تغییرات NULL SAFETY

- تایپ ها به صورت دیفالت توی کد non-nullable هستن = متغیرها نمیتونن مقادیر null داشته باشن مگر اینکه ما به صورت واضح بهشون اجازه بدیم.

- ارورهای زمان اجرا مربوط به null تبدیل شدن به ارورهای مربوط به آنالیزور در زمان کدنویسی (edit-time) = سریعترین راه برای مشاهده و رفع هر مشکلِ مربوط به null

- هدف null safety حذف کامل مقدار null نیست
  - مقدار null همچنان در زبان دارت وجود داره
  - null عدم وجود یک مقدار رو برجسته میکنه
  - مشکل ما خود مقدار null نیست، بلکه مشکل داشتن مقدار null در جاییست که ما انتظارش رو نداریم

- هدف null safety اینه که کنترل داشته باشیم که کی، کجا و چگونه مقدار null توی برناممون وجود داشته باشه

- تایپ ها با استفاده از **پسوند علامت سوال (?)** به تایپ **null able** تبدیل میشن
مثال : **String? , int? , double? , num? , List<int?> , List<String?>? , ...**

- بعد از **null safety** قابلیت **implicit downcast** (تبدیل ضمنی تایپ والد به تایپ فرزند) کردن به صورت اتوماتیک توسط آنالیزور، **حذف شد**.

- شما **نمیتوانید** به **property** (ویژگی) و **method** های پایهِ یک **تایپ null able** به صورت عادی دسترسی داشته باشید.

مثال : **String? s; s.toUppercase();**

این کار **مجاز نیست** چون متغیر **s** ممکنه مقدار **null** داشته باشه

- به صورت عادی تنها **سه** ویژگی روی متغیرهای **null able** **قابل استفاده** هستن :
toString() , hashCode , ==

- در **null safety** تایپ **Object? والد** همهِ تایپ های دیگر است

- بعد از **null safety** تایپ **Never** به **انتهای درخت** تایپ ها (type tree) اضافه شد که در نتیجه **فرزند** همهِ تایپ های دیگه شد. این تایپ برای **قطع کردن جریان برنامه** با **throw** کردن یک **Exception** استفاده میشود.

- **توابعی** که خروجی آن ها **void** و **null able نیست**، همیشه باید نوع **non-nullable** مطابق با خروجی رو برگردونن.

- متغیرهای **non-nullable** **سراسری** و **استاتیک** در حالت عادی **همیشه** باید **در زمان تعریف مقدار دهیه اولیه شوند**

- متغیرهای **non-nullable** **کلاسی** در حالت عادی **همیشه** باید **قبل از بدنهِ تابع سازنده مقدار دهی اولیه شوند** (یا مقدار اولیه بگیرند یا با استفاده از this در آرگومان های ورودی سازنده یا با استفاده از initializer list مقدار دهی شوند)

- **پارامترهای اختیاریِ non-nullable** در **ورودی توابع**، حتما **باید مقداردهیِ پیش فرض شوند.**

مثال: **f ([String name = 'ali']){}** یا **f ({ int number = 0 }){}**

- **چهار تغییر قبل** که گفته شد برای متغیرهای **nullable** صادق نیستند و **میشه** اون ها رو **نادیده گرفت** = متغیرهای عادیِ **nullable نیاز به مقدار اولیه ندارند.**

- بعد از **null safety** توانایی **آنالیزور کنترل جریان دارت** به شدت **افزایش یافته.**
  - توانایی **Reachability analysis**
  - توانایی **Definite assignment analysis**
  - توانایی **Type promotion on null checks**
  - توانایی **Unnecessary code warnings**

- قابلیت **type promotion فقط** روی **متغیرهای محلی کار میکنه**

- **آنالیزور کنترل جریان دارت** (Control flow analysis) **بعد** از **null safety** میتونه بررسی کنه که آیا یک **متغیر عادی یا final محلی** قبل از استفاده مقدار دهی شده یا نه = میتونیم متغیرهای محلی از هر نوعی داشته باشیم **بدون اینکه نیاز به مقدار دهیه اولیه باشن**

- یک تایپ **nullable** میتونه به روش های مختلف به یک تایپ **non-nullable** **ارتقا پیدا کنه** :
  - استفاده از **if check** در **متغیرهای محلی**
  - استفاده از اپراتور **as** برای cast کردن
  - استفاده از **null aware operator (?)**
  - استفاده از **bang operator (!)**

- کلمه کلیدی **late** به ما این قابلیت رو میده که به آنالیزور بگیم این **متغیر کاملا امنه** و **100% قبل از استفاده مقدار دهی میشه** = میتونیم متغیرهای **non-nullable** ای داشته باشیم که **نیاز نباشه زمان تعریف مقدار دهیه اولیه بشن.**

- قابلیت دیگر کلمه کلیدی **late** **مقدار دهی به متغیرها به صورت lazy هست**. (lazy initialization)

- زمان استفاده از کلمه کلیدی **late** باید **100% مطمئن باشیم** که متغیر **قبل از استفاده مقدار دهی میشه** چون اگه این اتفاق نیوفته **به ارور زمان اجرا میخوریم**

- پارامترهای **Named non-nullable** در ورودی **توابع**، اگه ما نخوایم بهشون مقدار پیش فرض بدیم باید با استفاده از کلمه کلیدی **required** به صورت **اجباری** تعریف بشن.
مثال : **f ({ required String name , required int number }){}**