunk meg
  - Hogy ne generálja újra a teljes fát változásra
  - v-key=“n”
  - A kulcs probléma azonos React-ben is

Események – v-on (@)

- HTML elem eseményeinek kezelése
- Egy függvényt adhatunk meg

```
<button v-on:click="reverseNumbers">Nyomj meg</button>
```

- Amit a methods-ban definiálunk

```
methods:{
  reverseNumbers(){
    this.numbers = this.numbers.reverse();
  }
}
```

Események – v-on (@)

- Van pár módosító kényelmi okok miatt
  - .prevent: Meghívja a preventDefault-ot
  - .stop: stopPropagation
  - .self: csak akkor hívja meg, ha ez a vezérlő váltotta ki az eseményt
  - Stb.

```
<button @click.prevent="reverseNumbers">Gomb</button>
```

Két irányú adatkötés – v-model

- Tipikusan input esetén
  - Ha változik az adat, akkor frissüljön az input
  - Ha a felhasználó beír valamit, akkor frissüljön az adat

```
<input v-model="price" />
```

- Tetszőleges adatra köthetünk
  - Ahogy az egyirányú esetben
  - Akár checkbox-ot string-re

Stílus – v-bind:class (:class)

- Vesszővel elválasztott osztálylista a feltételekkel
- Objektum szintaktika
  - Az osztálynév amit rá akarok tenni
    - Nem kell ténylegesen létezzen
    - Ha van benne kötőjel
      - Idézőjel kell (‘font-size’), vagy
      - camelCase (fontSize)
  - A feltétel egy JS kifejezés, ami igaz/hamis

```
<h1 :class="{center: isCentered}">Hello, {{ name }}!</h1>
```

Stílus – v-bind:class (:class)

- Tömb szintaktika
  - Egy tömbben felsorolva JS kifejezések
  - Mindegyiknek egy string-re kell kiértékelődni
  - Ezt az osztályt teszi rá
  - Megjegyzi, hogy melyik tette rá, így ha legközelebb mást tesz rá, akkor a régit le tudja venni

```
<h1 :class=“[centered]">Hello, {{ name }}!</h1>
```

- Automatikusan kezeli a vendor prefixeket

Komponens

Részei és kompozíció

Tulajdonságok – props

- props
  - Kívülről állíthatjuk (akár adatkötéssel is)
  - HTML szerű attribútumként jelenik meg
  - Csak olvasható
  - Egyirányban köthető
  - Tömb

```
export default {
  props: ['title'],
  // data, methods, computed, components, …
}
```

Belső állapot – data

- data
  - Belső állapot
  - Kétirányú adatkötésben is benne lehet
  - Függvény adja vissza az objektumot

```
data(){
  return {
    numbers: [1, 2, 4, 5]
  };
}
```

Segédfüggvények – methods

- methods
  - A segédfüggvényeink helye
  - Objektumon belül függvények

```
methods:{
  reverseNumbers(){
    this.numbers = this.numbers.reverse();
  }
}
```

Függőségi gráf – computed

- computed
  - Csak olvasható tulajdonságok
  - Függvény szintaktika (objektumon belül)

```
computed: {
  filteredNumbers() {
    return this.numbers.filter(x => x < 5);
  }
}
```

  - Függőségi gráfot épít fel
    - Akkor számolja újra, amikor valamelyik függőség változik (numbers)

Változás figyelés – watch

- watch
  - Bármelyik adatra rátehetünk egy függvényt, ami akkor hívódik meg, amikor az adat változik
  - A függvény neve az adat neve

```
watch: {
  price(newValue: string, oldValue: string) {
    alert(oldValue + "=>" + newValue);
  }
}
```

  - Cél: debug, log, vagy átmenet kezelése
    - Például animálni a változást

Kompozíció – components

- components
  - Gyerek komponensek felsorolása
  - Template-ben használhatók
    - Többször is, több példány jön létre

```
<script>
import Counter from './Counter.vue'

export default {
  components: {
    Counter
  }
}
</script>
<template>
  <h1>Hello, Leo!</h1>
  <Counter />
</template>
```

Életciklus

- created: miután a komponens létrejött (konstruktor)
- mounted: benne van a virtuális fában
- updated: render után
- destroyed: megszűnt (destruktor)

- Mindegyiknek van egy before… változata
  - Az adott funkció előtt hívódik meg
  - Például beforeUpdate a render előtt

Optimalizáció

- React-tel ellenétben itt nem kell PureComponent és társai
  - A függőségi gráf automatikusan megoldja ezt

Tooling

- Kényelmes fejlesztéshez kell .vue fájl
  - SFC: Single File Component
    - Benne van a HTML sablon, CSS és JS/TS
  - E nélkül gyenge a keretrendszer
    - Nincs benne HTML szintaktikai elemző, stb.
- Csak olyan eszköz jöhet szóba, ami támogatja
- Szerencsére sok ilyen van
  - Pl. Visual Studio Code, Webstorm

Kérdések?

Multiplatform szoftverfejlesztés

Összehasonlítás

React vs Vue vs Angular

Tudás

- Angular teljes keretrendszer
  - Nehezebb megtanulni
- React és Vue csak UI
  - Rátehetők az alkalmazás egy részére is
  - Gyorsan el lehet kezdeni használni őket
  - Van sok (főleg React-hoz) könyvtár
    - Ezekkel fel tudjuk hízlalni őket Angular szintre
    - Nem triviális mindent összehangolni

Sebesség

- Szinte azonos – ez alapján nem lehet dönteni
- Talán Vue a leggyorsabb (nagyon kicsi eltérés)
  - Főleg, ha nem figyelünk React-ben (PureComponent)
- Preact nagyon hasonló React-hez
  - Gyorsabb mindegyiknél
  - 9KB (3KB tömörítve)
  - Csak az alapokat tudja
- Nulláról megírva a kódot tudunk még gyorsabbat írni…

HTML

- Vue és Angular HTML sablonnal dolgozik
  - Deklaratív felület megadás
    - Bár tudunk kódot írni (kifejezést), de csak bizonyos helyekre
  - Attribútumokkal szabályoz
    - Feltételes elemek és ciklusok
- React nem tisztán deklaratív
  - Kódot írhatunk a felület leíró logikába (JSX, TSX)
  - JS/TS kód szabályozza a teljes rendszert
  - Ez elvileg többet tud
    - A valóságban ritka, amikor e nélkül nem megy

CSS

- Vue támogat CSS-t a .vue fájlban
  - Lehet scoped (csak arra a komponensre hat)
  - Nem feltétlen hasznos, attól függ, hogy ki csinálja a CSS-t
    - Ha designer, akkor nem hasznos
    - Ha a fejlesztő, akkor igen
- Angular komponensenkénti CSS-t használ
- React-ban alapban nincs ilyen
  - De vannak külső könyvtárak CSS-in-JS
    - Pl. emotion
    - Ezeknek általában van futásidejű költsége
  - Lehet fordítás idejű a CSS feldolgozás: Webpack, stb.

Fordítás

- Mindegyiket fordítani kell
  - React: JSX/TSX
  - Vue: .vue fájl
  - Angular: rengeteg fájlt csomagolni
- Elvileg lehet olyan kódot írni, amihez nem kell fordító
  - Általában borzalmas fejlesztői élmény
- Nem nagy gond a fordítás
  - Amennyire nehéz beállítani az egész rendszert

Nyelv

- JavaScript
  - Angular nem használható
  - Mindegyik másik igen
- TypeScript
  - Angular TS-ben van írva, és ezt támogatja
  - Vue (3.0) TS-ben van írva, mindent támogat
  - React JS-ben van írva, mindent támogat
    - Még TSX is van hozzá
    - Facebook Flow-t használ, ami típusos JS (nem igazán terjedt el)

Méret

- Angular a legnagyobb
  - Tree-shaking után is
- Vue és React hasonló, kisebb
  - De ha hozzáveszünk sok könyvtárat, hogy Angular képességűek legyenek, akkor megnőnek ezek is
- Preact és hasonlók nagyon kicsik

Csomagolás

- Mindegyik tudja
  - Lazy loading
  - Code splitting
  - Tree shaking
  - Minimalizálás
  - Stb.
- Webpack, Parcel, Rollup
  - Alapban a project létrehozó eszköz Webpack-et állít be
- ES csomagolók kezelik Vue-t és React-et

Komplex állapot kezelés

- Mindegyikhez van
  - Vue: Vuex
  - React: Redux
  - Angular: alapban benne van

- Nem feltétlen kell
  - Ha a komponenseknek csak a saját állapotuk kell
  - Vagy esetleg egy globális állapot elég
  - Nagyobb alkalmazásnál valami kell
    - Lehet saját megoldás is

IDE

- Sok szerkesztő van, általában amelyik tudja az egyiket, az tudja a másikat is
- Közepes eszközök
  - Ha valaki a régi IDE-ből jön, akkor jók
  - C# fejlesztőnek gyengék
  - Debuggolással gond van
    - Nem ott áll meg, nem jól lép át, nem jó a hibaüzenet
  - Csomagolás és konfiguráció problémás
    - Hozzáértést igényel
    - Nem feltétlen gyors
  - CLI megoldások: rugalmas, de bonyolult

Kérdések?

Multiplatform szoftverfejlesztés

Progressive Web Apps

Web alapú alkalmazás++

- Mint web alapú alkalmazás, plusz
  - Telepíthető
    - Indítható a bolt/web látogatása nélkül
    - Akár offline is működik – ha olyan
  - Képes elérni olyan OS szolgáltatásokat, amit weben nem lehet

Telepített kliens

- Electron
  - HTML+CSS+JS csomagolva
  - Node.js futtatja
  - Chrome motor renderel
  - API-t biztosít a fájlrendszer és egyéb OS szolgáltatások eléréséhez
  - Platform: Windows, Linux, macOS
- PWA – ezzel foglalkozunk csak
  - Telepíthető a webalkalmazás
  - Ablakban indul, böngésző UI nincs ott

PWA

PWA

- Web alapú alkalmazás
- App szerű (ez főleg UX kérdés)
  - Például nem görgethető az egész
- Telepíthető
  - Offline is működik
- Responsive – minden méretben jól működik
- OS integráció
  - Megosztás támogatás (forrás és cél)
  - Push notification
  - …

PWA – telepíthető

- Minimum követelmény
  - HTTPS
    - Ez nem gond, amúgy is így kommunikálunk
  - Manifest fájl
    - Az alkalmazás paramétereit írja le
  - Service Worker
    - Offline működést oldja meg
- Opcionális: prompt esemény
  - beforeinstallprompt
  - Ha nem kezeljük, akkor a felhasználó automatikusan kapja a telepíthető felszólítást (iOS-en nem)

HTML

- HTML head

```
<meta name="theme-color" content="white" />
<meta name="apple-mobile-web-app-title" content="My Chat" />
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="apple-mobile-web-app-status-bar-style" content="white" />
<link rel="manifest" href="manifest.json" />
<link rel="apple-touch-icon" href="Images/Logo180w.png" />
```

- manifest kötelező
- A többi csak iOS miatt kell

manifest.json

```
{
  "short_name": "My Chat",
  "name": "My Chat – Group Chat",
  "start_url": "index.html?utm_source=homescreen",
  "display": "standalone",
  "theme_color": "white",
  "background_color": "white",
  "icons": [
    {
      "src": "Images/Logo48.png",
      "type": "image/png",
      "sizes": "48x48"
    },…
  ]
}
```

manifest.json beállításai

- Telepítés után érvényesek csak
  - name: ez lesz az alkalmazás neve
  - start_url: ezt indítja el az OS
  - display: böngésző ablakban, vagy anélkül
  - theme_color: a böngésző ablakát átszínezi, ha van
  - background_color: betöltés közbeni szín
  - icons: a telepített ikon
    - Több méretben kell, mert az OS azt használja mindig ami a legjobban passzol
    - Például Windows taskbaron vs start menüben
  - share_target: megjelenik a share listában

Böngészőnként eltér

- Sajnos a manifest.json támogatása böngészőnként eltér
  - Valahol kitesz splash screen-t – itt kell nagy kép
  - Felhasználja a színeket, vagy nem
  - Stb.
- Szerencsére az unió működik
  - Mindent beállítunk minden módon – ez megy
- Bizonyos funkcionalitások nem mindenhol támogatottak
  - iOS lemaradásban
  - Címsor desktop OS-eket testre szabható

Service Worker

Service Worker

- Egy külső JS fájl
  - Nincs benne a csomagban
  - Mi írjuk meg
- Az alkalmazásunktól függetlenül fut
  - Akkor is futhat
    - Ha nem futtatjuk az alkalmazást
    - Ha a böngésző nincs elindítva
  - Háttér szolgáltatás (OS service) futtatja
- Feladata
  - Offline működés – cache
  - Push Notification kezelés

Web Worker

- Web Worker != Service Worker
  - Többszálúságot biztosít
  - Minden szál (worker) az adott alkalmazásban fut
    - Mint normális többszálú programokban
    - Életciklusuk azonos az appéval
  - Felülethez nem férhetnek hozzá
    - Az továbbra is csak a fő szálból érhető el
  - Üzenetekkel kommunikál (mint SW)
  - Elér pár API-t, amit SW nem
    - Például indexedDB, WebSocket

Service Worker

- Telepítése

```
navigator.serviceWorker.register( 'sw.js' ).then(
  reg => …,
  err => console.log('ServiceWorker registration failed') );
```

- sw.js az SW kódja
- Sikeres register hívás esetén elkezd működni
  - Tudunk push notificationt fogadni
  - Cache elindul
- Csak a következő betöltés megy teljesen cache-ből

Service Worker

- fetch esemény
  - Ha nincs benne a cache-ben, akkor töltse le

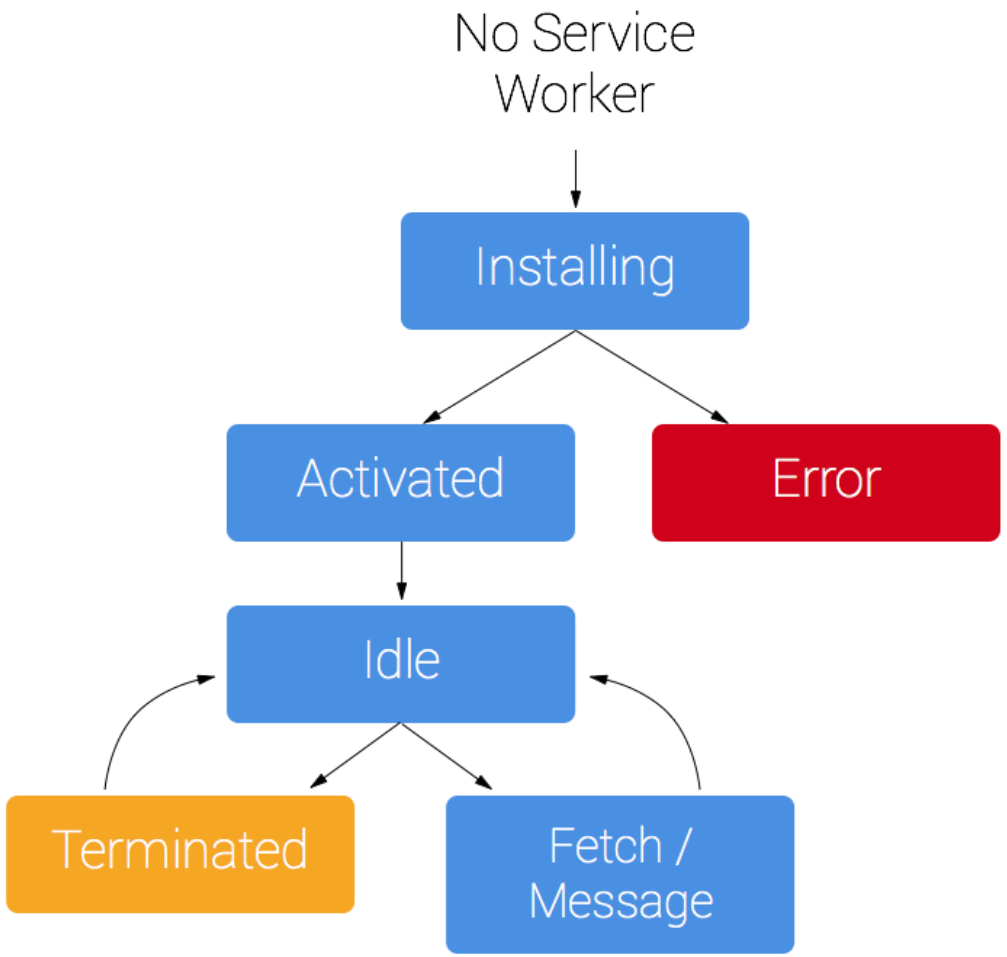
```
self.addEventListener('fetch', function (event) {
  event.respondWith(
    caches.match(event.request)
      .then(function (response) {
          if (response)
            return response;
          return fetch( event.request );
        })
      );
});
```

Service Worker életciklus

- Amikor először elindul az alkalmazásunk
  - Az eredeti címről letöltve
  - Még nincs telepítve SW
  - Böngésző elkezdi letölteni és telepíteni
    - El is indítja, amiről kapunk eseményt
      - Az SW-ben is (install esemény)
      - Az alkalmazásban is (registered promise)
- Ez a példány nem fog SW-t használni
- A következő indításkor viszont már igen

Service Worker életciklus

- A teljes életciklus így néz ki
  - Egy verzióra
- Amikor frissítjük
  - A régi verzió fut, amíg be nem záródik minden példánya az alkalmazásunknak
- Cache-re visszatérünk



Service Worker események

- Életciklus
  - install: telepítéskor és új verziónál
  - activate: működik, régi cache törölhető
- Cache
  - fetch: az oldal le akar tölteni valamit
- Kommunikáció
  - message: az alkalmazás küldött üzenetet
    - Ez az egyetlen mód a kommunikációra

Service Worker események

- Push notification
  - push: push notification jött
  - notificationclick: a felhasználó rányomott a notification-re
  - notificationclose: a felhasználó bezárta az üzenetet
  - pushsubscriptionchange: megszűnik a push notification feliratkozás
    - Bizonyos körülmények között újra feliratkozhatunk
- Egyelőre óvatosan kell használni
  - Mert iOS csak 16.4-től tudja

Cache

Alkalmazás oldal – Helyi tár

- Tárolhatunk adatot helyben
  - localStorage
  - IndexedDB
  - 10MB+ (akár GB feletti méretű is lehet)
- Nem feltétlen elég nagy képek és videók tárolására
  - De lehet, hogy a kezdő oldal tartalma belefér

Cache API

- Tipikusan SW-ben használjuk
- Minden függvény Promise alapú
- caches globális objektum
  - open függvénye megnyitja a megadott cache-t
  - Több cache is létezhet egyszerre
- Cache.addAll mindent letölt

```
self.addEventListener('install', function (event) {
  event.waitUntil(caches.open(CACHE_NAME)
    .then(function (cache) {
      return cache.addAll(urlsToCache);
    }));
});
```

Cache API

- Régi cache-t törölni kell, ha aktiválódott az új

```
self.addEventListener('activate', function (event) {
  event.waitUntil(
    caches.keys()
      .then(function (cacheNames) {
          return Promise.all(cacheNames
            .filter(function (cacheName) {
              return cacheName !== CACHE_NAME;})
            .map(function (cacheName) {
              return caches.delete(cacheName); }));
        })
    );
});
```

Cache stratégiák

- Nincsen mindig jó megoldás
- Minden letöltendő fájlról meg kell mondani
  - Benne legyen-e a cache-ben
  - Frissítsük-e
    - Ha igen, mikor
- A build tool összeállíthat egy fájllistát
- Dinamikusan előálló tartalom gond lehet
- Van pár tervezési minta – cache stratégiák
  - Lekérdezés típusonként kiválasztjuk a megfelelőt

Mindent cache-be

- Egyszerű, gyors

```
self.addEventListener( 'fetch', function ( event ){
  event.respondWith( caches.match( event.request ) );
} );
```

- Csak offline alkalmazásnál jó
- Minden fájlnak benne kell lennie
  - Dinamikus fájlok nem lehetnek
- Kezdő letöltés lassú, minden fájl kell
- Ha egy fájl nincs meg, akkor 404

Mindent hálózatról

- Egyszerű, lassú

```
self.addEventListener( 'fetch', function ( event ){
  event.respondWith( fetch( event.request ) );
} );
```

- Azonos működése van, mintha nem lenne
- Lassabb, mintha nem írtunk volna semmit
  - Át kell menni a hívásnak az SW-n
- Valóságban ezt sosem használjuk

Cache, majd hálózat

- Ha megvan, akkor nem töltjük le
- Az előző kettőnél több esetben használható
- Cache itt nem dinamikus
  - Ha valami nincs benne, akkor azt letöltjük, de nem adjuk hozzá
  - Ez bizonyos fájlok/kérések esetén fontos
    - Például állandóan változó hírlista

Hálózat, majd cache

- Ha nincs net, akkor a tárolt változatot adjuk vissza
- Cél
  - Offline is kéne működni
  - De ha van friss adat, akkor azt mutassuk
- Dinamikus cache
  - A cache-t frissíthetjük a letöltött adattal
- Hátrány: lassú hálózat belassít mindent

Hálózat és cache

- Elindítjuk mindkét lekérést
  - Cache előbb visszatér, azt visszaadjuk
  - Majd hálózat visszatér
    - Frissítjük cache-t
    - Értesítjük az appot, hogy van új adat
- Komplex megoldás
  - Az appnak is együtt kell működnie
- Egyszerűsítési lehetőség
  - Nem értesítjük az appot
  - Legközelebb már friss adatot adunk vissza

Kommunikáció

XHR

- Régi módszer
  - Az új verzió a fetch
- Http kérést indít egy szerver felé
- Aszinkron módon kap választ
- A kérés HTTP headerjeit részben írhatjuk
- A body-t teljesen egészében mi írjuk
  - Lehet JSON, XML, bináris, …
- A válasz headerjeit részben olvashatjuk
- A válasz body-t teljesen olvashatjuk

XHR

- Tömörítés
  - Támogatott: gzip, deflate, brotli (20%-kal jobb, mint gzip)
  - Ez a Content-Type headerben van jelezve és csak akkor jön, ha az Accept-Encoding headerben kérte a böngésző

```
let xhr = new XMLHttpRequest();
xhr.addEventListener( "load", () =>
  console.log( xhr.response ) );
xhr.open( "GET", "http://www.example.org/example.txt" );
xhr.send();
```

fetch

- XHR helyett – de nem tud mindent
  - Általában jó
- Promise-t ad vissza (lehet await-elni)

```
fetch( 'http://example.com/movies.json' )
    .then( ( response ) =>
    {
        return response.json();
    } )
    .then( ( data ) =>
    {
        console.log( data );
    } );
```

```
let res = await fetch( 'http://example.com/movies.json' );
let obj = await res.json();
```

WebSocket

- Üzenet alapú keretező protokoll
- Kétirányú csatorna jön létre
  - Nem szűnik meg csomagonként
  - Titkosítás azonos, mint HTTP esetén
- Kliens is és szerver is küldhet csomagot
  - Alacsony késleltetés
- Kicsi az overhead: max 8 bájt csomagonként
- Ezt használja számos klienst értesítő keretrendszer (pl. SignalR)

WebSocket

- Tömörítés van, de csak deflate (nincs brotli)
- Cache nincs
- A kapcsolódás egy HTTP kéréssel indul
  - A szerver visszaküldi, hogy OK, és nem zárja be a kapcsolatot
    - Az eredeti TCP kapcsolaton keresztül megy minden további kommunikáció
- Nem csak böngészőkben van implementálva
  - Két szerver is kommunikálhat WebSockettel
  - De botok nem támogatják

```
let ws = new WebSocket( "ws://echo.websocket.org/" );
ws.onmessage = e => console.log( e.data );
ws.onopen = () => this.webSocket.send( "Hello, World!" );
```

Share API

- navigator.share meghívja az OS beépített megosztás kezelőjét
  - Meg lehet adni: url, title, text
  - Csak mobilokon csinál valamit
  - Asztali OS-en copy-paste a megszokott mód
    - Esetleg valamilyen popupban listázhatjuk a szokásos dolgokat
- Feltehetjük az appunkat a megosztási listára
  - Az OS beépített megosztás kezelőjében ott lesz
  - manifest fájlba kell megadni, hogy mi történjen
  - El lehet kapni a SW-ben

Share API

- https://twitter.com/manifest.json részlet

```
"share_target": {
  "action": "compose/tweet",
  "enctype": "multipart/form-data",
  "method": "POST",
  "params": {
    "title": "title",
    "text": "text",
    "url": "url",
    "files": [
      {
        "name": "externalMedia",
        "accept": ["image/jpeg", "image/png", "image/gif", "video/quicktime", "video/mp4"]
      }
    ]
  }
}
```

Játékok

Piaci részesedés

- A legtöbb alkalmazás játék kategóriában van a piactereken
  - Mobilra igaz csak
  - Kevés a productivity alkalmazás
- Bevétel nagy része játék kategóriában van
  - Bizonyos piactereken 75% feletti a játék bevétel
  - PC-n árnyaltabb a helyzet
    - Itt is az egyik legnagyobb a játék költés
  - (Konzolon szinte csak az van)

Web alapú játékok

- Web alapú technológia elsősorban nem játékokra volt tervezve
  - HTML5 nagy előre lépés volt
  - Canvas és WebGL megjelent
- JavaScript sebesség problémák
  - Sokat javult a helyzet – közel C# sebesség
  - Köszönhető az optimalizáló fordítónak
  - Közel sem tartunk C++ sebességnél
- Eleinte egyszerűbb, ma már közepes grafikájú játékok is elfutnak

Web alapú játékok

- Megjelentek játékmotorok webes célplatformmal
  - Unity 3D
  - Unreal
  - WebGL-t és WebAssembly-t használnak
- Megjelentek 3D motorok JS-ben
  - Three.js
  - Babylon.js
- Nem csak grafika
  - Fizikai motorok, AI, video és audio támogatás

Canvas és 3D API

- <canvas> elem, amire renderelni lehet
- Működési módjai
  - 2d – lassabb, mint WebGL, de még így is sokkal gyorsabb, mint HTML elemekkel operálni
  - webgl – OpenGL ES 2.0 mód
  - webgl2 – OpenGL ES 3.0 mód
- navigator.gpu – WebGPU (Chrome/Edge 113+, Opera 99+)
  - Alacsony szintű 3D API
    - Mint Vulcan, vagy DirectX 12
  - Nincs köze OpenGL ES-hez

2d

- getContext-ben 2d-t adunk meg
- 2D rajzoló parancsok
  - rect, ellipse, drawImage, …

```
let context = canvas.getContext( "2d" );
context.rect( 0, 0, 100, 100 );
```

- Mindig az aktuális kitöltéssel és tollal rajzol
  - fillStyle és strokeStyle színekkel
- Nem törli automatikusan, nekünk kell
- A transzformációkat is megőrzi
  - Kényelmes

2d

- Általában képeket rajzolunk
  - Gond nélkül lehet több száz elemet rajzolni
  - De több ezer már problémás lehet 60 FPS-nél
- Felhasználói felületet (HUD) nem rajzolunk
  - HTML-ben oldjuk meg a canvas felett
  - Szöveget kiírhatunk canvasra is
- WebGL is alkalmas 2d-ben rajzolásra
  - Gyorsabb, kevésbé kényelmes
  - Vannak hozzá könyvtárak: PixiJS, TwoJS

requestAnimationFrame

- Bármikor lehet rajzolni
- Stabil, magas FPS-hez akkor rajzolunk, amikor böngésző amúgy is kiteszi a képet
- requestAnimationFrame pont ezt csinálja
- Általában 60 FPS
  - Mobilokon 30 FPS bizonyos esetekben
  - Erősebb mobilokon a képernyő frissítést tartja
    - 90, vagy akár 120 FPS
  - PC-ken is 60 fölé megy, ha a monitor engedi

WebAssembly

- Letölthető és futtatható .wasm bináris fájl
- Fordítók
  - Emscripten: C, C++
  - Blazor: C#, beleteszi a CLR-t is
- Nem tud hozzáférni a DOM-hoz
  - Egyenlőre, de talán soha…
- Jól elkülöníthető algoritmusokat lehet futtatni
- Nem feltétlen gyorsabb, mint JS
  - JS erősen optimalizált már
  - Gyorsabb, ha az adatformátum nem JS szerű

Kérdések?

Multiplatform szoftverfejlesztés

Teljesítmény, csomagolás és formátumok

Teljesítmény

Fogalmak és mérőszámok

Teljesítmény

- A szoftver
  - Letöltési/betöltési ideje
  - Parse-olás/JIT fordítási ideje
  - Futási ideje
  - Mérete
    - Ez erősen összefügg mindegyikkel
- Főleg a kiadott csomagnál fontos
  - De fejlesztés közben is szempont

Mérőszámok

- Első rajzolás (FP – first paint)
  - Amikor bármi változást látunk (pl. háttér)
- Első tartalom rajzolás (FCP – first contentful paint)
  - Amikor az első HTML-beli elemet látjuk
- DOMContentLoad esemény
  - Minden globális függvény lefutott
  - A HTML betöltve, a DOM felépítve
- Load esemény
  - Minden betöltve (CSS is)

Mérőszámok

- Oldal használható (TTI – time to interactive)
  - Látható az oldal és reagál 50 ms alatt
  - Ez a legutolsó és legfontosabb mérőszám
- Sokat idézett kutatás (Google ad network mérése alapján): Ha az oldal 3 másodperc alatt nem töltődik be, akkor a felhasználók 53%-a elhagyja a oldalt
  - Ezt nehéz megoldani, ha nem készülünk rá

Optimalizáció

- Caching – volt
- Gyors/kicsi könyvtárak használata
- Csomagolás (bundling)
  - Tree shaking
- Lazy loading
- Média formátumok (pl. webp)
- Tömörítés
- HTTP/2, HTTP/3

Csomagolás

Fordítás

- Fordítás (compiling, transpiling)
  - Kód átalakítása JS-re
- Például
  - .ts – TypeScript
  - .vue – Vue SFC (Single File Component)
  - .tsx és .jsx (React, vagy HTML template-et tartalmazó fájl)
- Gyors művelet
  - Forrás és cél formátum nagyon hasonló
  - Nincs szükség strukturális átalakításra
    - De szintaktikai ellenőrzés kell

Source map: app.js.map

- A fordítás/átalakítás miatt szükséges eltárolni, hogy adott kódrészlet hol volt az eredeti kódban
  - Célja a breakpoint támogatás, illetve más debug eszközök (soronként léptetés, függvénybe lépés, átugrás)
- Többszörös átalakítást támogatni kell
  - Például TS => JS => bundle

Bundling

- Több fájl egymás után másolása
  - Majd modulokra bontása
- Célok
  - Egyesítés után kevesebb fájlt kell letölteni
  - Modulok kialakítása, hogy ne egyben töltődjön le a teljes csomag
  - Általában: a kiadott csomag függetlenné tétele a fejlesztés alatt kialakított struktúrától
- Modul rendszert át kell alakítani
  - Vagy egy kimeneti fájl esetén meg kell szüntetni

Tree Shaking

- Felesleges kód eltávolítása – DCE (Dead Code Elimination)
  - Meg kell találni a nem használt függvényeket
  - Problémás osztályok esetén
  - Általában is nehéz, ha nem erre van optimalizálva egy könyvtár
    - Nem talál meg mindent
- Célok
  - A saját kódunk kisebbé tétele
  - A használt könyvtárak kisebbé tétele
    - Vagy akár teljes kiszűrése

Tree Shaking

- Sajnos a cél elérése nem garantált
  - Saját kódunkban általában kevés nem használt függvény van
  - Külső könyvtárak esetén lehet hasznos, ha úgy vannak megírva
  - Számos könyvtár jön testre szabható (customize) formában – én választom ki, hogy mi kell belőle
  - Példa: Pixi.js (2D render motor)
    - 370K a min.js verzió (1.2M az eredeti verzió)
    - Tree shaking szinte semmit sem tud kivágni
      - Az egyes modulok módosítják a belsejét (pl. Batch render)
    - 180K, ha kézzel válogatom be, ami kell

Minify: app.min.js

- Nevek lecserélése rövidre
- Célok
  - Kisebb fájl méret
    - Jelentős méretcsökkenés (1:3, 1:4)
    - Tömörítve nem annyira nagy a hatás, de még úgy is jelentős
  - Gyorsabb
    - Letöltés – cache esetén kicsi a hatása, de nem nulla
    - Betöltés – itt is számít a méret
  - Nehezebb olvasni (mellékhatás) – nehezíti a visszafejtést/megértést
    - Ha ez nem cél, akkor adunk mellé egy „fejlesztői” verziót, amiben jók a nevek

Csomagolás CSS – Minify

- CSS méretének csökkentése a felesleges whitespace-ek kivételével és akár átnevezéssel
- Célok – azonos a JS minifierrel
- Az átnevezés nem triviális, mert a HTML-t és JS-t is módosítani kell hozzá
  - Vannak eszközök rá
- Nehéz debuggolni
  - Nincs source map
- Kicsi a hatása, mert tömörítés után alig lesz kisebb és a CSS fájlok eleve kisebbek

Lazy loading

Modulok

- A legjobb módszer csomagokra bontani a kódot
  - Ez nem triviális feladat
  - Értelmes csomagokat kell kapjunk
  - Például hiába van szétszedve, ha a fő oldalhoz mindegyik kell
- Import és export használata
- Modul betöltő automatikusan megoldja
  - Fordító paramétere, hogy melyiket használjuk
  - Lehet használni a natív modulkezelőt
    - Ez már széleskörben támogatott

Modulok – kimeneti formátumok

- CommonJS – szerver oldal Node.js
- AMD – require.js
- UMD – mindkettő + globális változó
  - A fájl úgy kezdődik, hogy megnézi, hogy van-e require.js, és így megy tovább
- Natív ES modul

- A kimeneti formátumon nem működik tree-shaking

Modulok – natív modulok

- ES6-tól van
  - A böngészőre bízzuk a betöltést
  - Ez gyorsabb, mint a JS megoldások
  - Tree-shaking működik, ha további feldolgozásra van szükség
- Jelenleg csak Rollup tud ilyen kimenetet adni
- Hiába gyorsabb, több száz fájl esetén lassul
  - Nem publikálhatjuk az összes fájlt
  - Továbbra is kell csomagoló (Rollup)
    - Vagy nem használunk natív modulokat – ez a bevett módszer még

Külső könyvtárak

- Itt is működik a modul módszer
  - De lehetséges, hogy nincs így csomagolva
- Késleltetni lehet a betöltést
  - async és defer: aszinkron tölti a scriptet
    - Párhuzamosít, így javít a TTI-n
  - Manuálisan csak a gombnyomásra betölteni
  - Prediktív módszerek
    - Pl. elindítani a betöltést mouseover-re
      - Mobilon nem működik (touch)
    - Vagy amikor a gomb bejön a képernyőre
      - Ez jó mobilon

Képek és egyéb tartalom

- Ami nem látszik, azt lehet késleltetni
- Natív megoldás van, és már multiplatform (iOS 15.4-től)
  - loading="lazy"
- Fontos, hogy mit késleltetünk
  - Ha akkor döntjük el, amikor a JS már fut, az késő
  - Böngésző párhuzamosan tölt és futtat
  - Csak azt lehet késleltetni, amiről biztosan tudjuk, hogy nem kell a TTI-hez
    - Image carousel
    - Oldalon később jövő anyagok
    - …

Media formátumok

webp, webm, ...

Mindenhol működő formátumok

- JPEG
  - Veszteséges tömörítés, nincs átlátszóság
- PNG
  - Veszteségmentes tömörítés, van átlátszóság
- SVG
  - Vektorgrafikus ábra, vonal, spline, …
- TTF, OTF, WOFF
  - Font, vektorgrafikus, egyszínű, subpixel rendering támogatás
  - Főleg kicsi képek esetén fontos

Újabb formátumok

- WOFF2
  - Tudásban azonos, jobban tömörített (brotli)
- WebP
  - Erős tömörítés, átlátszóság, veszteséges/mentes
  - JPEG-hez képest 30-70%-kal kisebb képek azonos minőség mellet (másfajta tömörítési hibákat ad)
- WebM/VP9 codec (iOS nincs)
  - Videó
- AVIF (Edge nincs)
  - Még jobb tömörítés, mint WebP

picture – összes böngésző támogatása

- picture támogatás nélkül is megy
  - img fog csak érvényesülni

```
<picture>
    <source type="image/webp"
        srcset="images/p-5-256.webp 256w, images/p-5-512.webp 512w,
images/p-5-1024.webp 1024w"
        sizes="320px">
    <source
        srcset="images/p-5-256.jpg 256w, images/p-5-512.jpg 512w,
images/p-5-1024.jpg 1024w"
        sizes="320px">
    <img alt="" src="images/p-5-256.jpg">
</picture>
```

Tömörítés

- XHR esetén deflate vagy brotli
- WebSocket esetén csak deflate
  - Vagy bináris és mi magunk írunk tömörítőt
  - Például wasm-brotli
- Média webp/webm/stb.
- JS/CSS minified
  - És tömörítve jön le (brotli/deflate)
- HTTP/2

HTTP/2 és HTTP/3

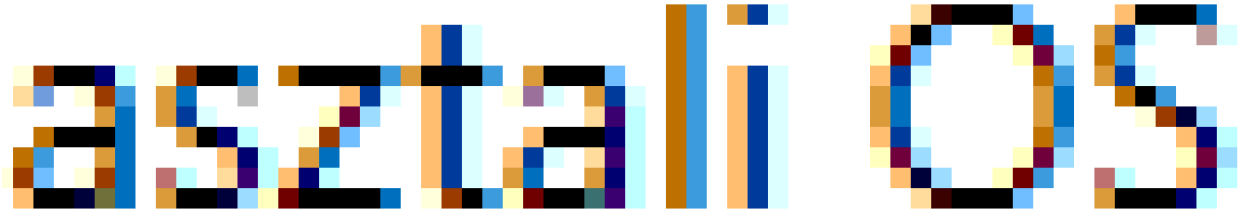
- HTTP/2
  - Push resource
  - Header tömörítés
  - Multiplexing – egy TCP csatorna több HTTP csomag (akár több kérés válasz előtt)
- HTTP/3
  - UDP alapú (UDP-QUIC-HTTP)
  - TLS 1.3 beépítve
  - Safari (macos és iOS) még nem támogatja általában

Betűtípusok

- Lehet saját betűtípust használni
  - Nem csak szöveg célból
  - Ikonokra is, pl. Font Awesome
- Formátumok
  - ttf, otf, woff (mindenhol), woff2
  - A különbség főleg méretben van, nem tudásban
  - Ezek mind spline-ból kirakott alakzatok, így tetszőleges méretben rajzolhatók
- Az ikonokat célszerű így tárolni a Subpixel rendering miatt
- Színes nem lehet, de adhatunk neki színt

Subpixel rendering

- Csak font rendereléskor támogatott, és csak asztali OS-eken



- Háromszorozza a vízszintes felbontást
  - Egy 100 dpi-s monitor 300 dpi-sként látszik
  - Függőleges felbontás marad, de arra nem vagyunk érzékenyek, főleg nem szövegnél
  - Vízszintesen is csak mozgatni lehet, nem lesz több pixel
- Nehéz megoldani, játékokban sajnos nincs
  - Canvas-on és bizonyos CSS animációk esetén kikapcsol

Subpixel rendering

- Ha eleve nagy a DPI, akkor nincs rá szükség
  - Például mobilok: 500+ DPI
  - 4K-s laptopok
- Attól is függ, hogy milyen távolról nézzük
  - PPD: Pixel per Degree
  - Mobil tipikusan 30 cm-re van
    - 600 DPI felett tökéletes kép
    - 300-600 DPI: egyesek kezdik látni
    - 300 DPI alatt mindenkinek feltűnik a probléma
  - Laptop 60 cm
  - PC 80-100 cm

Kérdések?