📘 Ripasso Git e Visual Studio

🔹 Git

Concetti principali sui branch

- Il primo branch che viene creato in una repository è `main`, che rappresenta la base stabile del progetto.
- Dal **main** si possono creare altri branch a seconda delle esigenze:
    - **Release branch** → è una "fotografia" stabile del main, che viene testata e corretta prima di andare al cliente.
    - **Feature branch** → è un ramo temporaneo creato per sviluppare una nuova funzionalità. Una volta completato e verificato, viene mergiato nel main.
    - **Hotfix branch** → usato per correggere velocemente bug critici. Dopo l'hotfix, il merge va fatto sia sul branch **release** (per consegnare una versione corretta al cliente) sia sul **main** (per mantenere la correzione nel codice futuro).

👉 La logica è: **main sempre stabile**, sviluppo su **feature**, rilascio con **release**, correzioni rapide con **hotfix**.

Comandi principali di Git

- `git clone <url>` → clona una repository remota in locale.
- `git checkout <branch>` → permette di cambiare branch o ripristinare un file a una versione precedente.
- `git branch` → elenca i branch esistenti; con `git branch <nome>` ne crea uno nuovo, con `-d` lo elimina, con `-m` lo rinomina.
- `git commit -m "messaggio"` → registra le modifiche nello storico locale.
- `git push` → invia i commit locali alla repository remota.
- `git fetch` → scarica gli aggiornamenti dal remoto ma non li integra.

- `git pull` → scarica e integra subito gli aggiornamenti dal remoto nel branch locale.
- `git stash` → salva temporaneamente le modifiche locali senza committarle. Molto utile se devi cambiare branch senza perdere il lavoro in corso.
- `git blame <file>` → mostra chi ha modificato ogni riga di un file.
- `git cherry-pick <commit>` → applica uno o più commit specifici in un altro branch.
- `git log` → visualizza la cronologia dei commit. Con `--oneline` diventa più leggibile.
- `git diff` → mostra le differenze tra file o commit.
- `git reset` → annulla modifiche locali e riporta allo stato di un commit precedente. Può essere "soft" (mantiene i file modificati) o "hard" (cancella tutto e torna esattamente al commit scelto).
- `git revert <commit>` → annulla un commit creando un nuovo commit che inverte le modifiche. È più sicuro di `reset` quando si lavora in team perché non riscrive la storia.

Strategie di merge

- **Fast forward merge** → unisce i branch spostando semplicemente il puntatore in avanti. Non crea un nuovo commit ed è usato quando non ci sono divergenze.
- **Three way merge** → serve quando i due branch hanno sviluppi paralleli. Git confronta tre versioni (base, locale, remoto). Se ci sono conflitti, lo sviluppatore deve risolverli manualmente.

Pull request

- Una **pull request (PR)** è una richiesta di unire le modifiche di un branch in un altro (es. da feature a release).
- Viene usata soprattutto nei team perché:
    - Permette di **revisionare** le modifiche (code review).
    - Può avere **regole** che definiscono chi può approvarla.
    - Garantisce che solo codice controllato finisca nel branch di destinazione.

🔹 Visual Studio

Struttura del progetto

- **Progetto** → è l'unità principale che contiene il codice (classi, dipendenze, file di configurazione). Può essere di diversi tipi:
    - **Application (exe)** → produce un file eseguibile con un **entry point** (`Main`).
    - **Class Library (dll)** → produce una libreria con classi e metodi riutilizzabili.
- **Classe** → file C# dove si scrive effettivamente il codice.
- **Assembly** → il risultato della compilazione del progetto. Se è una **application** produce un `.exe`, se è una **class library** produce un `.dll`.

Debugging

Il debugging serve per analizzare e trovare errori nel codice.
Strumenti principali:

- **Breakpoint** → punto nel codice dove l'esecuzione si ferma.
- **Step over** → esegue la riga senza entrare nei metodi.
- **Step into** → entra dentro il metodo chiamato.
- **Step out** → esce dal metodo corrente e torna al chiamante.
- **Watch variables** → osserva il valore delle variabili durante l'esecuzione.
- **Call stack inspection** → lista di tutti i metodi chiamati fino al punto in cui ci siamo fermati.

💡 Anche se avvio il progetto con **Start**, Visual Studio fa comunque prima la **build** (cioè compila il codice in `.exe` o `.dll`).

NuGet Package Manager

- Sistema di gestione delle librerie esterne.
- I pacchetti possono essere **pubblici** (dal feed ufficiale di NuGet) o **privati** (repository aziendale).
- Evita di dover scaricare manualmente codice da repository esterni.
- Permette di **riutilizzare librerie** in più progetti.

Differenza tra console application e class library

- **Console Application (exe)** → eseguibile, con un **entry point**. Può essere lanciato direttamente.
- **Class Library (dll)** → insieme di metodi e classi che non può essere eseguito da solo, ma deve essere referenziato in un altro progetto.

Differenza tra Debug e Release mode

- **Debug mode**:
    - Include file e simboli necessari al debug.
    - Permette di usare breakpoint, osservare variabili e call stack.
    - Non è ottimizzata per le prestazioni.
- **Release mode**:
    - Non include strumenti di debug.
    - Produce file più leggeri e ottimizzati.
    - È la modalità usata per distribuire il software al cliente.

👉 La differenza esiste perché i file di debug sono molto grandi e non devono essere consegnati agli utenti finali.

📘 Ripasso su .NET Framework e C#

🔹 .NET Framework

- Un **framework** è un insieme di **classi e librerie predefinite** che possiamo usare liberamente.
  → Ci semplifica il lavoro, perché molte funzionalità comuni sono già pronte.

- Ha anche una **gestione della memoria** integrata.

📌 .NET nello specifico

- **.NET** è sviluppato da **Microsoft**, nasce con il **C#** e viene usato soprattutto per applicazioni **Windows**.

- Si utilizza il **Framework** e non il **Core** perché storicamente è stato il più diffuso dagli anni 2000.

- Oltre alle librerie, include il **CLR (Common Language Runtime)** che gestisce:

  - **Garbage Collector** → libera la memoria da oggetti non più utilizzati.

  - **Gestione delle eccezioni**.

- È stato sviluppato in parallelo con **C#** per fare concorrenza a **Java**.

🔹 .NET Core

- È l'evoluzione del Framework.

- Comprende molte librerie di .NET Framework, ma è **cross-platform** (Windows, Linux, Mac).

- Quando parliamo di **.NET 5, 6, 7...** → si intende **.NET Core**.

- Anche il Core ha CLR e Garbage Collector.

📌 Componenti principali

- **Class Library** → input/output, collegamento a DB, gestione file, ecc.
- **CLR** → runtime per eseguire il codice, gestisce memoria ed eccezioni.

💡 In sintesi:

- **.NET Framework** → storico, solo Windows.
- **.NET Core / .NET 5+** → moderno, cross-platform.

🔹 Introduzione a C#

- Linguaggio orientato agli oggetti, pensato per scrivere codice **strutturato e riutilizzabile**.
- Nato con **.NET Framework**, ma con **.NET Core** può essere usato **ovunque**.

📌 Concetti base

- Ha **operatori**:
  - Aritmetici → `+`, `-`, `*`, `/`, `%`
  - Assegnazione → `=`, `+=`, `-=`...
  - Comparazione → `==`, `!=`, `<`, `>`
- Operatori di incremento:
  - `i++` → incrementa DOPO aver letto la variabile.
  - `++i` → incrementa PRIMA e poi legge la variabile.

🔹 Costrutti

- **Sequenze**: esecuzione lineare.
- **Selezione**: `if`, `switch`.

- **Iterazione (loop)**: `for`, `while`, `do while`.

♻️ Garbage Collector in .NET

Il **Garbage Collector (GC)** è un **meccanismo automatico di gestione della memoria** all'interno del **CLR (Common Language Runtime)**.
Il suo compito è **liberare la memoria occupata dagli oggetti che non sono più utilizzati** dal programma.

🔹 Come funziona

Quando crei un oggetto in C#, viene allocato nello **Heap** (memoria dinamica).
Esempio:

```
Persona p = new Persona();
```

1. → Qui l'oggetto `Persona` viene creato nell'**heap**, mentre la variabile `p` contiene solo il **riferimento (reference)**.

2. Se non ci sono più **riferimenti attivi** all'oggetto, significa che non può più essere usato.
   → A questo punto il Garbage Collector lo segna come "spazzatura" e lo rimuove dalla memoria.
3. Il GC lavora in automatico, ma può essere **forzato** (anche se sconsigliato) con:

   ```
   GC.Collect();
   ```

🔹 Generazioni del Garbage Collector

Per ottimizzare le performance, il GC divide gli oggetti in **generazioni**:

- **Generazione 0** → oggetti appena creati.
    - Raccolta molto frequente.
- **Generazione 1** → oggetti che sono "sopravvissuti" a una raccolta.
    - Raccolta meno frequente.

- **Generazione 2** → oggetti di lunga durata (es. variabili statiche, cache).
    - Raccolta ancora meno frequente.

💡 Questo approccio migliora l'efficienza: invece di controllare sempre tutta la memoria, il GC "si concentra" dove è più probabile trovare oggetti inutilizzati.

🔹 Vantaggi

- Non serve gestire manualmente la memoria (come in C o C++ con `malloc` e `free`).
- Riduce il rischio di **memory leaks** (perdita di memoria).
- Rende il codice più **sicuro e stabile**.

🔹 Svantaggi / Limiti

- Non hai **controllo totale**: il GC decide quando liberare la memoria.
- Può esserci un **pausa (stop the world)** quando il GC parte, perché sospende i thread per ripulire la memoria.
- Per applicazioni **real-time** (es. videogiochi o sistemi critici) questo può essere un problema → bisogna ottimizzare bene l'uso della memoria.

🔹 Tipi di dato in C#

📌 Value Types

- Il valore è salvato direttamente in memoria.

Esempio:

```
int a = 5;
int b = a;   // Ora ci sono due "5" distinti in memoria
```

📌 Reference Types

- La variabile contiene l'indirizzo di memoria dell'oggetto.
- Se copio una reference, entrambe le variabili puntano allo stesso oggetto.

🔹 Tipi numerici principali

- **int** → numeri interi, 32 bit.
- **double** → numeri decimali, 64 bit (precisione media).
- **decimal** → più preciso del double, ideale per valori finanziari.
- **float** → meno preciso del double.

🔹 Altri tipi

- **char** → singolo carattere Unicode, racchiuso da `'`.
- **string** → sequenza di caratteri (array di `char`).
    - Racchiusa da `"` doppi apici.
    - In C# le stringhe sono **immutabili** → ogni modifica crea una nuova stringa in memoria.

array → collezione di elementi dello stesso tipo, con **dimensione fissa** al momento della creazione:

```
int[] numeri = new int[5];
```

🔹 Loop Construct

- **for** → quando so quante volte ripetere.
- **while** → esegue finché la condizione è vera.
- **do while** → esegue almeno una volta e poi controlla la condizione.