

Avalonia solo in codice, con RAVAL

Programmare interfacce grafiche in C# senza XAML: dal primo programma ai grafici disegnati a mano

Ing. Roberto Arato

RAVAL 2.10.0 · Avalonia 12.1.4 · .NET 10 · novembre 2026

Indice

Prima di cominciare

- Come leggere questo libro

Parte I — Il linguaggio

1. C# in un capitolo
2. Più cose insieme: thread, Task e await

Parte II — Avalonia in codice

1. Il primo programma
2. Dove vanno i controlli: i pannelli
3. I controlli di tutti i giorni
4. Gli eventi, senza lambda
5. Chiaro e scuro: i temi
6. Elenchi e modelli
7. Menu e finestre di dialogo
8. Lavori lunghi: thread e Dispatcher

Parte III — Disegnare

1. Grafica disegnata a mano

Parte IV — I componenti di RAVAL

1. Le estensioni di RAva
2. Il progetto finale: un cruscotto
3. Un server di API JSON
4. Strumenti di misura: grafici, tabelle e stili pronti
5. Il log e le impostazioni
6. Gli alberi: JSON, XML e cartelle
7. Dialoghi e file in una riga
8. Appunti e trascinamento
9. Tabelle grandi e oggetti da modificare
10. Altri grafici: barre, mappe, carta di Smith e polare
11. Gli inciampi di Avalonia 12

Appendici

- A. Riferimento rapido
- B. Glossario
- C. Come sono fatte le figure

Come leggere questo libro

Questo libro è per chi sa già accendere un computer e scrivere qualche riga di codice, ma non ha mai costruito un programma con le finestre. Alla fine saprete scrivere un'applicazione grafica vera, che gira uguale su Windows, macOS e Linux, con i suoi menu, i suoi elenchi, i temi chiaro e scuro e i grafici disegnati da voi.

Usiamo tre ingredienti:

- **C#** su **.NET 10**: il linguaggio e la piattaforma. Il capitolo 1 riassume il linguaggio quanto basta.
- **Avalonia**: la libreria che disegna le finestre e i controlli. È gratuita, aperta, e non dipende dai controlli del sistema operativo: disegna tutto da sé con Skia, lo stesso motore grafico di Chrome.
- **RAVAL**, la libreria di questo tutorial (`RAVAL.dll`): aggiunge le **estensioni fluenti** che rendono breve il codice delle interfacce scritte senza XAML, e i pezzi pronti che quasi ogni programma tecnico si riscrive: un grafico a curve con i marcatori, una tabella, i numeri con i prefissi SI, le figure a 300 dpi, un server di API JSON. Contiene solo cose che servono a scrivere codice Avalonia.

Perché senza XAML

Quasi tutti i tutorial di Avalonia descrivono le finestre in XAML, un linguaggio a marcatori simile all'HTML, e collegano i dati con i *binding*. Funziona, ma per chi comincia vuol dire imparare due linguaggi, un compilatore che li unisce e una serie di errori che compaiono solo a programma avviato. In questo libro **tutto è C#**: la finestra si costruisce con `new`, le proprietà si assegnano, gli eventi si collegano a metodi. Il compilatore controlla ogni riga, l'editor completa i nomi, il debugger si ferma dove volete. È lo stesso modo in cui sono scritti i programmi veri dell'autore, compreso un intero programma di posta.

Le regole della casa

Il codice degli esempi segue le regole dell'autore, e le spiegheremo strada facendo:

- **niente lambda** (`x => ...`): gli eventi si collegano a **metodi con un nome**, i cicli si scrivono con `foreach`. Il codice è un po' più lungo ma si legge come un testo, ogni pezzo ha un nome che dice cosa fa, e nel debugger lo stack mostra nomi veri invece di `<Main>b__0_1`;
- **un commento su ogni funzione**, che spiega cosa fa e perché;
- **i nomi in italiano**, quando sono nostri; quelli di RAVAL sono in inglese, come quelli di Avalonia.

Gli esempi e le figure sono veri

Ogni esempio del libro è un file del progetto `RAVAL\ESEMPI_RAVAL`, e ogni figura che mostra una finestra è stata **disegnata da quel codice**: il programma apre le finestre senza schermo, ne cattura l'immagine e la salva in `DOC\immagini`. Anche i diagrammi sono disegnati in C#, con il controllo `Schema` che costruiremo nel capitolo 11. Se una figura non corrisponde al codice, è il codice che ha ragione: basta rigenerarla (appendice C).

Il codice di RAVAL e di tutti gli esempi si scarica da **https://raval.aratoingegneria.it** ed è sotto licenza **MIT**: potete usarlo nei vostri programmi, anche commerciali, conservando la nota di copyright.

Alla fine di ogni capitolo ci sono alcuni **esercizi**: fateli, è lì che si impara davvero.

Parte I — Il linguaggio

1. C# in un capitolo

> **Cosa imparerai:** com'è fatto un programma C#, i tipi, le decisioni e i cicli, le raccolte, i metodi; poi classi, oggetti, proprietà, eventi, ereditarietà, interfacce, record, null, generici, delegati, metodi d'estensione, `async` / `await` ed eccezioni. Sono tutte cose che useremo nei capitoli seguenti.

C# è un linguaggio **a oggetti** e **fortemente tipizzato**: ogni valore ha un tipo noto al compilatore, che vi ferma prima di sommare una data a un nome. Il codice si scrive in file `.cs`, organizzati in un **progetto** (`.csproj`), che il comando `dotnet build` compila in un programma.

Gli esempi di questo capitolo sono nel file `Esempi\LinguaggioCs.cs` e si eseguono con:

```
dotnet run --project ESEMPI_RAVAL -- --csharp
```

1.1 Variabili, tipi e stringhe

```
// tipi semplici, var, stringhe interpolate, if e cicli
private static void Variabili() {
  int pezzi = 12;                                  // intero
  double prezzo = 4.5;                             // numero con la
virgola
  decimal totale = 54.00m;                         // soldi: decimal, mai
double
  bool urgente = true;
  string cliente = "Officine Rossi";
  var data = new DateTime(2026, 10, 24);           // var: il tipo lo
deduce il compilatore
  Console.WriteLine($"{cliente}: {pezzi} pezzi a {prezzo:0.00} =
{totale:0.00} EUR, consegna {data:dd/MM/yyyy}");
  if (urgente && pezzi > 10) Console.WriteLine("ordine grosso e urgente");
  for (int i = 1; i <= 3; i++) Console.Write($"collo {i}  ");
  Console.WriteLine();
}
```

Le cose da notare:

- una variabile si dichiara con il **tipo** e il **nome**; `var` lascia dedurre il tipo, ma il tipo c'è lo stesso;
- `decimal` è per i soldi: `double` è binario e `0.1 + 0.2` non fa esattamente `0.3`;
- la **stringa interpolata** `$"..."` mette i valori fra graffe, con il formato dopo i due punti (`{prezzo:0.00}`, `{data:dd/MM/yyyy}`);
- `if`, `for`, `while` funzionano come in C e in Java; `&&` è «e», `||` è «o», `!` è «non».

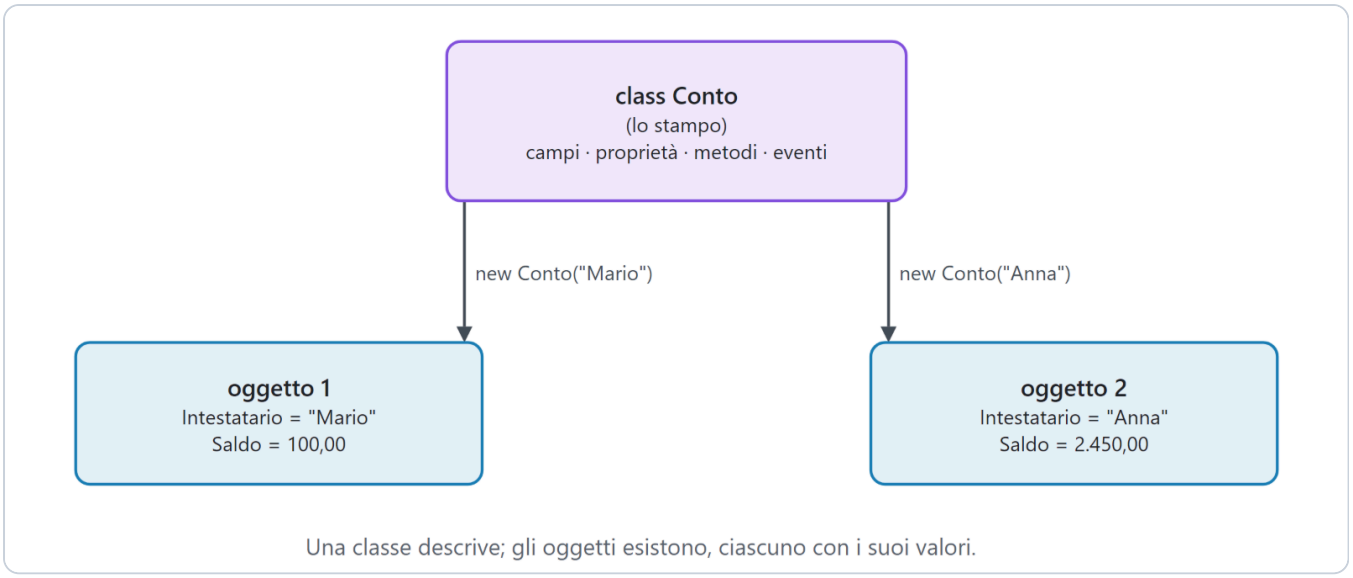
1.2 Liste e dizionari

```
// liste e dizionari, foreach
private static void Raccolte() {
  var citta = new List<string> { "Torino", "Milano", "Genova" };
  citta.Add("Bologna");
  foreach (string c in citta) Console.Write(c + " ");
  Console.WriteLine($"({citta.Count} citta')");
  var abitanti = new Dictionary<string, int> { ["Torino"] = 841_000,
["Genova"] = 558_000 };
  if (abitanti.TryGetValue("Torino", out int n)) Console.WriteLine($"Torino
ha {n:N0} abitanti");
}
```

`List<T>` è un elenco che cresce, `Dictionary<K, V>` associa una chiave a un valore. Fra le parentesi angolari c'è il **tipo degli elementi**: una `List<string>` contiene solo stringhe. `foreach` le percorre una per una. `TryGetValue` è il modo sicuro di leggere un dizionario: restituisce `false` se la chiave non c'è, invece di lanciare un errore.

1.3 Classi e oggetti

Una **classe** è uno stampo: descrive quali dati ha un oggetto (i **campi**), come li si legge (le **proprietà**), cosa sa fare (i **metodi**) e di cosa avvisa (gli **eventi**). Un **oggetto** è un esemplare costruito con `new`, con i suoi valori.



Una classe descrive; gli oggetti esistono, ciascuno con i suoi valori.

```
/// <summary>Un conto corrente: il saldo si cambia solo con Versa e Preleva.
</summary>
public sealed class Conto {

  private decimal _saldo;                                          // campo: lo
stato, nascosto

  /// <summary>Proprieta' in sola lettura dall'esterno.</summary>
  public string Intestatario { get; }

  /// <summary>Proprieta' calcolata: legge il campo.</summary>
  public decimal Saldo => _saldo;

  /// <summary>Evento: chi si iscrive viene avvisato a ogni movimento.
</summary>
  public event EventHandler<decimal>? Movimento;

  /// <summary>Costruttore: il conto nasce con un intestatario e saldo zero.
</summary>
  public Conto(string intestatario) { Intestatario = intestatario; }

  /// <summary>Aggiunge denaro; un importo non positivo e' un errore del
programma.</summary>
  public void Versa(decimal importo) {
    if (importo <= 0) throw new ArgumentOutOfRangeException(nameof(importo),
"l'importo deve essere positivo");
    _saldo += importo;
    Movimento?.Invoke(this, importo);                              // ?. : solo
se qualcuno ascolta
  }

  /// <summary>Toglie denaro se basta; restituisce false se non basta.
</summary>
  public bool Preleva(decimal importo) {
    if (importo > _saldo) return false;
    _saldo -= importo;
    Movimento?.Invoke(this, -importo);
    return true;
  }
}
```

- `private` nasconde: da fuori nessuno può scrivere `_saldo = 1000000`. È l'**incapsulamento**: lo stato cambia solo passando dai metodi, che controllano.

- `public string Intestatario { get; }` è una **proprietà** che si assegna solo nel costruttore.
- `Saldo => _saldo` è una proprietà **a espressione**: si calcola ogni volta che la si legge. (Non è una lambda, anche se usa la freccia: è solo un modo corto di scrivere un `get`.)
- `event` dichiara un **evento**: altri oggetti si iscrivono e vengono chiamati quando succede qualcosa. Avalonia è fatta tutta così: un pulsante ha l'evento `Click`.
- `sealed` dice che nessuno può ereditare da questa classe: è una scelta di prudenza.

Usare la classe:

```
// una classe con il suo evento
private static void Oggetti() {
  var conto = new Conto("Mario Rossi");
  conto.Movimento += MovimentoRegistrato;                        // ci si
iscrive con un metodo
  conto.Versa(100);
  bool ok = conto.Preleva(250);
  Console.WriteLine($"prelievo di 250: {(ok ? "fatto" : "rifiutato")}, saldo
{conto.Saldo}");
  try {
    conto.Versa(-5);
  } catch (ArgumentOutOfRangeException e) {
    Console.WriteLine("errore: " + e.Message.Split('\n')[0]);
  }
}

// Conto.Movimento
private static void MovimentoRegistrato(object? mittente, decimal importo) {
  if (mittente is Conto c) Console.WriteLine($"movimento di
{importo:+0.00;-0.00} sul conto di {c.Intestatario}");
}
```

`conto.Movimento += MovimentoRegistrato` iscrive **un metodo** all'evento: il metodo deve avere la firma giusta (`object? mittente, decimal importo`). È esattamente quello che faremo con i pulsanti. `try` / `catch` intercetta un'**eccezione**: l'errore lanciato con `throw` risale finché qualcuno lo prende.

1.4 Ereditarietà, interfacce e record

```
/// <summary>Qualcosa che sa descriversi a parole.</summary>
public interface IDescrivibile {
  /// <summary>La descrizione in una riga.</summary>
  string Descrivi();
}

/// <summary>Una figura geometrica: ogni figura calcola l'area a modo suo.
</summary>
public abstract class Figura : IDescrivibile {
  /// <summary>L'area, diversa per ogni figura.</summary>
  public abstract double Area();
  /// <summary>La stessa descrizione per tutte, con l'area di ciascuna.
</summary>
  public string Descrivi() => $"{GetType().Name} di area {Area():0.00}";
}

/// <summary>Un cerchio.</summary>
public sealed class Cerchio : Figura {
  public double Raggio { get; init; }
  public override double Area() => Math.PI * Raggio * Raggio;
}

/// <summary>Un record: un tipo di soli dati, confrontato per valore.
</summary>
public sealed record Punto(double X, double Y);
```

- Un'**interfaccia** è un contratto: «chi mi implementa ha il metodo `Descrivi`». Avalonia ne usa molte; nel capitolo 8 implementeremo `IDataTemplate`.
- Una classe **astratta** non si può costruire, serve da base. `Cerchio : Figura` **eredita**: ha tutto quello che ha `Figura` e riscrive (`override`) `Area`. Ogni controllo di Avalonia eredita da `Control`, e nel capitolo 11 scriveremo controlli nostri allo stesso modo.
- `init` permette di assegnare la proprietà solo durante la costruzione: `new Cerchio { Raggio = 1 }`. Questa sintassi fra graffe si chiama **inizializzatore di oggetto** e la useremo continuamente.
- Un **record** è una classe di soli dati, con il costruttore e l'uguaglianza per valore già pronti.

```
// lo stesso metodo, comportamenti diversi
private static void Polimorfismo() {
  var figure = new List<Figura> { new Cerchio { Raggio = 1 }, new Rettangolo
{ Base = 3, Altezza = 2 } };
  foreach (Figura f in figure) Console.WriteLine(f.Descrivi());
  Console.WriteLine(new Punto(1, 2) == new Punto(1, 2) ? "due record uguali"
: "diversi");
  Console.WriteLine("avalonia".Maiuscola());
}
```

1.5 null, generici e delegati

```
// null: ?, ?. e ??
private static void Nulli() {
  string? nota = null;                                          // ? : puo'
non esserci
  Console.WriteLine($"lunghezza della nota: {nota?.Length ?? 0}");
  nota = "da richiamare";
  Console.WriteLine($"lunghezza della nota: {nota?.Length ?? 0}");
}

// un metodo generico: vale per ogni tipo confrontabile
private static T Massimo<T>(IEnumerable<T> valori) where T : IComparable<T>
{
  bool primo = true;
  T meglio = default!;
  foreach (T v in valori) {
    if (primo || v.CompareTo(meglio) > 0) meglio = v;
    primo = false;
  }
  return meglio;
}

// Func e Action: un metodo passato come valore (gruppo di metodi, senza
lambda)
private static void Delegati() {
  Func<double, double> calcolo = Quadrato;
  Action<string> stampa = Console.WriteLine;
  stampa($"il quadrato di 7 e' {calcolo(7)}");
}

// il metodo passato a Func
private static double Quadrato(double x) => x * x;
```

- `string?` dice che la variabile **può essere null** (vuota). `nota?.Length` vale null se `nota` è null, invece di andare in errore; `?? 0` sostituisce il null con 0. Il compilatore vi avvisa quando usate un valore che potrebbe essere null senza controllarlo.
- Un metodo **generico** (`Massimo<T>`) funziona con tipi diversi: qui interi e stringhe.
- Un **delegato** è un metodo trattato come un valore. `Action` è un metodo senza risultato, `Func<A, R>` uno che riceve `A` e restituisce `R` . Si assegnano con il **nome del metodo** (gruppo di metodi): `Func<double, double> calcolo = Quadrato;` . Molti tutorial scriverebbero `x => x * x` : noi diamo un nome anche a questo, e lo faremo con tutti gli eventi di Avalonia.

1.6 Metodi d'estensione

```
/// <summary>Metodi d'estensione sulle stringhe: lo stesso meccanismo di
RAva sui controlli.</summary>
public static class EstensioniTesto {
  /// <summary>La prima lettera maiuscola.</summary>
  public static string Maiuscola(this string s) => s.Length == 0 ? s :
char.ToUpper(s[0], CultureInfo.CurrentCulture) + s[1..];
}
```

La parola `this` davanti al primo parametro trasforma un metodo statico in un **metodo d'estensione**: si chiama come se fosse della stringa, `"avalonia".Maiuscola()` . È il trucco su cui si regge RAva: metodi come `Margin` , `Children` , `Text` si «aggiungono» a tutti i controlli di Avalonia senza toccarli, e restituendo il controllo stesso permettono di scrivere in catena: `new TextBlock().Text("Ciao").Margin(8)` .

1.7 async e await

```
// async/await: si aspetta senza bloccare il thread
private static async Task Asincrono() {
  Console.WriteLine("scarico...");
  string risultato = await ScaricaFinto();
  Console.WriteLine(risultato);
}

// finge un'operazione lunga (rete, disco)
private static async Task<string> ScaricaFinto() {
  await Task.Delay(200);
  return "scaricato dopo 200 ms";
}
```

Un metodo `async` può **aspettare** (`await`) un'operazione lunga senza bloccare il thread: nel frattempo il programma fa altro. In un'applicazione grafica è fondamentale, perché il thread dell'interfaccia deve restare libero di ridisegnare la finestra. `Task<string>` è «una stringa che arriverà».

Qui è solo un assaggio: thread, `Task` , `async` e `await` hanno un capitolo tutto loro, il prossimo, con esempi da eseguire e diagrammi. Chi non li ha mai usati può cominciare dai thread (§2.3) e dai Task senza `await` (§2.6), e arrivare all'`await` dopo.

1.8 Il risultato

Ecco cosa scrive il programma degli esempi:

```
Officine Rossi: 12 pezzi a 4,50 = 54,00 EUR, consegna 24/10/2026
ordine grosso e urgente
collo 1  collo 2  collo 3
Torino Milano Genova Bologna (4 citta')
Torino ha 841.000 abitanti
movimento di +100,00 sul conto di Mario Rossi
prelievo di 250: rifiutato, saldo 100
errore: l'importo deve essere positivo (Parameter 'importo')
Cerchio di area 3,14
Rettangolo di area 6,00
due record uguali
Avalonia
lunghezza della nota: 0
lunghezza della nota: 13
massimo: 17 e zucca
il quadrato di 7 e' 49
scarico...
scaricato dopo 200 ms
```

Notate la virgola decimale e il punto delle migliaia: il formato segue la **cultura** del sistema (qui italiana).

Esercizi

1. Aggiungete a `Conto` un metodo `Giroconto(Conto destinazione, decimal importo)` che usa `Preleva` e `Versa` .
2. Scrivete una classe `Triangolo : Figura` e aggiungetela all'elenco di `Polimorfismo` .
3. Scrivete un metodo d'estensione `Iniziali(this string nome)` che da `"Mario Rossi"` restituisce `"MR"` .

2. Più cose insieme: thread, Task e await

> **Cosa imparerai:** cos'è un thread e come se ne avvia uno; perché due thread che toccano lo stesso dato sbagliano il conto, e come si rimedia con `lock`; come un thread aspetta un segnale; cosa sono il pool e i `Task`; cosa fanno davvero `async` e `await`; perché in un programma con finestre il codice dopo un `await` torna sul thread dell'interfaccia, e come questo può bloccare tutto; come si gestiscono errori e annullamenti. Tutto con esempi che girano: `dotnet run --project ESEMPI_RAVAL -- --thread`.

Questo è il capitolo più importante della parte sul linguaggio. Un programma con finestre fa sempre più cose insieme: ridisegna, risponde al mouse, legge un file, parla con una macchina, aspetta la rete. Se le fa una dopo l'altra, la finestra si congela. C# offre tre strumenti, nati in epoche diverse, che qui vediamo in ordine: i **thread** (i più vecchi e i più semplici da capire), i **Task** (il lavoro mandato al pool) e **async/await** (il modo moderno di aspettare senza bloccare). Si può scrivere un programma vero con ognuno dei tre, e più avanti useremo tutti e tre.

2.1 Perché più cose insieme

Un **thread** è una sequenza di istruzioni eseguita dal processore una dopo l'altra. Ogni programma parte con un thread, quello che esegue `Main`. Se un'istruzione aspetta (un file lento, una risposta dalla rete, un `Thread.Sleep`), il thread è fermo lì e non fa nient'altro.

In un programma Avalonia il thread di `Main` diventa il **thread dell'interfaccia**: è l'unico che può toccare i controlli, e tra un evento e l'altro ridisegna la finestra. Se un gestore del clic lavora per cinque secondi, per cinque secondi la finestra non si ridisegna e non risponde: il sistema operativo la segna come «non risponde». Per questo i lavori lunghi vanno **da un'altra parte**.

2.2 Un thread solo: una cosa dopo l’altra

Prendiamo un lavoro finto: tre passi da 100 ms, ciascuno stampa il nome dell'operaio, il thread e l'ora.

```
/// <summary>Un lavoro con un nome: tre passi da 100 ms. Il nome viaggia nel
campo, non in una lambda.</summary>
public sealed class Operaio {
  private readonly string _nome;
// chi lavora
  /// <summary>L'operaio con il suo nome.</summary>
  public Operaio(string nome) { _nome = nome; }
  /// <summary>Il lavoro: il metodo che il thread esegue.</summary>
  public void Lavora() {
    for (int passo = 1; passo <= 3; passo++) {
      Thread.Sleep(100);
// finge un lavoro di 100 ms
      Console.WriteLine($"   {_nome}{passo}  (thread
{Environment.CurrentManagedThreadId}, {Orologio.ElapsedMilliseconds} ms)");
    }
  }
}
```

Due operai sullo stesso thread lavorano uno dopo l'altro:

```
new Operaio("A").Lavora();
new Operaio("B").Lavora();
```

```
   A1  (thread 2, 114 ms)
   A2  (thread 2, 221 ms)
   A3  (thread 2, 330 ms)
   B1  (thread 2, 437 ms)
   B2  (thread 2, 546 ms)
   B3  (thread 2, 654 ms)
   fatto in 654 ms
```

Sei passi da 100 ms: circa 600 ms, tutti sul thread 2.

2.3 Due thread: due cose insieme

`new Thread(metodo)` crea un thread che eseguirà quel metodo; `Start()` lo fa partire; `Join()` aspetta che finisca. Il metodo è un metodo **con un nome**, senza argomenti: i dati che gli servono stanno nei campi del suo oggetto. È per questo che `Operaio` è una classe: `new Operaio("A").Lavora` è «il metodo `Lavora` di quell'operaio», e porta con sé il nome. È la stessa tecnica che usiamo per gli eventi (capitolo 6), e ci risparmia le lambda.

```
var a = new Thread(new Operaio("A").Lavora);
var b = new Thread(new Operaio("B").Lavora);
a.Start();
b.Start();
a.Join();
// aspetta che A finisca
b.Join();
// e poi B
```

```
   A1  (thread 4, 762 ms)
   B1  (thread 5, 762 ms)
   B2  (thread 5, 872 ms)
   A2  (thread 4, 872 ms)
   B3  (thread 5, 981 ms)
   A3  (thread 4, 981 ms)
   fatto in 327 ms
```



Tre cose da notare:

- il tempo è dimezzato: i due lavori sono andati avanti **insieme**, ciascuno sul suo thread (4 e 5);
- l'ordine delle righe **cambia** da un'esecuzione all'altra (qui B2 è prima di A2): due thread non si aspettano a vicenda, e chi arriva prima scrive prima;
- `Join` è il modo più semplice di aspettare: il thread che lo chiama si ferma finché l'altro non ha finito.

Un thread ha anche la proprietà `IsBackground`: un thread «di sfondo» non tiene vivo il programma, che si chiude quando finisce `Main` anche se lui sta ancora lavorando; uno normale (di primo piano) sì. I thread che fanno da servizio a una finestra conviene crearli di sfondo:

```
var lettore = new Thread(macchina.LeggiSempre) { IsBackground = true, Name =
"lettura macchina" };
lettore.Start();
```

Il `Name` compare nel debugger, nella finestra dei thread: con dieci thread aperti, è l'unico modo di capire chi è chi.

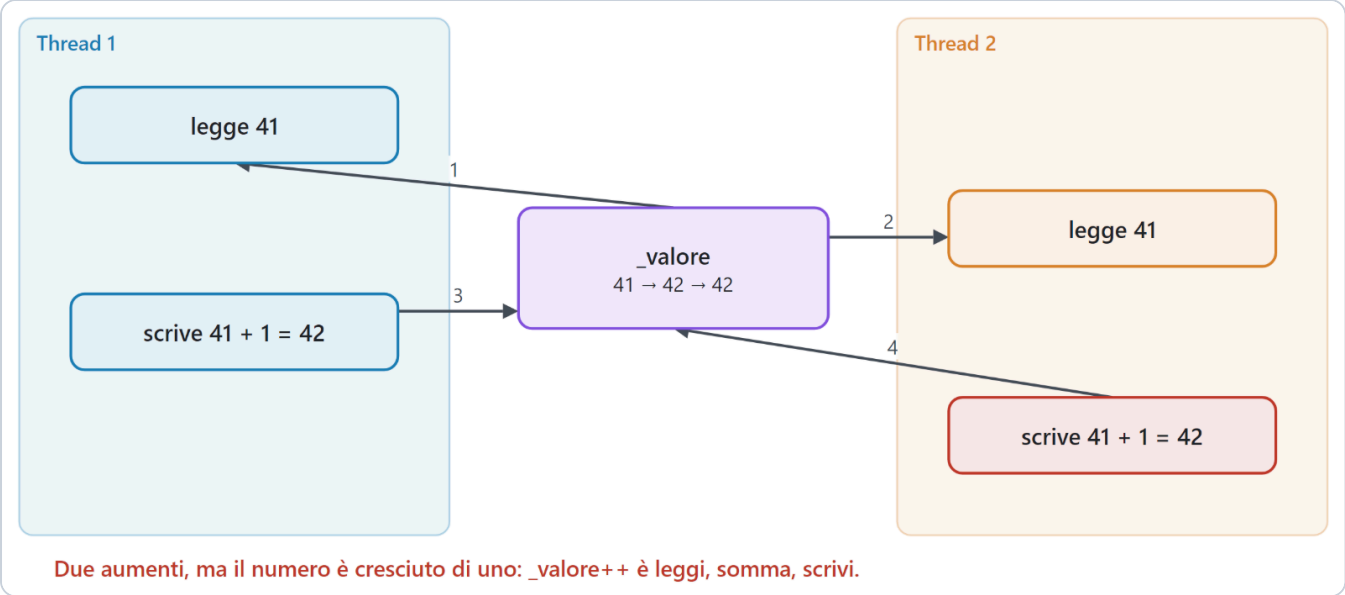
2.4 Dati condivisi: il conto sbagliato e il lock

Due thread che leggono dati diversi non si danno fastidio. Due thread che **scrivono lo stesso dato** sì. Proviamo: due thread aumentano lo stesso numero un milione di volte ciascuno.

```
/// <summary>Un numero condiviso fra thread, aumentato in tre modi.
</summary>
public sealed class Contatore {
  private readonly object _chiave = new();
// il «lucchetto» del lock
  private int _valore;
// il numero condiviso
  /// <summary>Il valore attuale.</summary>
  public int Valore => _valore;
  /// <summary>Sbagliato: _valore++ e' leggi, somma, scrivi; due thread
possono leggere lo stesso numero.</summary>
  public void AumentaSenzaLock() { for (int i = 0; i < 1_000_000; i++)
_valore++; }
  /// <summary>Giusto: dentro il lock entra un thread per volta.</summary>
  public void AumentaConLock() { for (int i = 0; i < 1_000_000; i++) lock
(_chiave) _valore++; }
  /// <summary>Giusto e piu' veloce per un numero solo: l'aumento e'
un'operazione unica del processore.</summary>
  public void AumentaInterlocked() { for (int i = 0; i < 1_000_000; i++)
Interlocked.Increment(ref _valore); }
}
```

```
   senza lock:  1.044.811  (dovrebbe essere 2.000.000)
   con lock:    2.000.000
   Interlocked: 2.000.000
```

Senza protezione si perde quasi metà degli aumenti, e il numero cambia ogni volta. Il motivo è che `_valore++` non è un'operazione sola: è **leggi** il valore, **somma** uno, **scrivi** il risultato. Se i due thread leggono lo stesso 41 prima che uno dei due scriva, entrambi scrivono 42: due aumenti, uno perso.



Si chiama **corsa** (*race condition*), ed è il difetto più insidioso della programmazione con i thread: non dà errori, i conti sbagliano solo ogni tanto, e quasi mai mentre si prova. I rimedi:

- `lock (oggetto) { ... }`: dentro il blocco entra un thread per volta; gli altri aspettano fuori. L'oggetto è un «lucchetto» privato (`private readonly object _chiave = new();`), sempre lo stesso per lo stesso dato. Dentro il lock si fa il minimo indispensabile e **mai** un'attesa lunga;
- `Interlocked`: per un numero solo (aumentare, diminuire, scambiare) c'è un'istruzione del processore che lo fa in un colpo, senza lucchetto;
- **non condividere**: il rimedio migliore. Ogni thread lavora sui suoi dati e consegna il risultato alla fine (`Join`, un segnale, un `Task<T>`). È quello che faremo quasi sempre.

2.5 Aspettare un segnale

`Join` aspetta che un thread **finisca**. Spesso invece serve aspettare che un thread arrivi a un certo punto: «i dati sono pronti», «il server è in ascolto». Si usa un **segnale**: `ManualResetEventSlim` è una lampadina, spenta all'inizio; `Set()` la accende, `Wait()` aspetta che sia accesa.

```
/// <summary>Prepara un piatto in un altro thread e accende Pronto quando ha
finito.</summary>
public sealed class Cucina {
  /// <summary>Il segnale: spento all'inizio, acceso da Prepara.</summary>
  public ManualResetEventSlim Pronto { get; } = new(false);
  /// <summary>Il risultato, valido dopo il segnale.</summary>
  public string Piatto { get; private set; } = "";
  /// <summary>Il lavoro del thread.</summary>
  public void Prepara() {
    Thread.Sleep(300);
    Piatto = "pasta al pomodoro";
    Pronto.Set();
// sveglia chi aspetta
  }
}
```

```
var cucina = new Cucina();
new Thread(cucina.Prepara).Start();
Console.WriteLine("   aspetto il segnale...");
cucina.Pronto.Wait();
// fermo qui finche' Set
Console.WriteLine($"   arrivato: {cucina.Piatto}");
```

Il server HTTP di RAVAL (capitolo 14) funziona proprio così: `Run()` avvia il server e aspetta un segnale che `Stop()`, chiamato da un altro thread, accende. Mai aspettare con `Thread.Sleep(1000)` «che sia pronto»: o si aspetta troppo, o troppo poco.

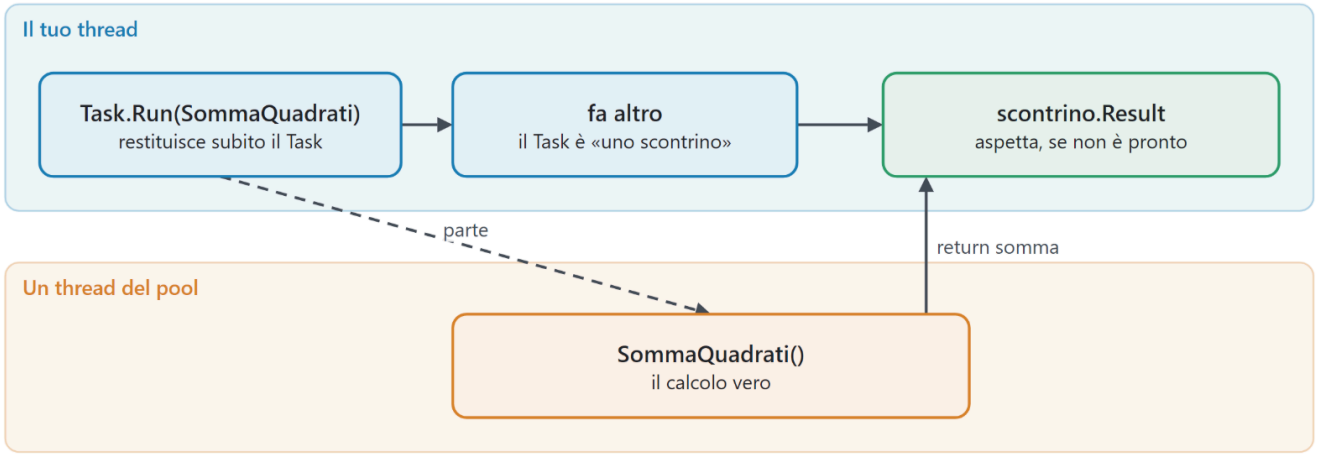
2.6 Il pool e i Task, senza await

Creare un thread costa: memoria per la sua pila, tempo per avviarlo. Per i lavori brevi .NET tiene pronto un **pool**, un gruppo di thread riutilizzabili. Non si usano direttamente: si consegna il lavoro con `Task.Run(metodo)`, che lo mette in coda e restituisce subito un **`Task`**.

Un `Task` è come lo **scontrino** al bancone: lo ricevi subito, il piatto arriva dopo. `Task<long>` è lo scontrino di un `long`. Con lo scontrino in mano si può fare altro, e quando serve il risultato si chiede `Result`: se è pronto arriva subito, altrimenti il thread aspetta lì.

```
Task<long> scontrino = Task.Run(SommaQuadrati);
Console.WriteLine($"   il Task e' partito; intanto il thread
{Environment.CurrentManagedThreadId} fa altro");
long somma = scontrino.Result;
// qui si ferma finche' non e' pronto
Console.WriteLine($"   risultato: {somma:N0}");
```

```
   il Task e' partito; intanto il thread 2 fa altro
   calcolo sul thread 13
   risultato: 40.000.002
```



Più Task insieme si aspettano con `Task.WaitAll`:

```
Task a = Task.Run(Attendi300);
Task b = Task.Run(Attendi300);
Task c = Task.Run(Attendi300);
Task.WaitAll(a, b, c);
```

```
   tre attese da 300 ms in 308 ms
```

Fin qui niente `async` e niente `await`: `Task.Run` più `Result` o `Wait` è già un modo completo di lavorare in parallelo, ed è quello che conviene usare finché `await` non è chiaro. **Una regola però vale da subito: `Result` e `Wait()` fermano il thread che li chiama**, quindi nel thread dell'interfaccia vanno usati solo per attese brevissime. Per le altre serve `await`, o un thread con il Dispatcher (capitolo 10).

2.7 async e await

`await` è il modo di aspettare un `Task` **senza fermare il thread**. Un metodo che usa `await` si dichiara `async` e restituisce `Task` (o `Task<T>` se ha un risultato):

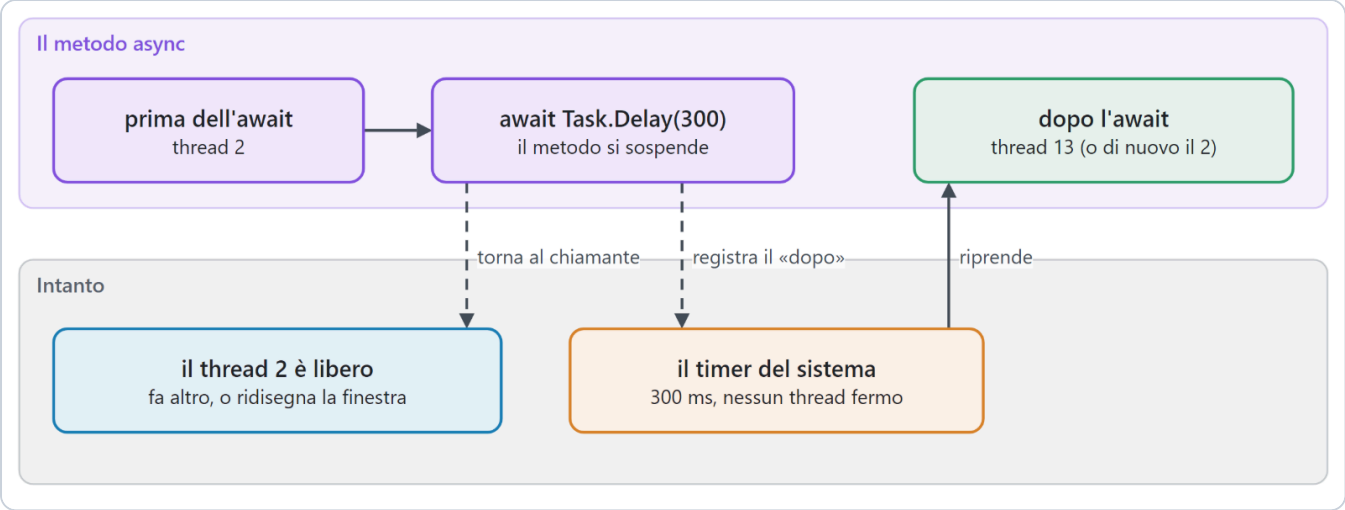
```
// aspetta 300 ms senza occupare nessun thread, poi restituisce il risultato
private static async Task<string> PreparaCaffeAsync() {
  await Task.Delay(300);
// nessun thread e' fermo qui
  return "caffe' pronto";
}

/// <summary>await libera il thread mentre aspetta; il resto del metodo
riprende dopo, anche su un altro thread.</summary>
public static async Task AsyncAwait() {
  Console.WriteLine($"   prima dell'await: thread
{Environment.CurrentManagedThreadId}");
  string caffe = await PreparaCaffeAsync();
  Console.WriteLine($"   dopo l'await:     thread
{Environment.CurrentManagedThreadId}  -> {caffe}");
}
```

```
   prima dell'await: thread 2
   dopo l'await:     thread 13  -> caffe' pronto
```

Cosa succede, passo per passo:

1. il metodo esegue normalmente fino all'`await`;
2. all'`await`, se il Task non è ancora finito, il compilatore **divide il metodo in due**: registra «il resto del metodo» come cosa da fare quando il Task finisce, e **torna al chiamante**. Il thread è libero;
3. mentre si aspetta, **nessun thread è fermo**: `Task.Delay` è un timer del sistema operativo, la lettura di un file o della rete è il disco o la scheda di rete che lavorano da soli;
4. quando il Task finisce, «il resto del metodo» riprende. In un programma console riprende su un thread qualsiasi del pool (qui il 13), in un programma con finestre sul thread dell'interfaccia (§2.8).



È questa la differenza con `Result`: `Result` **ferma** il thread finché il Task non finisce; `await` **libera** il thread e fa riprendere il metodo dopo. Per questo `await` è il modo giusto di aspettare nel thread dell'interfaccia.

Due regole pratiche:

- **`async` è contagioso.** Un metodo che usa `await` diventa `async` e restituisce un `Task`; chi lo chiama, per aspettarlo senza bloccarsi, usa `await` a sua volta, e così via fino in cima: in un programma con finestre la cima è un gestore di evento, che può essere `async void` (l'unico caso in cui `async void` va bene). In un `Main` console si può scrivere `static async Task Main()`;
- **il nome finisce con `Async`** (`CaricaAsync`, `StartAsync`): chi legge sa che restituisce un Task da aspettare.

Uno dopo l'altro o insieme. `await` aspetta un Task alla volta. Due `await` di seguito sono due attese in fila; per farle insieme si fanno **partire** entrambe e poi si aspettano con `Task.WhenAll`:

```
string a = await PreparaCaffeAsync();
string b = await PreparaCaffeAsync();
// parte quando a e' finito

Task<string> primo = PreparaCaffeAsync();
// parte subito
Task<string> secondo = PreparaCaffeAsync();
// parte subito anche lui
string[] tutti = await Task.WhenAll(primo, secondo);
```

```
   uno dopo l'altro: 616 ms (caffe' pronto, caffe' pronto)
   insieme:          314 ms (2 caffe')
```

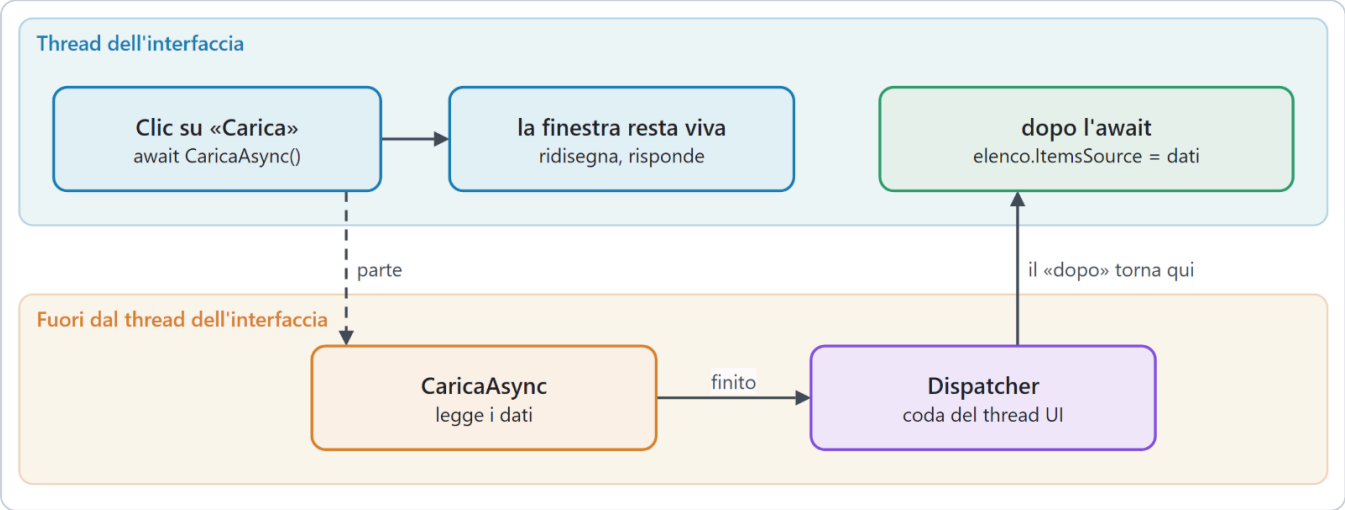
`WhenAll` è il `WaitAll` dell'`await`: aspetta tutti senza fermare il thread.

2.8 Il thread dell'interfaccia e il contesto

In un programma con finestre `await` fa una cosa in più, ed è quella che lo rende comodo: **dopo l'`await` il metodo riprende sul thread dell'interfaccia**, quindi può toccare i controlli.

```
// il gestore del clic: async void, l'unico caso in cui va bene
private async void CaricaClic(object? mittente, RoutedEventArgs e) {
  stato.Text = "carico...";
  List<string> dati = await CaricaAsync();
// la finestra resta viva
  elenco.ItemsSource = dati;
// di nuovo sul thread UI
  stato.Text = dati.Count + " righe";
}
```

Come fa? Prima di dividere il metodo, `await` guarda il **contesto di sincronizzazione** del thread (`SynchronizationContext.Current`). Il thread dell'interfaccia di Avalonia ne ha uno, che sa mettere un lavoro nella coda del **Dispatcher**: «il resto del metodo» viene messo lì, e il thread dell'interfaccia lo esegue appena è libero, come un evento qualsiasi. Il thread di una console non ha un contesto: per questo nell'esempio di §2.7 il resto del metodo riprendeva sul thread 13.



Chi lavora con i thread invece del `await` fa la stessa cosa a mano: il thread di lavoro non tocca i controlli, ma chiede al Dispatcher di eseguire un metodo sul thread dell'interfaccia.

```
// in un thread di lavoro: i controlli si toccano solo attraverso il
Dispatcher
private void Leggi() {
  _dati = macchina.LeggiTutto();
// lavoro lungo, fuori dall'interfaccia
  Dispatcher.UIThread.Post(Mostra);
// Mostra girera' sul thread UI
}

// sul thread dell'interfaccia
private void Mostra() { elenco.ItemsSource = _dati; }
```

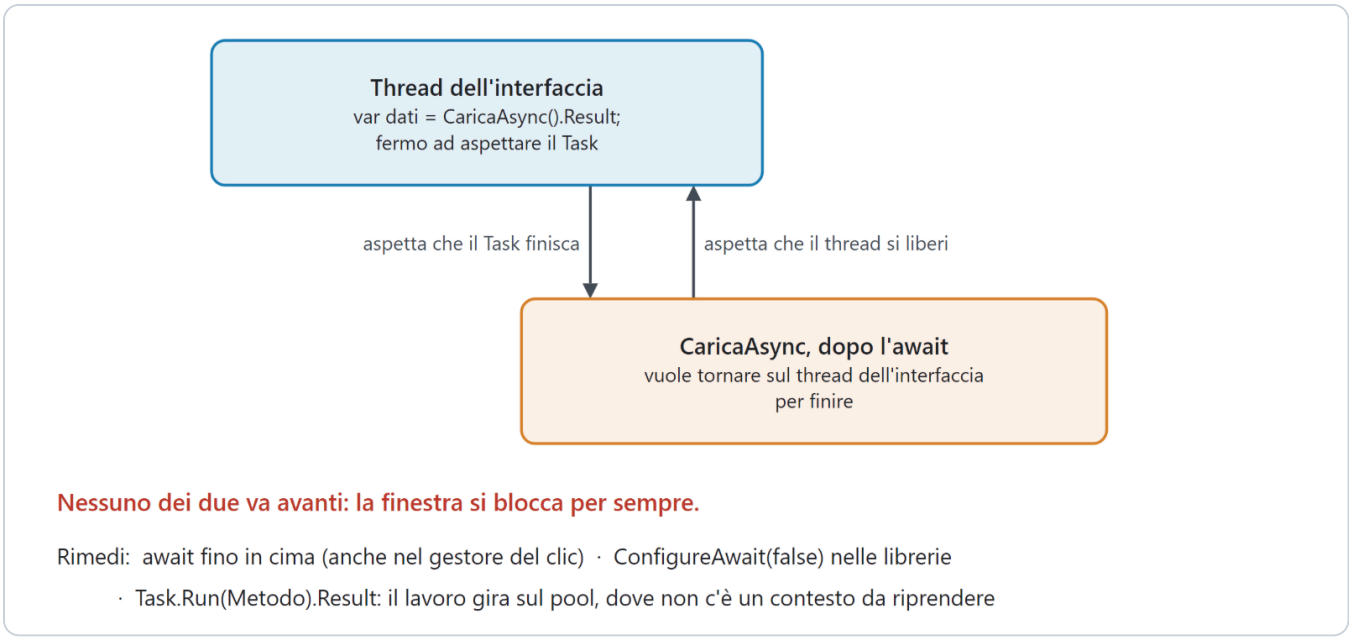
Il capitolo 10 costruisce un esempio completo con una barra di avanzamento.

2.9 Il blocco mortale

Ora il difetto più famoso. Nel gestore del clic, invece di `await`, scriviamo `Result` (per non mettere `async` dappertutto):

```
private void CaricaClic(object? mittente, RoutedEventArgs e) {
  List<string> dati = CaricaAsync().Result;
// NO: la finestra si blocca
  elenco.ItemsSource = dati;
}
```

La finestra si blocca, **per sempre**. Il thread dell'interfaccia è fermo in `Result`, ad aspettare che `CaricaAsync` finisca. Ma `CaricaAsync` contiene un `await`, e dopo l'`await` vuole riprendere sul thread dell'interfaccia (§2.8), che è fermo. Ognuno aspetta l'altro: è un **blocco mortale** (*deadlock*).



I rimedi, dal migliore:

1. **`await` fino in cima.** Il gestore diventa `async void` e usa `await`: il thread non si ferma mai;
2. **`ConfigureAwait(false)` nelle librerie.** Chi scrive un metodo `async` che non tocca i controlli può dire «non mi serve tornare sul contesto»: `await CaricaDalDiscoAsync().ConfigureAwait(false);`. Il resto del metodo riprende sul pool, e un `Result` sul thread dell'interfaccia non lo blocca più. Le librerie (quelle di .NET, e RAVAL) lo fanno sempre;
3. **`Task.Run(metodo).Result`.** Il lavoro gira su un thread del pool, dove non c'è un contesto da riprendere; il thread dell'interfaccia resta fermo per la durata del lavoro, ma non per sempre. Va bene per attese brevi, come avviare o fermare un servizio.

Un caso vero. Il server HTTP di RAVAL ha `Start()` e `Stop()` senza `await`, per chi non lo usa ancora. La prima versione di `Stop()` era `StopAsync().GetAwaiter().GetResult()`: chiamata dal thread dell'interfaccia, si bloccava per sempre, perché dentro ASP.NET un `await` tornava sul contesto della finestra. La versione giusta è il rimedio 3: `Task.Run(FermaAsync).GetAwaiter().GetResult()`. Il programma di prova `TEST_RAVAL` lo verifica a ogni compilazione.

2.10 Le eccezioni

Un'eccezione dentro un Task non si perde: viaggia con il Task e salta fuori quando lo si aspetta. Come salta fuori dipende da come si aspetta:

```
try {
  await GuastoAsync();
} catch (InvalidOperationException e) {
  Console.WriteLine($"   con await:  {e.GetType().Name}: {e.Message}");
}
try {
  GuastoAsync().Wait();
} catch (AggregateException e) {
  Console.WriteLine($"   con Wait(): {e.GetType().Name} con dentro
{e.InnerException?.GetType().Name}");
}
```

```
   con await:  InvalidOperationException: la macchina e' ferma
   con Wait(): AggregateException con dentro InvalidOperationException
```

Con `await` arriva l'eccezione originale, e il `try/catch` si scrive come per un metodo normale. Con `Wait()` e `Result` arriva un `AggregateException` (un Task può contenere più errori) che la contiene in `InnerException`; `GetAwaiter().GetResult()` invece la restituisce com'è, ed è per questo che si usa al posto di `Result` quando non si può usare `await`.

Due trappole:

- **un'eccezione in un thread creato con `new Thread`**, se nessuno la cattura, **chiude il programma**. Il metodo del thread deve avere il suo `try/catch`;
- **un'eccezione in un `async void`** non ha un Task che la porti: finisce al Dispatcher e chiude il programma. Nei gestori `async void` si mette sempre un `try/catch` attorno agli `await`.

2.11 Annullare

Un lavoro lungo deve potersi fermare: l'utente preme «Annulla», la finestra si chiude. Non si «uccide» un thread o un Task da fuori: gli si **chiede** di fermarsi con un `CancellationToken`, e il lavoro controlla la richiesta quando può.

```
/// <summary>Dieci passi da 100 ms che controllano il token a ogni passo.
</summary>
public sealed class LavoroLungo {
  /// <summary>Quanti passi sono stati fatti.</summary>
  public int Fatti { get; private set; }
  /// <summary>Il lavoro: si ferma con OperationCanceledException appena il
token e' annullato.</summary>
  public async Task EseguiAsync(CancellationToken token) {
    for (int i = 0; i < 10; i++) {
      await Task.Delay(100, token);
// anche l'attesa si interrompe
      Fatti++;
    }
  }
}
```

```
using var annulla = new CancellationTokenSource();
annulla.CancelAfter(250);
// come se l'utente premesse «Annulla»
var lavoro = new LavoroLungo();
try {
  await lavoro.EseguiAsync(annulla.Token);
} catch (OperationCanceledException) {
  Console.WriteLine($"   annullato dopo {lavoro.Fatti} passi su 10");
}
```

```
   annullato dopo 2 passi su 10
```

Il `CancellationTokenSource` sta a chi decide (il pulsante «Annulla» chiama `annulla.Cancel()`); il `CancellationToken` va a chi lavora, che lo passa a ogni attesa (`Task.Delay`, le letture asincrone) e, nei cicli di calcolo, chiama `token.ThrowIfCancellationRequested()`. Lo stesso token funziona anche in un thread classico: `if (token.IsCancellationRequested) return;`.

2.12 Tre modi per la stessa cosa

	Thread	Task, senza await
far partire un lavoro	`new Thread(Metodo).Start()`	`Task.Run(Metodo)`
aspettare la fine	`Join()`	`Wait()`, `Result`
aspettare più lavori	`Join()` di ognuno	`Task.WaitAll(...)`
un risultato	un campo dell'oggetto, letto dopo `Join`	`Task<T>.Result`
aspettare un punto preciso	`ManualResetEventSlim`	`ManualResetEventSlim`
tornare sul thread UI	`Dispatcher.UIThread.Post(Mostra)`	`Dispatcher.UIThread.Post(Mostra)`
errori	`try/catch` nel metodo del thread	`AggregateException` (o `GetAwaiter().GetResult()`)
annullare	`CancellationToken`, controllato a mano	`CancellationToken`
il thread che aspetta	fermo	fermo
quando usarlo	un servizio che gira sempre (lettura di una macchina, un server)	un calcolo da mandare fuori

In breve: un **thread** per chi lavora sempre, dall'inizio alla fine del programma; **`Task.Run`** per spostare un calcolo fuori dal thread dell'interfaccia; **`await`** per aspettare qualunque cosa senza fermare nessuno. E in tutti e tre i casi: **i controlli si toccano solo dal thread dell'interfaccia**, e i dati condivisi si proteggono con `lock` o, meglio, non si condividono.

Esercizi

1. Nell'esempio dei due operai aggiungete un terzo operaio «C» con cinque passi: quanto dura tutto? E con tre thread?
2. Nel `Contatore` togliete il `lock` ma usate un contatore **per thread** (due campi, uno per ciascuno), sommati alla fine dopo i `Join`. Il conto torna? Quanto è veloce rispetto al `lock`?
3. Riscrivete `TaskSenzaAwait` con `await` invece di `Result`. Cosa cambia nell'output?
4. Scrivete un metodo `async` che scarica tre «pagine» finte (`Task.Delay` da 200, 300 e 400 ms) prima in sequenza e poi con `WhenAll`, e stampate i due tempi.
5. In `LavoroLungo` sostituite `Task.Delay(100, token)` con `Thread.Sleep(100)` e `token.ThrowIfCancellationRequested()`. L'annullamento funziona ancora? Quando arriva, rispetto a prima?

Parte II — Avalonia in codice

3. Il primo programma

> **Cosa imparerai:** come si crea il progetto, cosa fanno `Program.Main`, `App` e la finestra, come si avvia il programma.

3.1 Cosa serve

- **.NET 10 SDK** da `dot.net` (Windows, macOS o Linux). Verificate con `dotnet --version`.
- Un editor: Visual Studio, Rider o Visual Studio Code con l'estensione C#.
- `RAVAL.dll`, compilata dalla soluzione `RAVAL.slnx` (`dotnet build RAVAL.slnx`): dipende solo da Avalonia, quindi non servono altre DLL.

3.2 Il progetto

Si parte da un progetto console vuoto:

```
dotnet new console -n PrimoProgramma
```

e si sostituisce il `.csproj` con questo:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>          <!-- su Windows niente finestra
nera della console -->
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Avalonia" Version="12.1.4" />
    <PackageReference Include="Avalonia.Desktop" Version="12.1.4" />
    <PackageReference Include="Avalonia.Themes.Fluent" Version="12.1.4" />
  </ItemGroup>

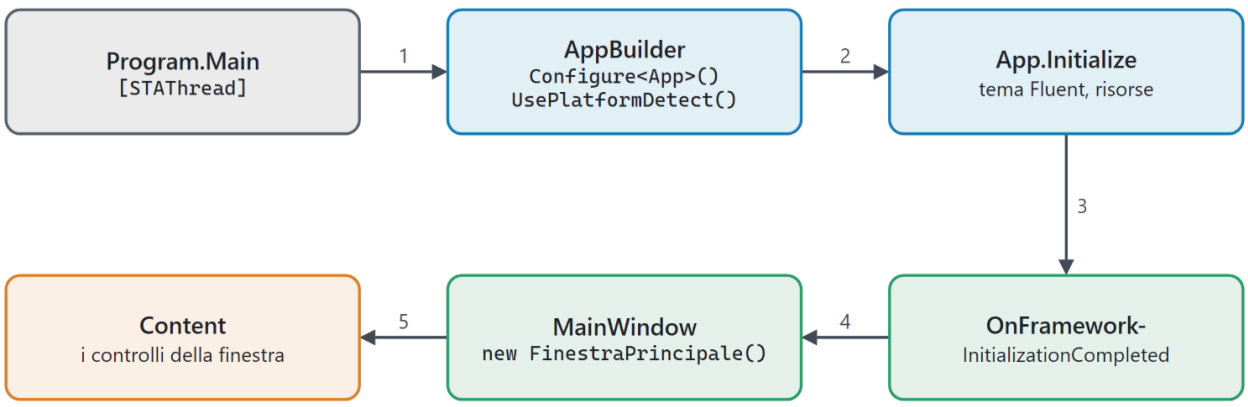
  <ItemGroup>
    <!-- RAVAL si usa come DLL: un riferimento al file compilato -->
    <Reference Include="RAVAL">
      <HintPath>..\RAVAL\RAVAL\bin\Debug\net10.0\RAVAL.dll</HintPath>
    </Reference>
  </ItemGroup>

</Project>
```

- `Avalonia` è il cuore, `Avalonia.Desktop` porta le finestre vere di Windows, macOS e Linux, `Avalonia.Themes.Fluent` l'aspetto dei controlli (il tema «Fluent», simile a Windows 11).
- `Nullable` acceso fa controllare al compilatore i valori null (§1.5).
- RAVAL si usa come **DLL** con `HintPath`: chi modifica la libreria la ricompila, e i programmi ricevono la versione nuova alla compilazione successiva. Se una funzione manca, la si aggiunge a RAVAL, non se ne scrive una copia nel programma.

3.3 Dall’avvio alla finestra

Un programma Avalonia attraversa sempre gli stessi passi:



1. `Program.Main` è il punto di partenza di ogni programma C#.
2. `AppBuilder` configura Avalonia: quale applicazione, quale piattaforma.
3. `App.Initialize` carica il tema e le risorse.
4. `OnFrameworkInitializationCompleted` arriva quando Avalonia è pronta: è il momento di creare la finestra.
5. La finestra mostra il suo `Content`, cioè i controlli.

Ecco il codice di questi passi (`Esempi\AvvioDesktop.cs`). L'applicazione, al posto del file `App.axaml` che vedreste nei tutorial con XAML:

```
/// <summary>L'applicazione del primo programma: tema Fluent e finestra
principale.</summary>
public sealed class App : Application {

  /// <summary>Il tema: senza questa riga i controlli non hanno aspetto.
</summary>
  public override void Initialize() {
    Styles.Add(new FluentTheme());
    RequestedThemeVariant = ThemeVariant.Default;               // chiaro o
scuro come il sistema
  }

  /// <summary>Avalonia e' pronta: si apre la finestra principale.</summary>
  public override void OnFrameworkInitializationCompleted() {
    if (ApplicationLifetime is IClassicDesktopStyleApplicationLifetime
desktop) desktop.MainWindow = new FinestraPrincipale();
    base.OnFrameworkInitializationCompleted();
  }
}
```

L'avvio. Negli esempi è un metodo chiamato da `Program.Main` quando si passa `--finestra`; nel vostro programma il corpo di `Avvia` è direttamente il `Main`, con sopra `[STAThread]` (serve su Windows per gli appunti e il trascinamento):

```
/// <summary>Avvia il programma con la finestra vera (ESEMPI_RAVAL --
finestra).</summary>
public static int Avvia(string[] args) =>
  AppBuilder.Configure<App>()
    .UsePlatformDetect()                                          // Windows,
macOS o Linux
    .LogToTrace()
    .StartWithClassicDesktopLifetime(args);
```

`StartWithClassicDesktopLifetime` non ritorna finché l'ultima finestra non si chiude: tutto il programma vive lì dentro.

E la finestra (`Esempi\PrimiPassi.cs`):

```
/// <summary>La prima finestra: un titolo, una misura e un contenuto.
</summary>
public sealed class FinestraPrincipale : Window {

  /// <summary>Titolo, misura e contenuto, tutto nel costruttore.</summary>
  public FinestraPrincipale() {
    Title = "Il mio primo programma";
    Width = 480;
    Height = 260;
    WindowStartupLocation = WindowStartupLocation.CenterScreen;
    Content = PrimiPassi.SalutoFluente();
  }
}
```

Nel progetto degli esempi questo programma si avvia con `dotnet run --project ESEMPI_RAVAL -- --finestra`.

3.4 Il contenuto, a mano

Il contenuto della finestra si può scrivere nel modo più semplice: si crea ogni oggetto e se ne assegnano le proprietà.

```
/// <summary>Il saluto scritto «a mano»: si crea l'oggetto e si assegnano le
proprieta'.</summary>
public static Control Saluto() {
  var titolo = new TextBlock();
  titolo.Text = "Ciao, Avalonia!";
  titolo.FontSize = 30;
  titolo.FontWeight = FontWeight.SemiBold;
  titolo.HorizontalAlignment = HorizontalAlignment.Center;

  var sotto = new TextBlock {                                        //
inizializzatore di oggetto
    Text = "Questa finestra non ha una riga di XAML.",
    FontSize = 15,
    HorizontalAlignment = HorizontalAlignment.Center,
  };

  var pila = new StackPanel { Spacing = 8, VerticalAlignment =
VerticalAlignment.Center };
  pila.Children.Add(titolo);
  pila.Children.Add(sotto);
  return pila;
}
```



3.5 Il contenuto, con RAva

Lo stesso contenuto con le estensioni di RAva: si legge dall'alto in basso, nello stesso ordine in cui i controlli stanno nella finestra.

```
/// <summary>Lo stesso saluto con le estensioni di RAva: si legge dall'alto
in basso come la finestra.</summary>
public static Control SalutoFluente() =>
  new StackPanel { Spacing = 8 }.VAlign(VerticalAlignment.Center).Children(
    new TextBlock { FontSize = 30, FontWeight = FontWeight.SemiBold }
      .Text("Ciao, Avalonia!").Align(HorizontalAlignment.Center,
VerticalAlignment.Top),
    new TextBlock { FontSize = 15 }
      .Text("Questa finestra non ha una riga di
XAML.").Align(HorizontalAlignment.Center, VerticalAlignment.Top));
```

Ogni estensione assegna una proprietà e **restituisce lo stesso controllo**, così se ne può chiamare un'altra di seguito. `Children(...)` aggiunge i controlli a un pannello. Con il tema scuro la stessa finestra diventa così, senza cambiare una riga:



Nel resto del libro useremo entrambi gli stili: l'inizializzatore fra graffe per le proprietà «di configurazione» (`FontSize`, `Spacing`), le estensioni per la disposizione e il contenuto.

Esercizi

1. Cambiate titolo e misura della finestra; aggiungete `CanResize = false`.

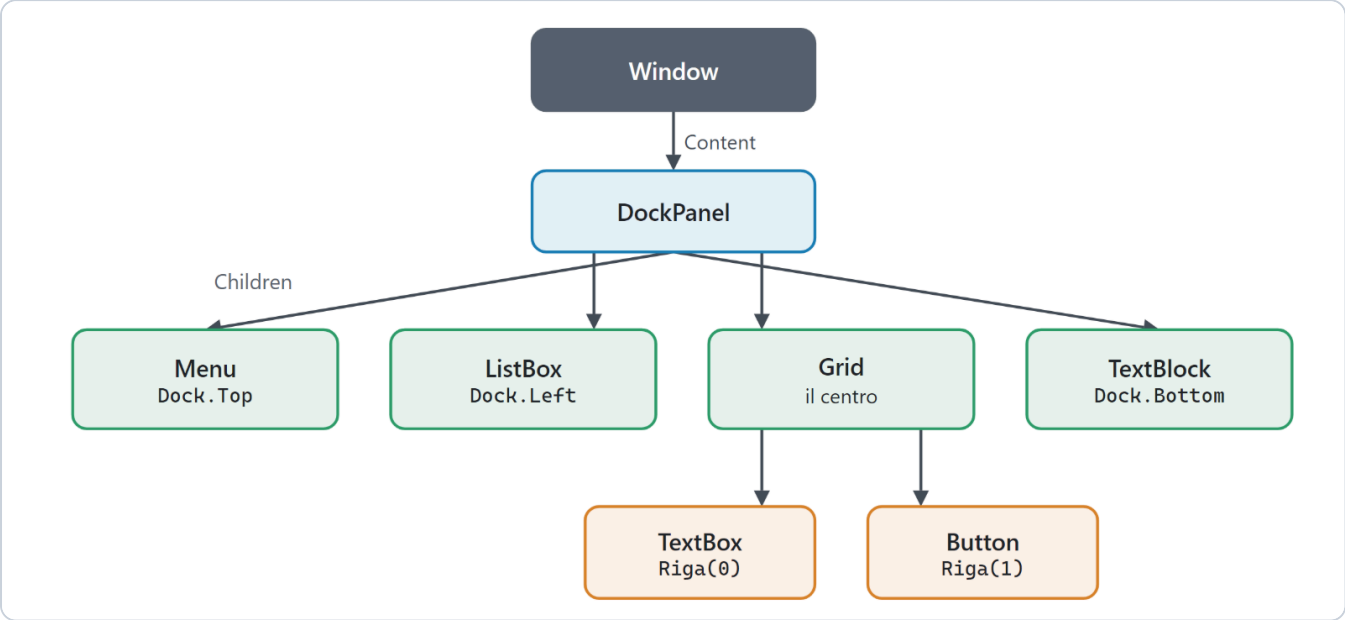
2. Aggiungete un terzo `TextBlock` con la data di oggi (`DateTime.Now.ToString("D")`).

4. Dove vanno i controlli: i pannelli

> **Cosa imparerai:** l'albero dei controlli, come Avalonia misura e dispone, e i cinque pannelli che bastano quasi sempre: StackPanel, Grid, DockPanel, WrapPanel e Border.

4.1 L'albero dei controlli

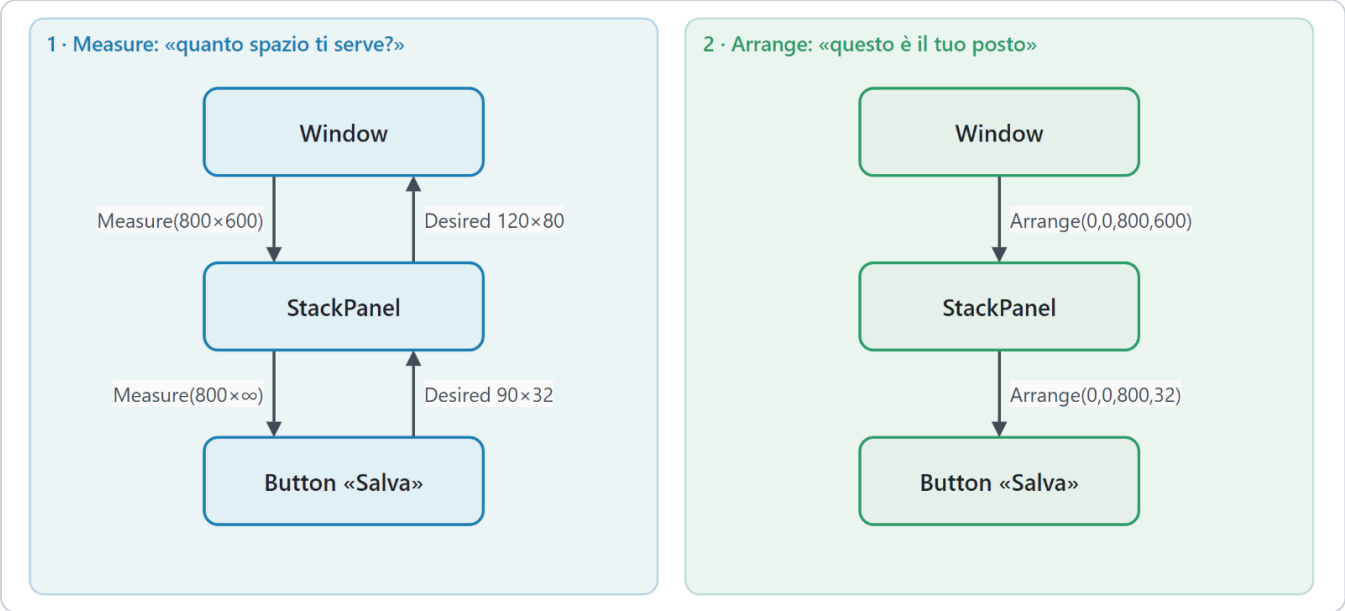
Una finestra contiene **un** controllo (`Content`). Se ne servono di più, quel controllo è un **pannello**, che ha dei figli (`Children`), che possono essere altri pannelli. Si forma un **albero**:



Un controllo ha **un solo genitore**: non si può mettere lo stesso `TextBlock` in due posti (capitolo 22).

4.2 Misura e disposizione

Ogni volta che la finestra cambia misura, Avalonia fa due passate sull'albero:



1. **Measure**: dall'alto in basso, ogni genitore chiede ai figli «quanto spazio ti serve, avendone tanto a disposizione?». Ogni figlio risponde con la sua `DesiredSize`.
2. **Arrange**: di nuovo dall'alto, ogni genitore assegna a ogni figlio il suo rettangolo definitivo.

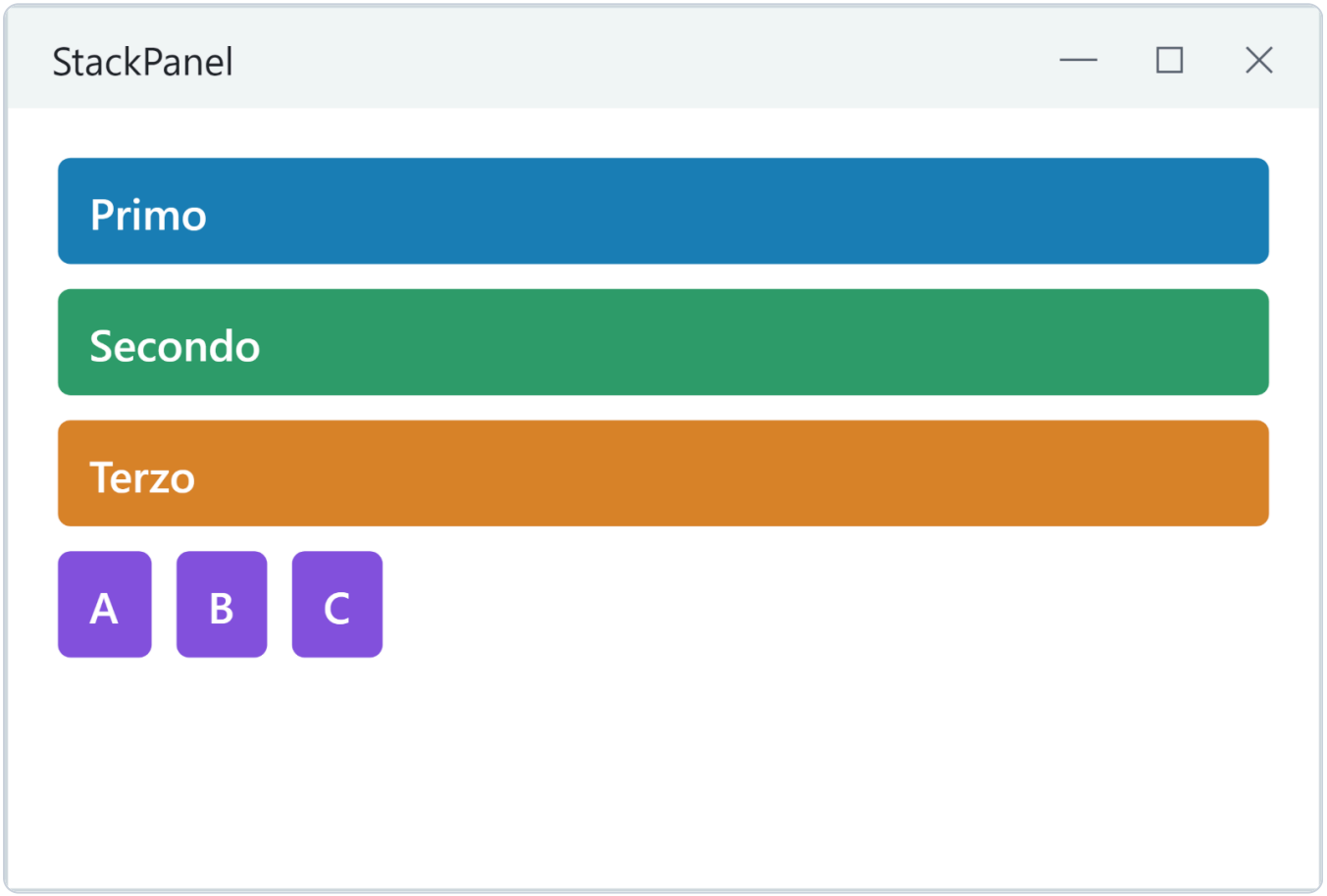
Il pannello decide **come** distribuire lo spazio: è l'unica differenza fra un pannello e l'altro. Le proprietà che ogni controllo ha per influire sono:

Proprietà	Estensione RAva	Effetto
`Margin`	`Margin(8)`, `Margin(8, 4)`, `Margin(s, a, d, b)`	spazio **fuori** dal controllo
`Padding`	—	spazio **dentro** (Border, Button, TextBox)
`HorizontalAlignment`, `VerticalAlignment`	`Align(o, v)`, `VAlign(v)`	dove sta nel suo spazio; `Stretch` lo riempie
`Width`, `Height`	`Width(l)`, `Height(a)`	misura fissa
`MinWidth`, `MaxWidth`…	—	limiti alla misura

4.3 StackPanel

I figli uno sotto l'altro (o uno accanto all'altro con `Orientation.Horizontal`), separati da `Spacing`.

```
/// <summary>StackPanel: i figli uno sotto l'altro (o uno accanto
all'altro), con uno spazio fisso.</summary>
public static Control Pila() =>
  new StackPanel { Spacing = 8 }.Margin(16).Children(
    Scatola("Primo", 0xFF1A7FB5),
    Scatola("Secondo", 0xFF2E9E6A),
    Scatola("Terzo", 0xFFD9822B),
    new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8
}.Children(
      Scatola("A", 0xFF8250DF), Scatola("B", 0xFF8250DF), Scatola("C",
0xFF8250DF)));
```

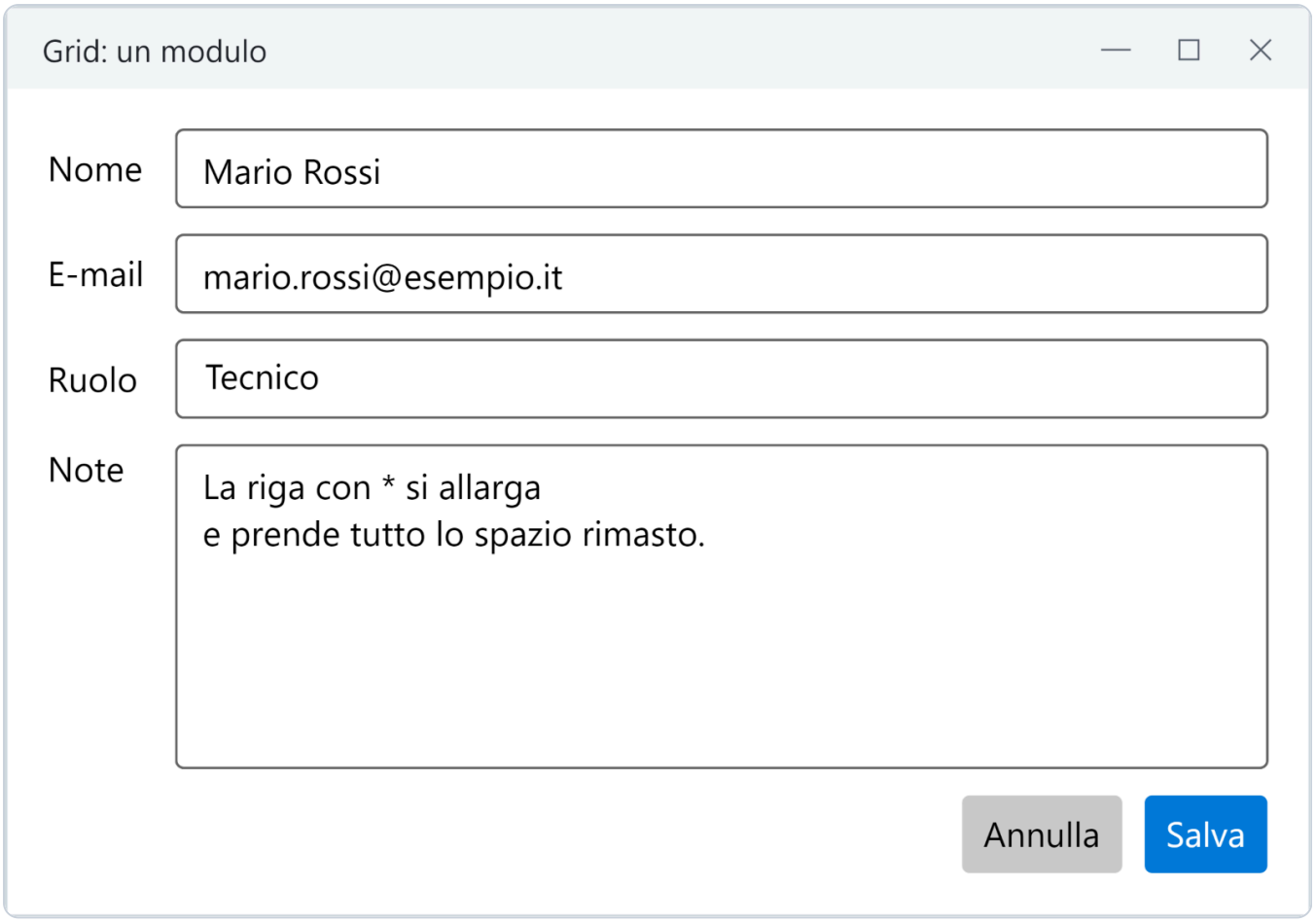


`Scatola` è un piccolo metodo dell'esempio che crea un `Border` colorato con una scritta: serve solo a far vedere lo spazio di ogni figlio.

4.4 Grid

Righe e colonne. Le misure si scrivono in una stringa: `Auto` prende quanto serve al contenuto, `*` divide lo spazio rimasto, `2*` ne prende il doppio, un numero è fisso in pixel. Ogni figlio dice la sua riga e la sua colonna con `Row(n)` e `Column(n)` (con `span` per occuparne più d'una).

```
/// <summary>Grid: righe e colonne; "Auto" prende il necessario, "*" il
resto, "2*" il doppio di "*".</summary>
public static Control Modulo() {
  var g = new Grid {
    ColumnDefinitions = new ColumnDefinitions("Auto,*"),
    RowDefinitions = new RowDefinitions("Auto,Auto,Auto,*,Auto"),
    ColumnSpacing = 12, RowSpacing = 10,
  }.Margin(16);
  g.Children(
    Etichetta("Nome").Row(0).Column(0),
    new TextBox { Text = "Mario Rossi" }.Row(0).Column(1),
    Etichetta("E-mail").Row(1).Column(0),
    new TextBox { Text = "mario.rossi@esempio.it" }.Row(1).Column(1),
    Etichetta("Ruolo").Row(2).Column(0),
    new ComboBox { ItemsSource = new[] { "Tecnico", "Commerciale" },
SelectedIndex = 0, HorizontalAlignment = HorizontalAlignment.Stretch
}.Row(2).Column(1),
    Etichetta("Note").Row(3).Column(0).VAlign(VerticalAlignment.Top),
    new TextBox { AcceptsReturn = true, TextWrapping = TextWrapping.Wrap,
Text = "La riga con * si allarga\ne prende tutto lo spazio rimasto."
}.Row(3).Column(1),
    new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8,
HorizontalAlignment = HorizontalAlignment.Right }
      .Row(4).Column(0, span: 2)
      .Children(new Button().Content("Annulla"), new Button { Classes = {
"accent" } }.Content("Salva")));
  return g;
}
```



La quarta riga (`*`) prende tutto lo spazio che avanza: allargando la finestra cresce la casella delle note, le altre restano della loro misura. La classe `accent` dà al pulsante il colore principale del tema.

4.5 DockPanel

I figli si **attaccano ai lati** nell'ordine in cui sono aggiunti, con `Dock(Dock.Top)` e simili; l'ultimo riempie quello che resta. È il pannello della cornice di quasi ogni applicazione: menu in alto, stato in basso, elenco a sinistra, documento al centro.

```
/// <summary>DockPanel: i figli si attaccano ai lati nell'ordine in cui
arrivano; l'ultimo riempie il centro.</summary>
public static Control Ancorato() =>
  new DockPanel().Children(
    Scatola("Top: barra degli strumenti",
0xFF1A7FB5).Height(44).Dock(Dock.Top),
    Scatola("Bottom: barra di stato",
0xFF59636E).Height(32).Dock(Dock.Bottom),
    Scatola("Left: elenco", 0xFF2E9E6A).Width(150).Dock(Dock.Left),
    Scatola("Right: dettagli", 0xFFD9822B).Width(130).Dock(Dock.Right),
    Scatola("il centro: tutto il resto", 0xFF8250DF));
```

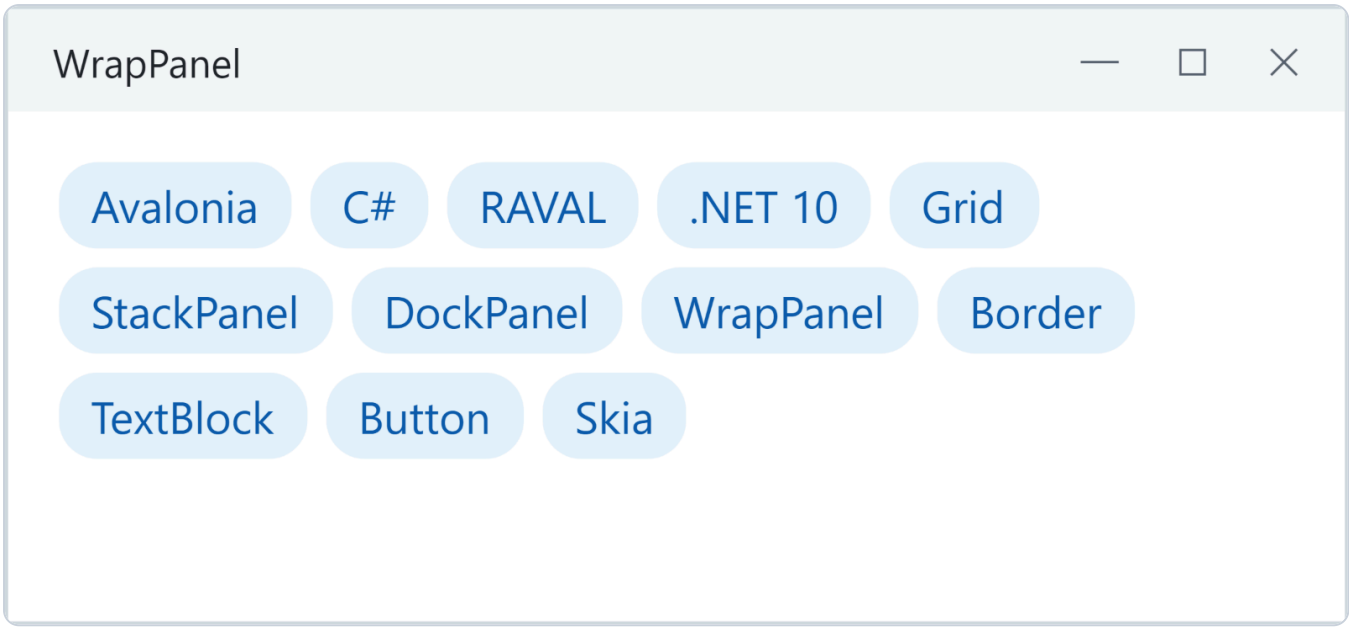


L'ordine conta: `Top` e `Bottom` sono aggiunti prima di `Left`, quindi occupano tutta la larghezza. Provate a spostare la riga del `Left` in cima e guardate cosa cambia.

4.6 WrapPanel e Border

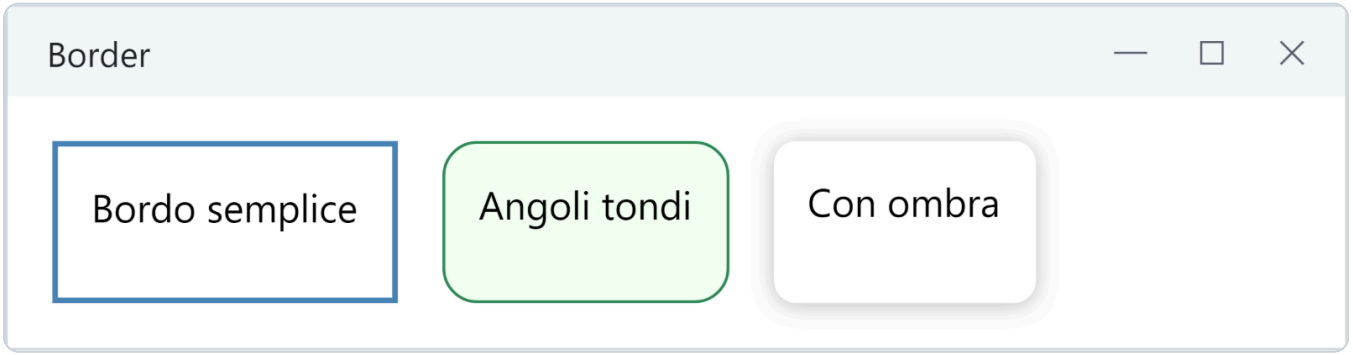
`WrapPanel` mette i figli in riga e va a capo quando la riga è piena, come le parole di un testo:

```
/// <summary>WrapPanel: i figli in riga, e quando la riga e' piena si va a
capo.</summary>
public static Control ACapo() {
  var p = new WrapPanel { ItemSpacing = 6, LineSpacing = 6 }.Margin(16);
  string[] parole = { "Avalonia", "C#", "RAVAL", ".NET 10", "Grid",
"StackPanel", "DockPanel", "WrapPanel", "Border", "TextBlock", "Button",
"Skia" };
  foreach (string parola in parole) p.Children.Add(Pillola(parola));
  return p;
}
```



`Border` non è un pannello ma una **cornice** attorno a un solo figlio (`Child(...)`): bordo, angoli arrotondati, sfondo, spazio interno, ombra.

```
/// <summary>Border: cornice, angoli arrotondati, sfondo e spazio interno
attorno a un solo figlio.</summary>
public static Control Cornici() =>
  new StackPanel { Orientation = Orientation.Horizontal, Spacing = 16
}.Margin(16).Children(
    new Border { BorderBrush = Brushes.SteelBlue, BorderThickness = new
Thickness(2), Padding = new Thickness(12) }
      .Child(new TextBlock().Text("Bordo semplice")),
    new Border { Background = Brushes.Honeydew, BorderBrush =
Brushes.SeaGreen, BorderThickness = new Thickness(1), CornerRadius = new
CornerRadius(12), Padding = new Thickness(12) }
      .Child(new TextBlock().Text("Angoli tondi")),
    new Border { Background = Brushes.White, CornerRadius = new
CornerRadius(8), Padding = new Thickness(12),
                 BoxShadow = BoxShadows.Parse("0 4 12 0 #40000000") }
      .Child(new TextBlock().Text("Con ombra")));
```

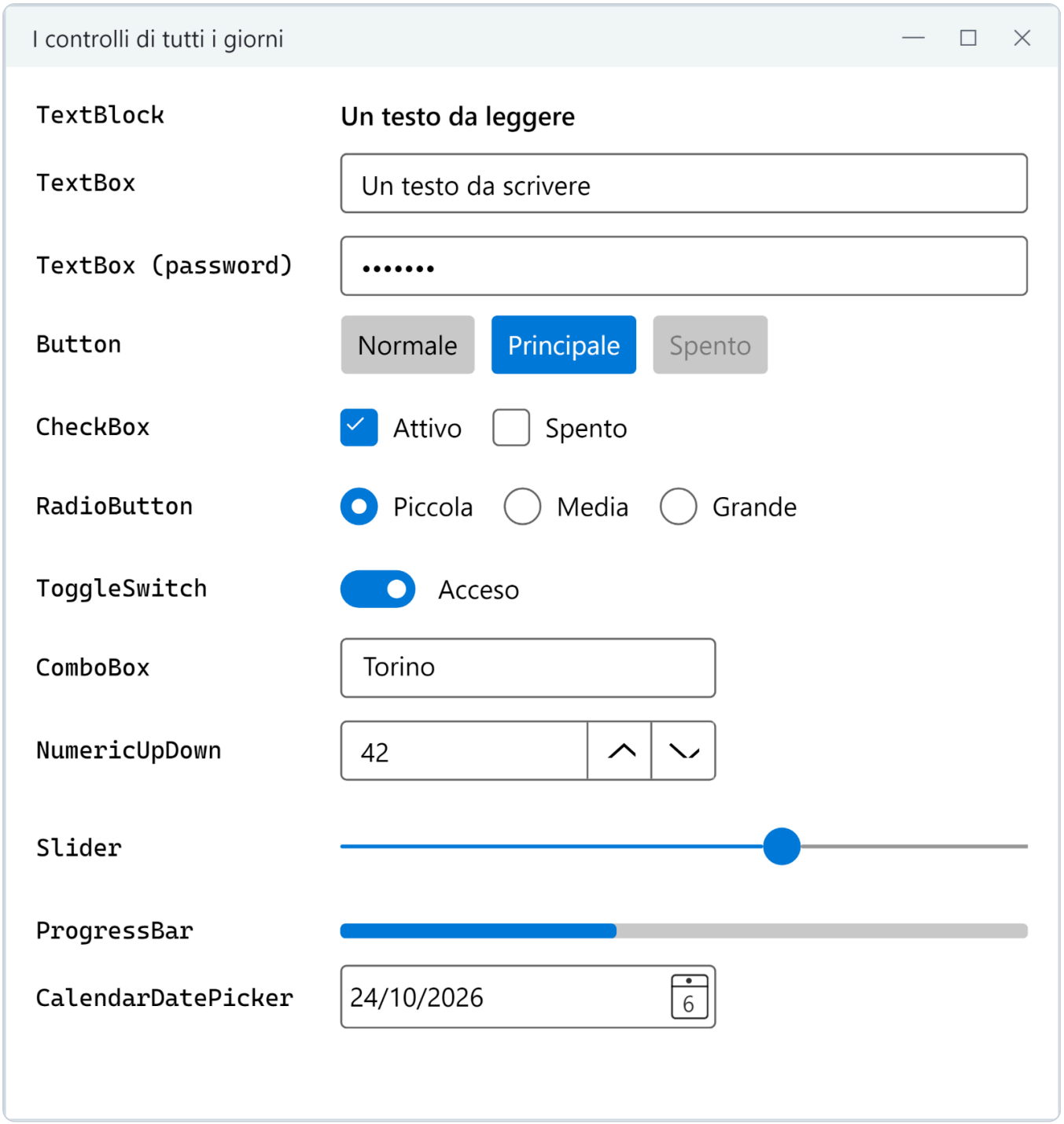


Esercizi

1. Rifate il modulo del §4.4 con una terza colonna `Auto` per un pulsante «…» accanto all'e-mail.
2. Costruite con DockPanel e Grid la cornice di un programma di posta: cartelle a sinistra, elenco dei messaggi in alto al centro, il messaggio sotto.

5. I controlli di tutti i giorni

> **Cosa imparerai:** i controlli che si usano in quasi ogni finestra e le loro proprietà principali.



Il codice della figura è in `Esempi\Controlli.cs`: una griglia a due colonne riempita da un metodo che aggiunge una riga alla volta.

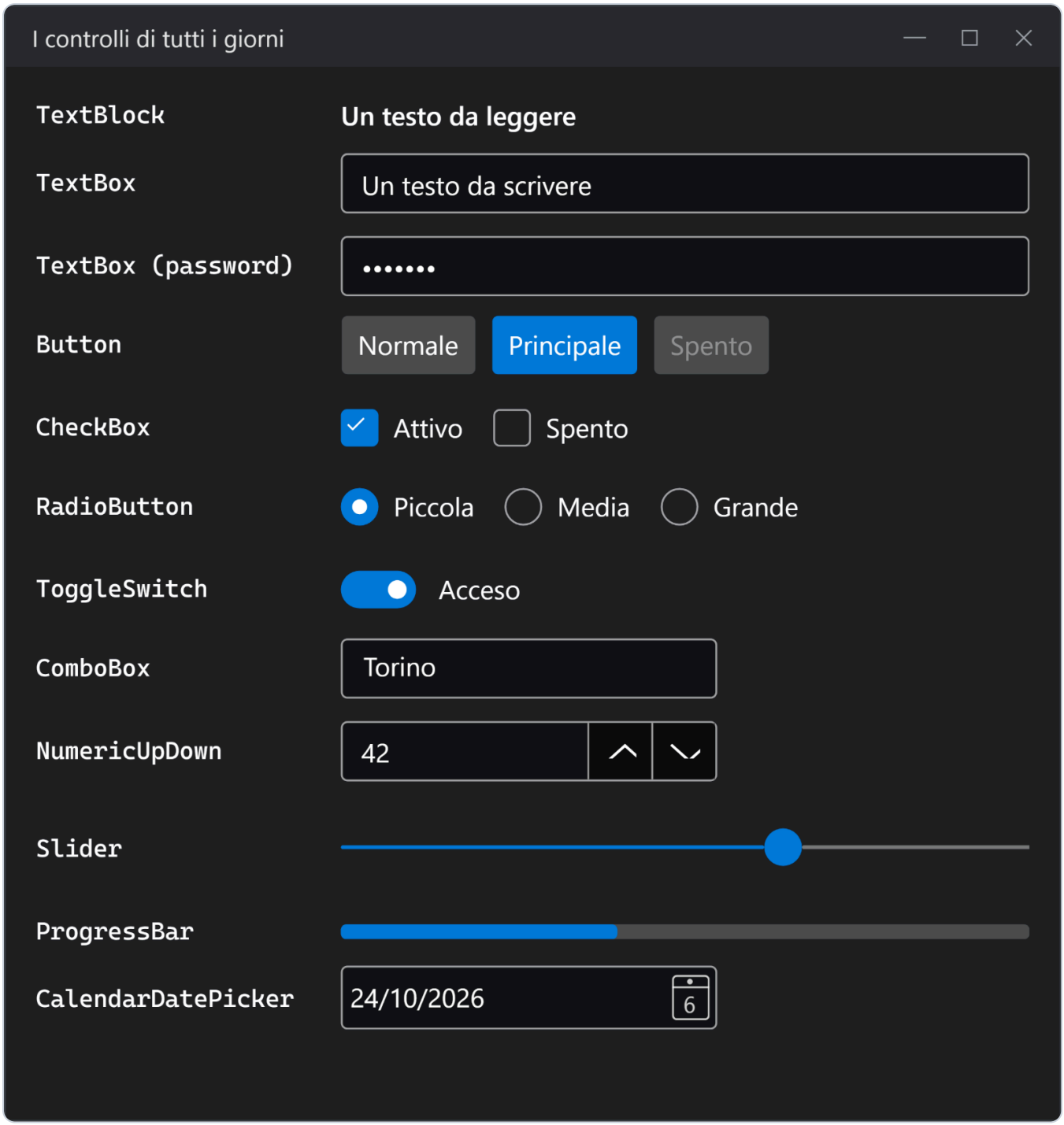
```
/// <summary>I controlli piu' usati, a due colonne: a sinistra il nome, a
destra il controllo.</summary>
public static Control Galleria() {
  var g = new Grid { ColumnDefinitions = new ColumnDefinitions("150,*"),
RowSpacing = 10, ColumnSpacing = 12 }.Margin(16);
  Riga(g, "TextBlock", new TextBlock { FontWeight = FontWeight.SemiBold
}.Text("Un testo da leggere"));
  Riga(g, "TextBox", new TextBox { Text = "Un testo da scrivere" });
  Riga(g, "TextBox (password)", new TextBox { PasswordChar = '•', Text =
"segreto" });
  Riga(g, "Button", new StackPanel { Orientation = Orientation.Horizontal,
Spacing = 8 }.Children(
    new Button().Content("Normale"), new Button { Classes = { "accent" }
}.Content("Principale"), new Button { IsEnabled = false
}.Content("Spento")));
  Riga(g, "CheckBox", new StackPanel { Orientation = Orientation.Horizontal,
Spacing = 16 }.Children(
    new CheckBox { IsChecked = true }.Content("Attivo"), new
CheckBox().Content("Spento")));
  Riga(g, "RadioButton", new StackPanel { Orientation =
Orientation.Horizontal, Spacing = 16 }.Children(
    new RadioButton { GroupName = "taglia", IsChecked = true
}.Content("Piccola"),
    new RadioButton { GroupName = "taglia" }.Content("Media"),
    new RadioButton { GroupName = "taglia" }.Content("Grande")));
  Riga(g, "ToggleSwitch", new ToggleSwitch { IsChecked = true, OnContent =
"Acceso", OffContent = "Spento" });
  Riga(g, "ComboBox", new ComboBox { ItemsSource = new[] { "Torino",
"Milano", "Genova" }, SelectedIndex = 0, Width = 200 });
  Riga(g, "NumericUpDown", new NumericUpDown { Value = 42, Minimum = 0,
Maximum = 100, Width = 200, HorizontalAlignment = HorizontalAlignment.Left
});
  Riga(g, "Slider", new Slider { Minimum = 0, Maximum = 100, Value = 65 });
  Riga(g, "ProgressBar", new ProgressBar { Minimum = 0, Maximum = 100, Value
= 40, Height = 8 });
  Riga(g, "CalendarDatePicker", new CalendarDatePicker { SelectedDate = new
DateTime(2026, 10, 24), Width = 200 });
  return g;
}

// aggiunge una riga alla griglia: il nome del controllo e il controllo
private static void Riga(Grid g, string nome, Control controllo) {
  int riga = g.RowDefinitions.Count;
  g.RowDefinitions.Add(new RowDefinition(GridLength.Auto));
  g.Children(
```

```
    new TextBlock { FontFamily = new FontFamily("Cascadia Mono, Consolas,
monospace"), FontSize = 13 }
      .Text(nome).VAlign(VerticalAlignment.Center).Row(riga).Column(0),
    controllo.VAlign(VerticalAlignment.Center).Row(riga).Column(1));
}
```

Controllo	Per	Proprietà da sapere	Evento
`TextBlock`	testo da leggere	`Text`, `FontSize`, `FontWeight`, `Foreground`, `TextWrapping`	—
`TextBox`	testo da scrivere	`Text`, `PlaceholderText`, `AcceptsReturn`, `PasswordChar`, `IsReadOnly`	`TextChanged`
`Button`	un comando	`Content`, `IsEnabled`, `IsDefault` (Invio), `IsCancel` (Esc)	`Click`
`CheckBox`, `ToggleSwitch`	sì/no	`IsChecked` (anche null: indeterminato)	`IsCheckedChanged`
`RadioButton`	una scelta fra poche	`GroupName`, `IsChecked`	`IsCheckedChanged`
`ComboBox`	una scelta fra tante	`ItemsSource`, `SelectedIndex`, `SelectedItem`	`SelectionChanged`
`NumericUpDown`	un numero	`Value` (decimal?), `Minimum`, `Maximum`, `Increment`	`ValueChanged`
`Slider`	un valore continuo	`Value`, `Minimum`, `Maximum`	`ValueChanged`
`ProgressBar`	un avanzamento	`Value`, `Maximum`, `IsIndeterminate`	—
`CalendarDatePicker`	una data	`SelectedDate` (DateTime?)	`SelectedDateChanged`

Lo stesso codice nel tema scuro:



Esercizi

1. Aggiungete alla galleria un `ListBox` con tre voci e un `Expander`.
2. Rendete la `ProgressBar` indeterminata (`IsIndeterminate = true`): cosa vedete nella finestra vera?

6. Gli eventi, senza lambda

> **Cosa imparerai:** come si risponde a un clic o a un tasto con metodi che hanno un nome, dove si tiene lo stato, cosa sono gli eventi instradati e perché Invio in una TextBox va preso nella fase Tunnel.

6.1 Lo stato in una classe

Un'interfaccia reagisce: l'utente preme un pulsante, il programma cambia qualcosa. Il modo più ordinato è una **classe** che tiene in campi privati sia lo stato sia i controlli che le servono, e collega gli eventi a **metodi** della classe stessa.

```
public sealed class Contatore {

  private int _valore;                                          // lo stato
del contatore
  private readonly TextBlock _numero = new() { FontSize = 48, FontWeight =
FontWeight.Bold, HorizontalAlignment = HorizontalAlignment.Center };

  /// <summary>Il pulsante «+1» (pubblico per poterlo premere dal programma
delle figure).</summary>
  public Button Piu { get; } = new Button { Classes = { "accent" }, MinWidth
= 80 }.Content("+1");

  /// <summary>Il pulsante che riporta a zero.</summary>
  public Button Azzera { get; } = new Button { MinWidth = 80
}.Content("Azzera");

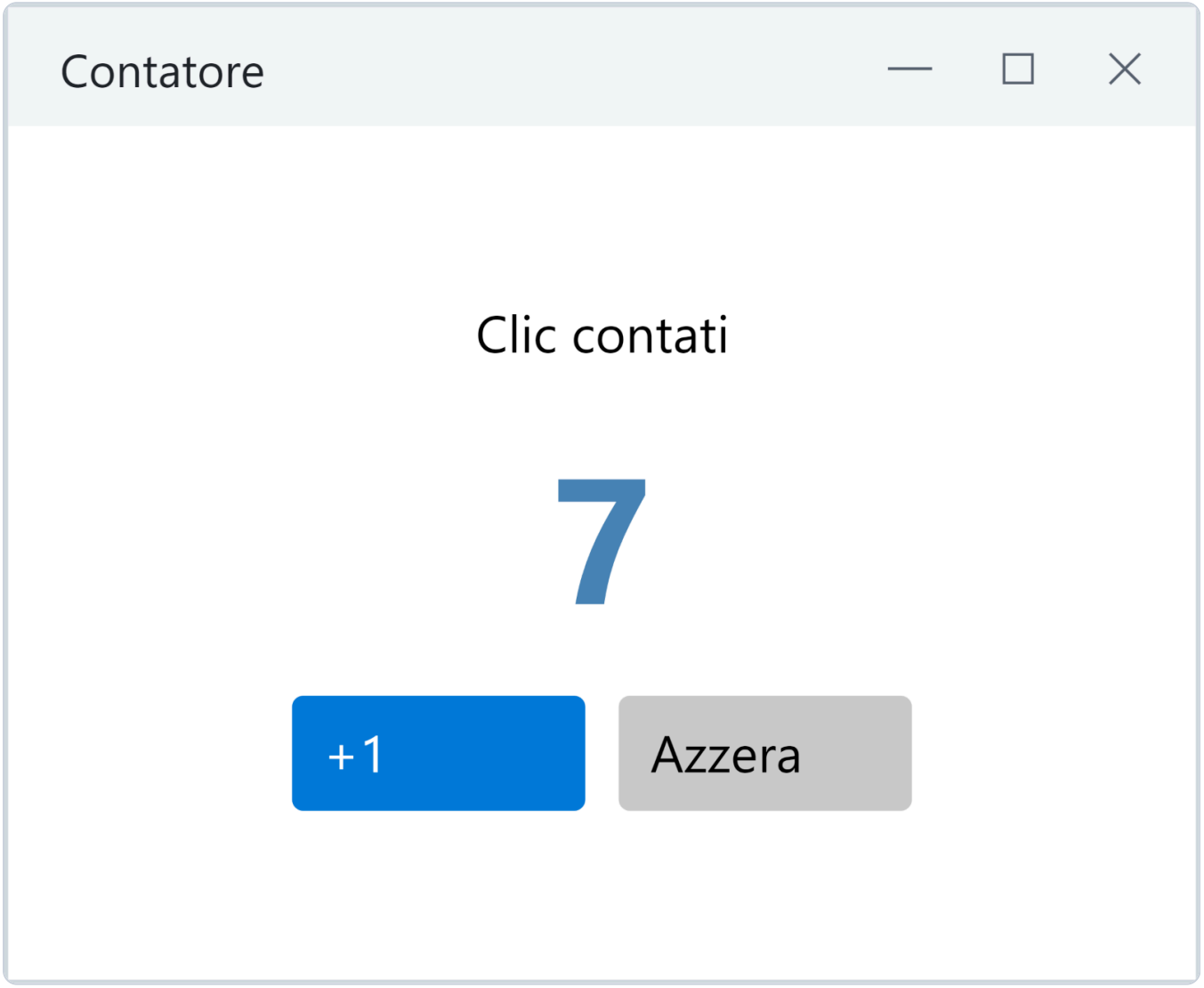
  /// <summary>Il contatore da mettere in una finestra.</summary>
  public Control Vista { get; }

  /// <summary>Costruisce la vista e collega gli eventi ai metodi.</summary>
  public Contatore() {
    Piu.Click += PiuCliccato;                                   // gruppo di
metodi, non lambda
    Azzera.Click += AzzeraCliccato;
    Vista = new StackPanel { Spacing = 12
}.Align(HorizontalAlignment.Center, VerticalAlignment.Center).Children(
      new TextBlock { FontSize = 14 }.Text("Clic
contati").Align(HorizontalAlignment.Center, VerticalAlignment.Top),
      _numero,
      new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8
}.Children(Piu, Azzera));
    Mostra();
  }

  // Button.Click di «+1»
  private void PiuCliccato(object? mittente, RoutedEventArgs e) {
    _valore++;
    Mostra();
  }

  // Button.Click di «Azzera»
  private void AzzeraCliccato(object? mittente, RoutedEventArgs e) {
    _valore = 0;
    Mostra();
  }
```

```
  // lo stato sullo schermo: un solo posto dove si aggiorna il testo
  private void Mostra() {
    _numero.Text = _valore.ToString();
    _numero.Foreground = _valore >= 10 ? Brushes.OrangeRed :
Brushes.SteelBlue;
  }
}
```



La figura è stata fatta premendo davvero sette volte il pulsante dal programma: `c.Piu.RaiseEvent(new RoutedEventArgs(Button.ClickEvent))`. Tre idee da portarsi via:

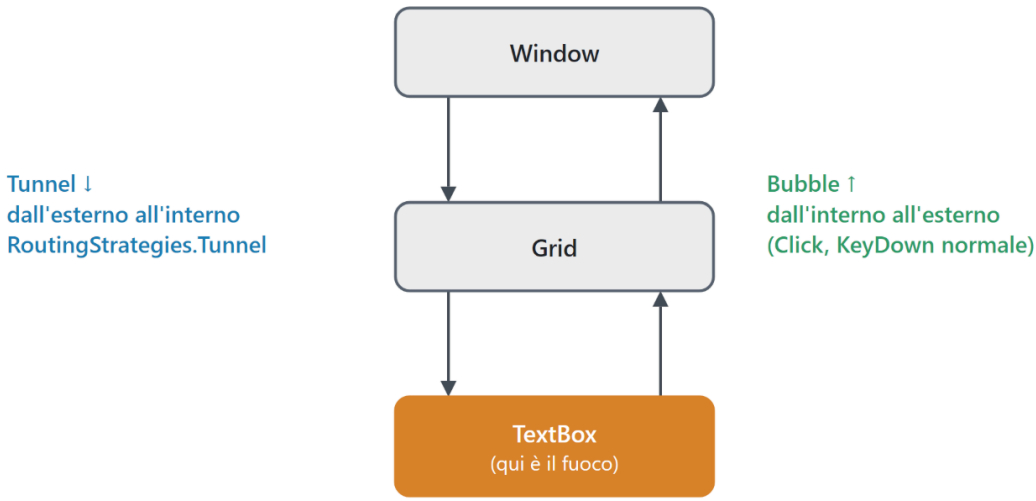
1. **Un gestore d'evento è un metodo** con la firma `(object? mittente, XxxEventArgs e)`. Lo si collega con `+=` e il suo nome, e il nome dice cosa fa: `PiuCliccato`, non `b__0_1`.
2. **Lo stato sta in un campo**, non nei controlli: `_valore` è la verità, il `TextBlock` ne è solo la rappresentazione.
3. **Un solo metodo aggiorna lo schermo** (`Mostra`): ogni gestore cambia lo stato e poi lo chiama. Così non si dimentica mai di aggiornare qualcosa.

Per i casi più semplici, quando il gestore non ha bisogno degli argomenti dell'evento, RAva offre `OnClick`, che accetta un' `Action` (un metodo senza argomenti):

```
var aggiungi = new Button().Content("Aggiungi").OnClick(Aggiungi).Column(1);
```

6.2 Gli eventi instradati

Molti eventi di Avalonia (tasti, mouse, `Click`) sono **instradati**: non arrivano solo al controllo che li riceve, ma viaggiano lungo l'albero.



Con AcceptsReturn la TextBox usa Invio e lo segna gestito: si ascolta nella fase Tunnel, prima di lei.

- **Tunnel**: dalla finestra verso il controllo che ha il fuoco. Serve a intercettare prima.
- **Bubble**: dal controllo verso la finestra. È la fase normale: `KeyDown` e `Click` si ascoltano qui.

Chi mette `e.Handled = true` ferma il viaggio: i gestori che vengono dopo non lo vedono più, a meno che non siano iscritti con `handledEventsToo: true`. Con Invio in una `TextBox` questo conta. Una casella su più righe (`AcceptsReturn = true`) usa Invio per andare a capo e lo segna come gestito: un `KeyDown` normale, nella fase Bubble, non lo vede mai. Una casella di una riga invece lo lascia passare; ma anche lì conviene ascoltare nella fase **Tunnel**, prima della TextBox: si riceve Invio qualunque cosa la casella ne farebbe, e mettendo `e.Handled = true` si decide che è nostro.

Un'avvertenza sull'ordine: sullo **stesso** controllo, i gestori della stessa fase vengono chiamati dall'ultimo iscritto al primo. Se l'ultimo segna l'evento come gestito, quelli iscritti prima non lo ricevono. (Misurato con Avalonia 12.1.4: `TEST_RAVAL\ProveInvio.cs` preme Invio davvero e registra chi lo riceve.)

L'esempio ascolta nella fase Tunnel:

```
public sealed class ListaSpesa {

  private readonly TextBox _nuova = new() { PlaceholderText = "Cosa manca?
(Invio per aggiungere)" };
  private readonly ListBox _elenco = new() { MinHeight = 140 };
  private readonly TextBlock _conteggio = new() { Opacity = 0.7 };
  private readonly List<string> _cose = new();

  /// <summary>La lista da mettere in una finestra.</summary>
  public Control Vista { get; }

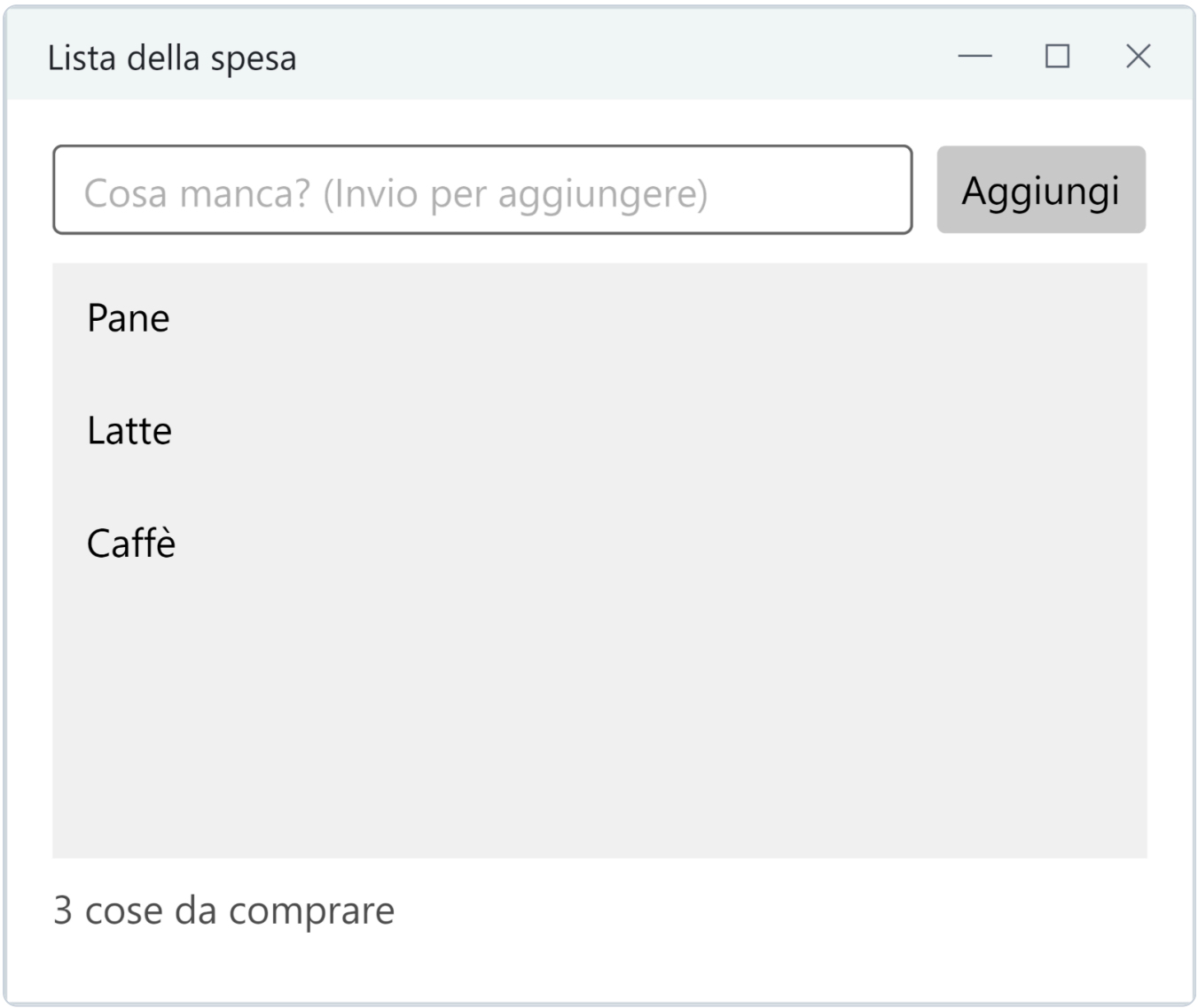
  /// <summary>Costruisce la vista e collega Invio e il pulsante allo stesso
metodo.</summary>
  public ListaSpesa() {
    _nuova.AddHandler(InputElement.KeyDownEvent, TastoPremuto,
RoutingStrategies.Tunnel);
    var aggiungi = new
Button().Content("Aggiungi").OnClick(Aggiungi).Column(1);   // OnClick di
RAva: un'azione senza argomenti
    Vista = new DockPanel().Margin(16).Children(
      new Grid { ColumnDefinitions = new ColumnDefinitions("*,Auto"),
ColumnSpacing = 8 }.Dock(Dock.Top).Children(_nuova, aggiungi),
      _conteggio.Margin(0, 8).Dock(Dock.Bottom),
      _elenco.Margin(0, 10, 0, 0));
    Mostra();
  }

  // KeyDown in fase Tunnel: Invio aggiunge, gli altri tasti passano
  private void TastoPremuto(object? mittente, KeyEventArgs e) {
    if (e.Key != Key.Enter) return;
    Aggiungi();
    e.Handled = true;
  }

  // il testo della casella diventa una voce dell'elenco
  private void Aggiungi() {
    string testo = (_nuova.Text ?? "").Trim();
    if (testo.Length == 0) return;
    _cose.Add(testo);
    _nuova.Text = "";
    Mostra();
  }

  // elenco e conteggio sullo schermo
  private void Mostra() {
```

```
    _elenco.ItemsSource = _cose.ToArray();
    _conteggio.Text = _cose.Count == 1 ? "1 cosa da comprare" : _cose.Count
+ " cose da comprare";
  }
}
```



Il tasto Invio e il pulsante chiamano **lo stesso metodo** `Aggiungi`: il comportamento è uno, i modi per chiederlo sono due.

Esercizi

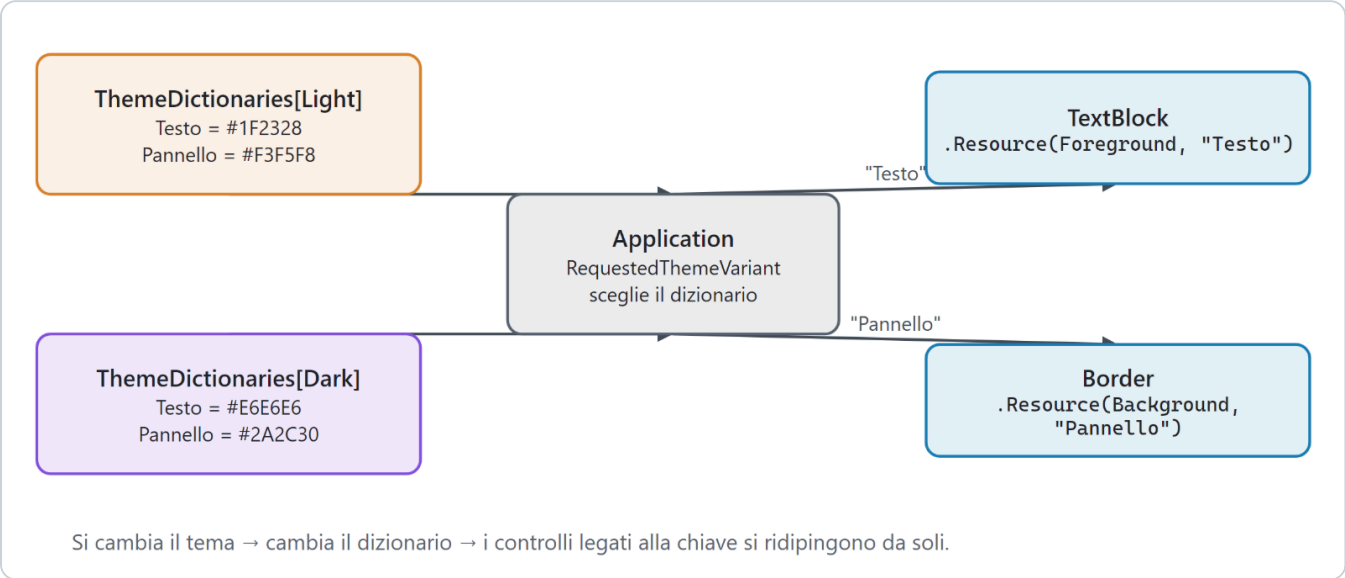
1. Aggiungete al contatore un pulsante «−1» che non scende sotto lo zero.
2. Nella lista della spesa, fate sì che Canc tolga la voce selezionata (`KeyDown` dell'elenco, `Key.Delete`).

7. Chiaro e scuro: i temi

> **Cosa imparerai:** dove si mettono i colori, come i controlli li seguono quando il tema cambia, e come cambiare tema dal programma.

7.1 Colori come risorse

Un colore scritto dentro un controllo (`Foreground = Brushes.Black`) resta nero anche nel tema scuro, dove diventa illeggibile. La soluzione è dare ai colori un **nome** e metterli in due dizionari, uno per tema:



Nell'applicazione degli esempi (`AppEsempi.cs`):

```
/// <summary>Tema Fluent e colori dei due temi.</summary>
public override void Initialize() {
  Styles.Add(new FluentTheme());
  RequestedThemeVariant = ThemeVariant.Light;
  Resources.ThemeDictionaries[ThemeVariant.Light] = Colori(0xFFFFFFFF,
0xFFF3F5F8, 0xFF1F2328, 0xFF59636E, 0xFFD1D9E0, 0xFF1A7FB5);
  Resources.ThemeDictionaries[ThemeVariant.Dark] = Colori(0xFF1E1F22,
0xFF2A2C30, 0xFFE6E6E6, 0xFFA0A4AB, 0xFF3A3D42, 0xFF30C0E0);
}

// i pennelli di un tema: sfondo, pannello, testo, testo tenue, bordo,
accento
private static ResourceDictionary Colori(uint sfondo, uint pannello, uint
testo, uint tenue, uint bordo, uint accento) => new() {
  ["Sfondo"] = new SolidColorBrush(Color.FromUInt32(sfondo)),
  ["Pannello"] = new SolidColorBrush(Color.FromUInt32(pannello)),
  ["Testo"] = new SolidColorBrush(Color.FromUInt32(testo)),
  ["TestoTenue"] = new SolidColorBrush(Color.FromUInt32(tenue)),
  ["Bordo"] = new SolidColorBrush(Color.FromUInt32(bordo)),
  ["Accento"] = new SolidColorBrush(Color.FromUInt32(accento)),
};
```

I colori si scrivono come `0xAARRGGBB`: i primi due numeri esadecimali sono l'opacità (`FF` = pieno).

7.2 Legare una proprietà a una risorsa

Il controllo non riceve il colore ma **la chiave**, con `Resource` di RAva:

```
var nome = new TextBlock { FontSize = 20, FontWeight = FontWeight.SemiBold
}.Text("Officine Rossi S.r.l.")
  .Resource(TextBlock.ForegroundProperty, "Testo");
```

`Resource(proprietà, chiave)` lega la proprietà alla risorsa con quel nome: quando il tema cambia, Avalonia trova il pennello nel dizionario nuovo e la proprietà si aggiorna da sola. Il primo argomento è la **proprietà di Avalonia** (`TextBlock.ForegroundProperty`, `Border.BackgroundProperty`...), cioè l'oggetto che descrive la proprietà, non il suo valore. La scheda completa:

```
/// <summary>La scheda di un cliente, tutta con colori del tema.</summary>
public static Control Scheda() {
  var nome = new TextBlock { FontSize = 20, FontWeight = FontWeight.SemiBold
}.Text("Officine Rossi S.r.l.")
    .Resource(TextBlock.ForegroundProperty, "Testo");
  var citta = new TextBlock().Text("Via Roma 12 · 10100
Torino").Resource(TextBlock.ForegroundProperty, "TestoTenue");
  var stato = new Border { CornerRadius = new CornerRadius(10), Padding =
new Thickness(8, 2), HorizontalAlignment = HorizontalAlignment.Left }
    .Resource(Border.BackgroundProperty, "Accento")
    .Child(new TextBlock { Foreground = Brushes.White, FontSize = 12
}.Text("Cliente attivo"));
  var dati = new Grid { ColumnDefinitions = new ColumnDefinitions("Auto,*"),
RowDefinitions = new RowDefinitions("Auto,Auto,Auto"), ColumnSpacing = 16,
RowSpacing = 4 };
  dati.Children(
    Tenue("Telefono").Row(0), Normale("011 123 4567").Row(0).Column(1),
    Tenue("E-mail").Row(1),
Normale("ordini@officinerossi.it").Row(1).Column(1),
    Tenue("Ordini").Row(2), Normale("128 nel 2026").Row(2).Column(1));
  var scheda = new Border { CornerRadius = new CornerRadius(10),
BorderThickness = new Thickness(1), Padding = new Thickness(18) }
    .Resource(Border.BackgroundProperty, "Pannello")
    .Resource(Border.BorderBrushProperty, "Bordo")
    .Child(new StackPanel { Spacing = 8 }.Children(nome, citta, stato, dati,
new Button { Classes = { "accent" }, Margin = new Thickness(0, 6, 0, 0)
}.Content("Nuovo ordine")));
  return new Border { Padding = new Thickness(20)
}.Resource(Border.BackgroundProperty, "Sfondo").Child(scheda);
}
```

Lo stesso metodo, chiamato una volta nel tema chiaro e una nel tema scuro:

Cliente

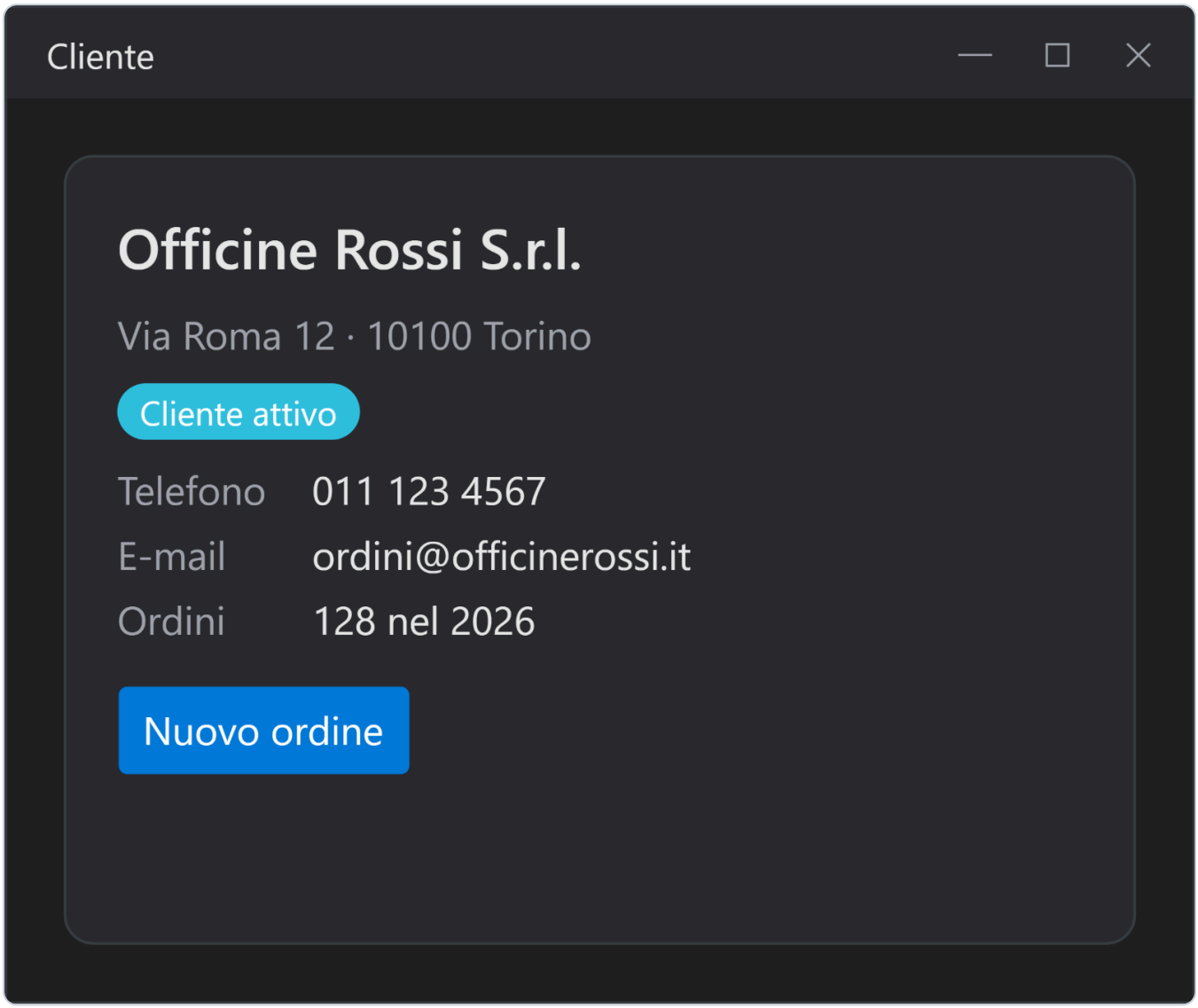
Officine Rossi S.r.l.

Via Roma 12 · 10100 Torino

Cliente attivo

Telefono	011 123 4567
E-mail	ordini@officinerossi.it
Ordini	128 nel 2026

Nuovo ordine



7.3 Cambiare tema

```
/// <summary>Passa dal tema chiaro allo scuro e viceversa, per tutta
l'applicazione.</summary>
public static void CambiaTema() {
  Application app = Application.Current!;
  app.RequestedThemeVariant = app.ActualThemeVariant == ThemeVariant.Dark ?
ThemeVariant.Light : ThemeVariant.Dark;
}
```

`RequestedThemeVariant` è il tema **chiesto** (`Default` vuol dire «come il sistema»), `ActualThemeVariant` quello **in uso**. Si può chiedere un tema anche per una sola finestra o un solo pannello: è quello che fa il programma delle figure, con `RequestedThemeVariant` sulla finestra.

Esercizi

1. Aggiungete una risorsa `"Errore"` (rosso chiaro nel tema scuro, rosso scuro nel chiaro) e usatela per una scritta d'avviso.
2. Mettete nella scheda un `ToggleSwitch` che chiama `CambiaTema`.

8. Elenchi e modelli

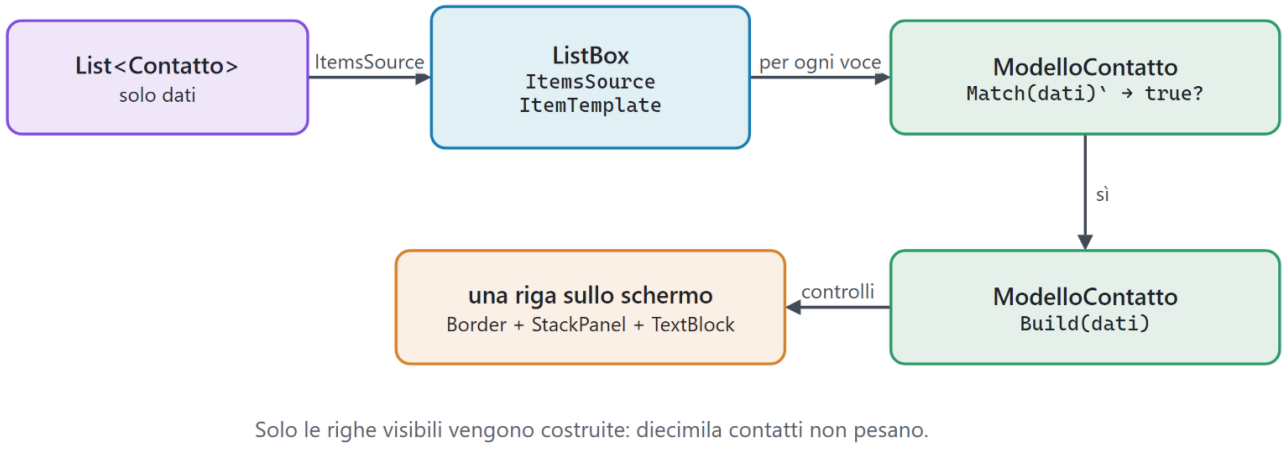
> **Cosa imparerai:** a separare i dati dalla loro rappresentazione, a mostrare un elenco di oggetti in un `ListBox` con un modello scritto come classe, e a reagire alla selezione.

8.1 Dati e modello

I dati non devono sapere niente di grafica. Un contatto è un `record`:

```
/// <summary>Un contatto della rubrica: solo dati.</summary>
public sealed record Contatto(string Nome, string Azienda, string Email,
uint Colore);
```

Chi sa disegnare un contatto è un **modello** (`IDataTemplate`), con due metodi: `Match` dice se il modello vale per quell'oggetto, `Build` costruisce i controlli di una riga.



Solo le righe visibili vengono costruite: diecimila contatti non pesano.

```
public sealed class ModelloContatto : IDataTemplate {

  /// <summary>Il modello vale per i Contatto.</summary>
  public bool Match(object? dati) => dati is Contatto;

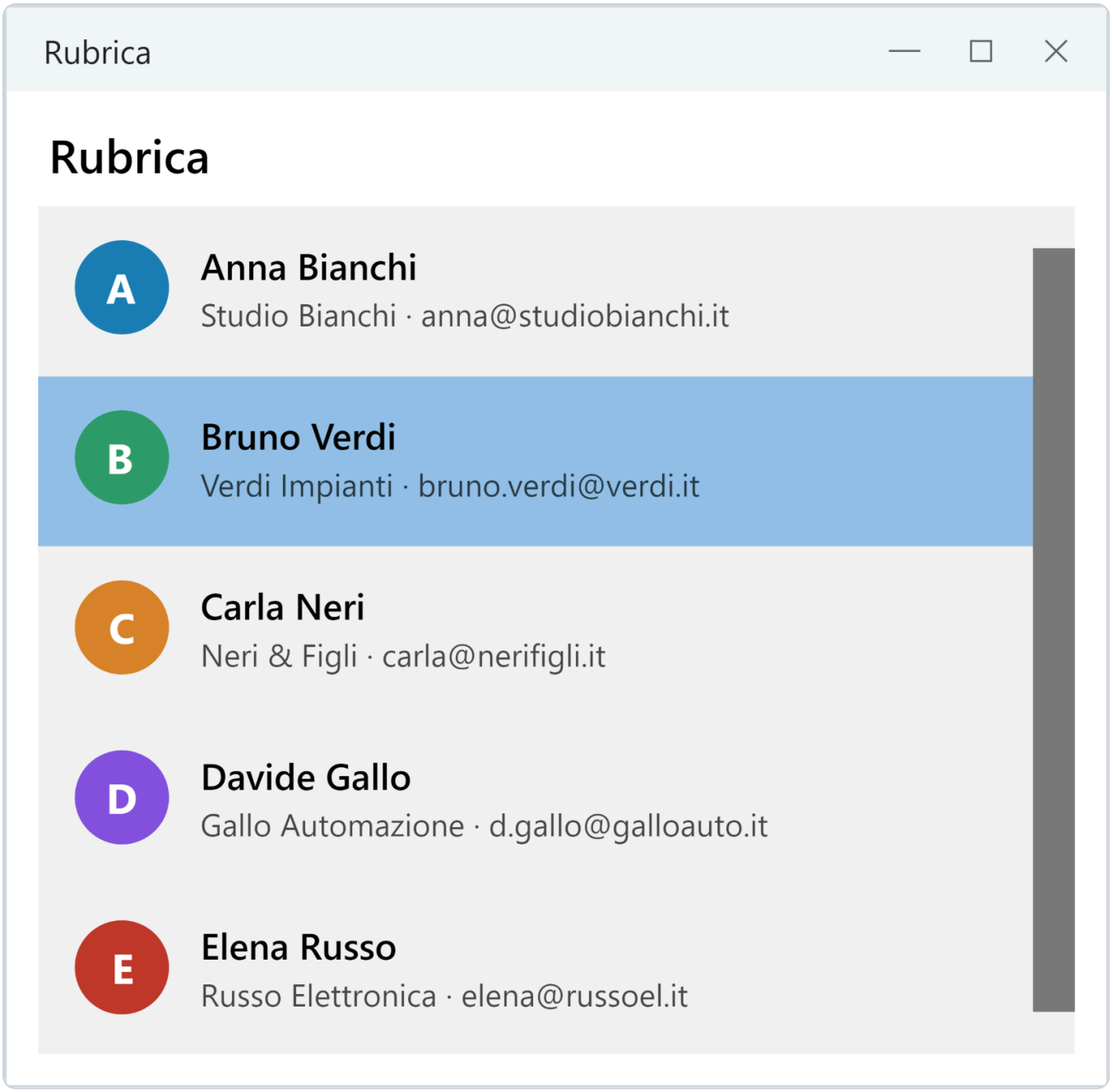
  /// <summary>I controlli di una riga: un tondo con l'iniziale, nome e
azienda, l'indirizzo.</summary>
  public Control? Build(object? dati) {
    if (dati is not Contatto c) return null;
    var iniziale = new Border { Width = 36, Height = 36, CornerRadius = new
CornerRadius(18), Background = new
SolidColorBrush(Color.FromUInt32(c.Colore)) }
      .Child(new TextBlock { Foreground = Brushes.White, FontWeight =
FontWeight.Bold, FontSize = 16 }
        .Text(c.Nome[..1]).Align(HorizontalAlignment.Center,
VerticalAlignment.Center));
    var testi = new StackPanel { Spacing = 1
}.VAlign(VerticalAlignment.Center).Children(
      new TextBlock { FontWeight = FontWeight.SemiBold }.Text(c.Nome),
      new TextBlock { FontSize = 12, Opacity = 0.7 }.Text(c.Azienda + " · "
+ c.Email));
    return new StackPanel { Orientation = Orientation.Horizontal, Spacing =
12 }.Margin(2, 4).Children(iniziale, testi);
  }
}
```

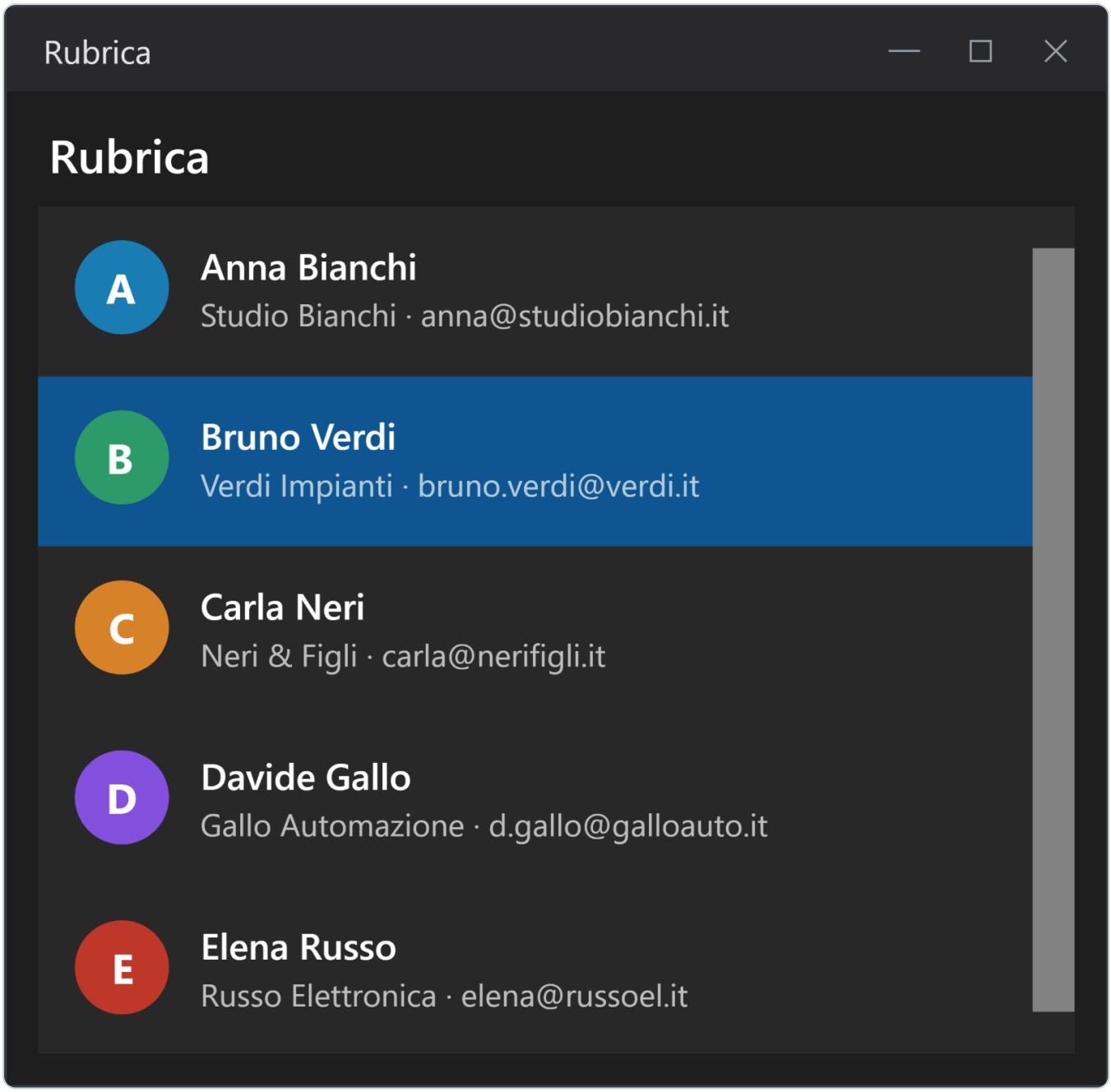
`dati is not Contatto c` è il **pattern matching**: controlla il tipo e, se va bene, dà all'oggetto il nome `c` già del tipo giusto. `c.Nome[..1]` è la prima lettera (un **intervallo**: dall'inizio al carattere 1 escluso).

8.2 Il ListBox

```
/// <summary>La rubrica in un ListBox, con il modello in una classe; la
seconda voce selezionata.</summary>
public static Control Lista() {
  var lista = new ListBox {
    ItemsSource = Rubrica(),
    ItemTemplate = new ModelloContatto(),
    SelectedIndex = 1,
  };
  lista.SelectionChanged += SelezioneCambiata;
  return new DockPanel().Margin(12).Children(
    new TextBlock { FontSize = 18, FontWeight = FontWeight.SemiBold
}.Text("Rubrica").Margin(4, 0, 0, 8).Dock(Dock.Top),
    lista);
}

// ListBox.SelectionChanged: qui si mostrerebbero i dettagli del contatto
scelto
private static void SelezioneCambiata(object? mittente,
SelectionChangedEventArgs e) {
  if (mittente is ListBox l && l.SelectedItem is Contatto c)
Console.WriteLine("scelto " + c.Nome);
}
```


Rubrica
Rubrica
A
Anna Bianchi
Studio Bianchi · anna@studiobianchi.it
B
Bruno Verdi
Verdi Impianti · bruno.verdi@verdi.it
C
Carla Neri
Neri & Figli · carla@nerifigli.it
D
Davide Gallo
Gallo Automazione · d.gallo@galloauto.it
E
Elena Russo
Russo Elettronica · elena@russoel.it



- `ItemsSource` riceve **i dati** (qualunque elenco), `ItemTemplate` **il modello**.
- Il `ListBox` costruisce solo le righe visibili (è *virtualizzato*): diecimila contatti non pesano.
- Per cambiare l'elenco si assegna di nuovo `ItemsSource`, come fa la lista della spesa. Se l'elenco cambia spesso, si usa un `ObservableCollection<T>`, che avvisa il `ListBox` a ogni aggiunta o rimozione.

Esercizi

1. Aggiungete a destra della rubrica una scheda dei dettagli che si aggiorna in `SelezioneCambiata`.
2. Aggiungete sopra l'elenco una `TextBox` che filtra i contatti per nome (un `foreach` che riempie una lista nuova).

9. Menu e finestre di dialogo

> **Cosa imparerai:** a costruire un menu con i tasti rapidi, ad aprire una finestra modale che restituisce una risposta, e a far scegliere un file all'utente.

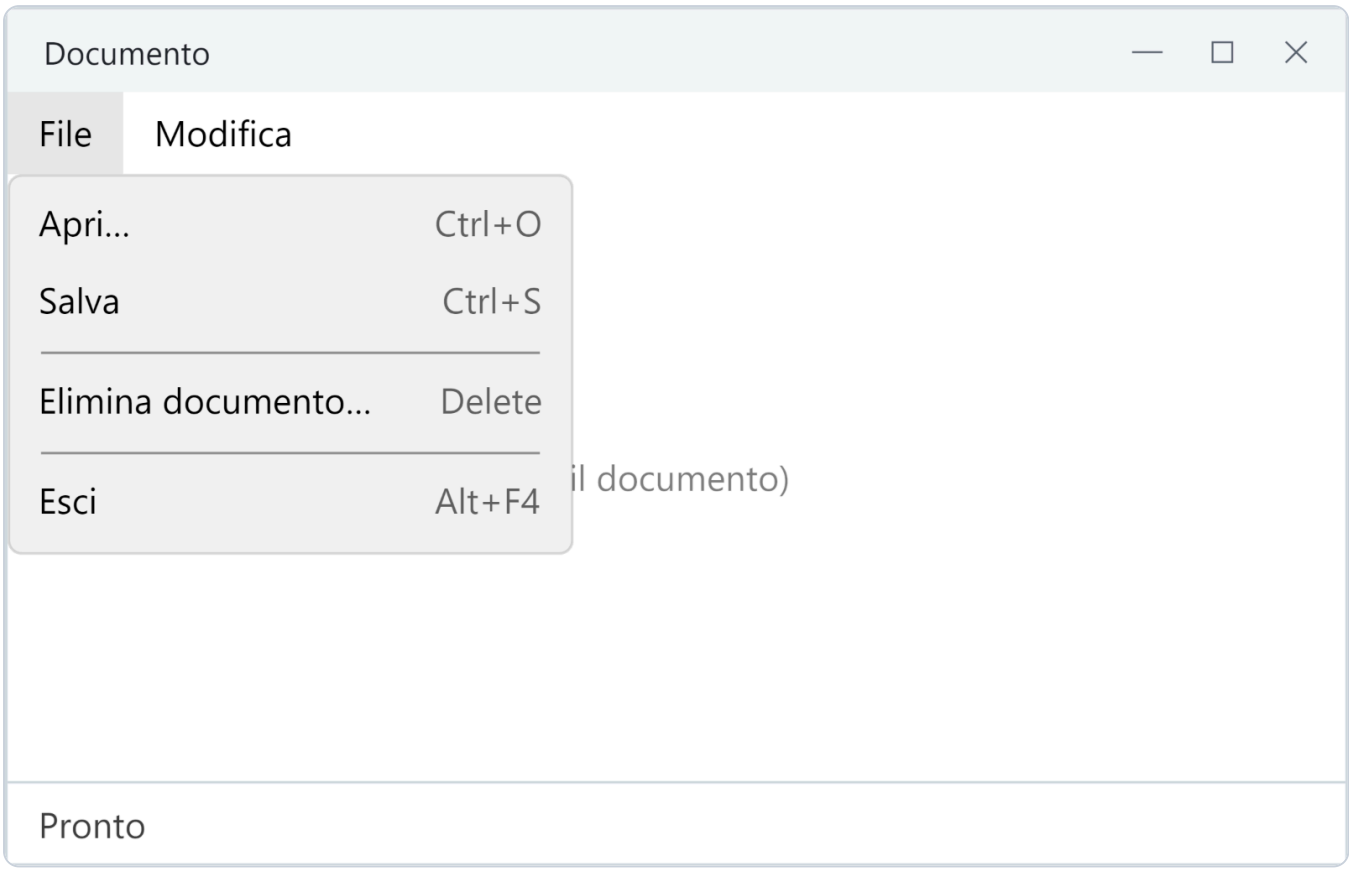
9.1 Il menu

```
/// <summary>Menu File e Modifica, ciascuna voce collegata a un metodo.
</summary>
public Dialoghi() {
  MenuFile = new MenuItem { Header = "_File" };
  MenuFile.Items.Add(Voce("_Apri…", "Ctrl+O", Apri));
  MenuFile.Items.Add(Voce("_Salva", "Ctrl+S", Salva));
  MenuFile.Items.Add(new Separator());
  MenuFile.Items.Add(Voce("_Elimina documento…", "Canc", Elimina));
  MenuFile.Items.Add(new Separator());
  MenuFile.Items.Add(Voce("_Esci", "Alt+F4", Esci));
  var modifica = new MenuItem { Header = "_Modifica" };
  modifica.Items.Add(Voce("_Copia", "Ctrl+C", Salva));
  modifica.Items.Add(Voce("_Incolla", "Ctrl+V", Salva));
  var menu = new Menu().Dock(Dock.Top);
  menu.Items.Add(MenuFile);
  menu.Items.Add(modifica);
  _stato.Text = "Pronto";
  Vista = new DockPanel().Children(
    menu,
    new Border { BorderThickness = new Thickness(0, 1, 0, 0)
}.Resource(Border.BorderBrushProperty,
"Bordo").Dock(Dock.Bottom).Child(_stato),
    new TextBlock { Opacity = 0.5 }.Text("(il
documento)").Align(HorizontalAlignment.Center, VerticalAlignment.Center));
}

// una voce di menu con il tasto rapido scritto a destra e il metodo da
chiamare
private static MenuItem Voce(string testo, string tasti, Action azione) {
  var voce = new MenuItem { Header = testo, InputGesture =
KeyGesture.Parse(tasti == "Canc" ? "Delete" : tasti) };
  voce.Click += new Azione(azione).Gestore;
  return voce;
}

// collega MenuItem.Click a un'azione senza argomenti (come fa RAva.OnClick
per i Button)
private sealed class Azione {
  private readonly Action _azione;
  // l'azione da chiamare
  public Azione(Action azione) { _azione = azione; }
  // MenuItem.Click
```

```
  public void Gestore(object? mittente, RoutedEventArgs e) => _azione();
}
```



- Il trattino basso in `"_File"` fa di **F** la lettera da premere con Alt.
- `InputGesture` **scrive** il tasto rapido a destra della voce; per farlo funzionare davvero si aggiunge alla finestra un `KeyBinding` (o si intercetta `KeyDown`).
- La piccola classe `Azione` è il modo di passare «quale metodo chiamare» senza una lambda: tiene l'azione in un campo e la chiama dal gestore. È lo stesso trucco che usa RAva dentro `OnClick`.

9.2 Una finestra di conferma

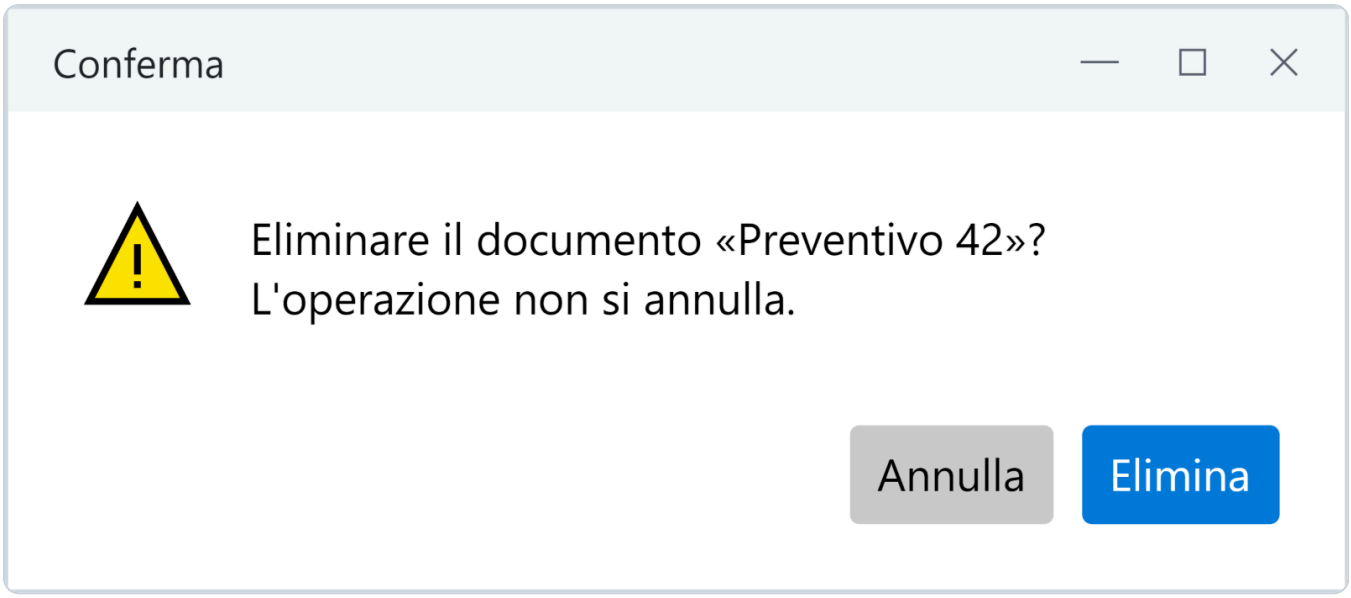
Una **finestra modale** blocca la finestra principale finché l'utente non risponde. `ShowDialog<T>` la apre e restituisce il valore passato a `Close`.

```
public sealed class FinestraConferma : Window {

  /// <summary>La domanda e il testo del pulsante che conferma.</summary>
  public FinestraConferma(string domanda, string conferma) {
    Title = "Conferma";
    Width = 420;
    SizeToContent = SizeToContent.Height;
    CanResize = false;
    WindowStartupLocation = WindowStartupLocation.CenterOwner;
    var si = new Button { Classes = { "accent" }, IsDefault = true
}.Content(conferma);
    var no = new Button { IsCancel = true }.Content("Annulla");
// Esc = Annulla
    si.Click += Conferma;
    no.Click += Annulla;
    Content = new DockPanel().Margin(20).Children(
      new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8,
HorizontalAlignment = HorizontalAlignment.Right }
        .Margin(0, 20, 0, 0).Dock(Dock.Bottom).Children(no, si),
      new TextBlock { Text = "⚠", FontSize = 30, Foreground =
Brushes.Orange }.Margin(0, 0, 14, 0).Dock(Dock.Left),
      new TextBlock { TextWrapping = TextWrapping.Wrap
}.Text(domanda).VAlign(VerticalAlignment.Center));
  }

  // «Elimina»: la finestra si chiude restituendo true
  private void Conferma(object? mittente, RoutedEventArgs e) => Close(true);

  // «Annulla» o Esc: la finestra si chiude restituendo false
  private void Annulla(object? mittente, RoutedEventArgs e) => Close(false);
}
```



E il metodo del menu che la usa:

```
// File → Elimina: si chiede conferma con una finestra modale
private async void Elimina() {
  if (TopLevel.GetTopLevel(Vista) is not Window proprietaria) return;
  bool ok = await new FinestraConferma("Eliminare il documento «Preventivo
42»? L'operazione non si annulla.", "Elimina")
    .ShowDialog<bool>(proprietaria);
  _stato.Text = ok ? "Documento eliminato" : "Eliminazione annullata";
}
```

- `TopLevel.GetTopLevel(controllo)` trova la finestra che contiene un controllo (capitolo 22).
- `async void` va bene **solo** per i gestori d'evento: il metodo aspetta la risposta senza bloccare la finestra, e quando l'utente chiude la conferma continua dalla riga dopo l'`await`.
- `IsDefault` e `IsCancel` legano Invio ed Esc ai due pulsanti.

9.3 Scegliere un file

```
// File → Apri: la finestra di sistema per scegliere un file
private async void Apri() {
  if (TopLevel.GetTopLevel(Vista) is not { } finestra) return;
  var opzioni = new FilePickerOpenOptions {
    Title = "Apri un documento",
    AllowMultiple = false,
    FileTypeFilter = new[] { new FilePickerFileType("Testo") { Patterns =
new[] { "*.txt", "*.md" } } },
  };
  IReadOnlyList<IStorageFile> scelti = await
finestra.StorageProvider.OpenFilePickerAsync(opzioni);
  _stato.Text = scelti.Count == 0 ? "Nessun file scelto" : "Aperto " +
scelti[0].Name;
}
```

`StorageProvider` apre la finestra **del sistema operativo** (quella di Windows, di macOS o di GNOME), con `OpenFilePickerAsync`, `SaveFilePickerAsync` e `OpenFolderPickerAsync`. Restituisce degli `IStorageFile`: il percorso vero si legge con `TryGetLocalPath()`.

Queste due finestre sono fatte a mano, per vedere come funzionano. Nei programmi veri RAVAL le apre in una riga: `RDialog.Confirm` e `RFiles.OpenAsync`, nel capitolo 18.

Esercizi

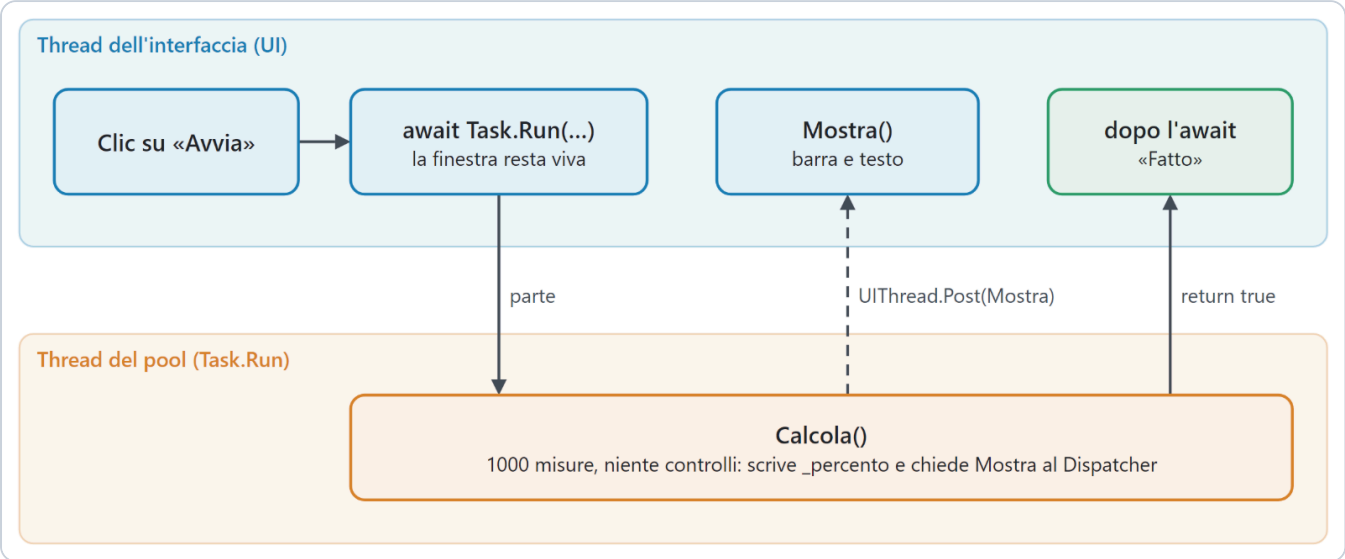
1. Collegate davvero Ctrl+S: `KeyBindings.Add(new KeyBinding { Gesture = KeyGesture.Parse("Ctrl+S"), Command = ... })` oppure, senza comandi, intercettate `KeyDown` della finestra.
2. Scrivete una `FinestraInput` che chiede un nome e restituisce la stringa scritta (`ShowDialog<string?>`).

10. Lavori lunghi: thread e Dispatcher

> **Cosa imparerai:** perché un calcolo lungo blocca la finestra, come spostarlo in un altro thread, e come riportare i risultati sullo schermo dal thread giusto. Thread, Task e `await` sono spiegati da capo nel capitolo 2: qui li mettiamo al lavoro in una finestra vera.

10.1 Il problema

Avalonia disegna, legge il mouse e la tastiera su **un solo thread**, il thread dell'interfaccia. Se un gestore d'evento lavora per dieci secondi, per dieci secondi la finestra non si ridisegna: sembra morta. E i controlli si possono toccare **solo** da quel thread: modificarli da un altro thread provoca un'eccezione.



La soluzione è in due mosse:

1. il lavoro lungo va in un thread del **pool** con `Task.Run`, e il gestore lo aspetta con `await`;
2. il lavoro, quando vuole mostrare qualcosa, chiede al **Dispatcher** di eseguire un metodo sul thread dell'interfaccia: `Dispatcher.UIThread.Post(Mostra)`.

10.2 L'esempio

```
public sealed class LavoroLungo {

  private readonly ProgressBar _barra = new() { Minimum = 0, Maximum = 100,
Height = 10 };
  private readonly TextBlock _testo = new() { FontSize = 14 };
  private readonly Button _avvia = new Button { Classes = { "accent" }
}.Content("Avvia il calcolo");
  private readonly Button _ferma = new Button { IsEnabled = false
}.Content("Ferma");
  private CancellationTokenSource? _annulla;
  private volatile int _percento;                                  // scritto
dal calcolo, letto da Mostra

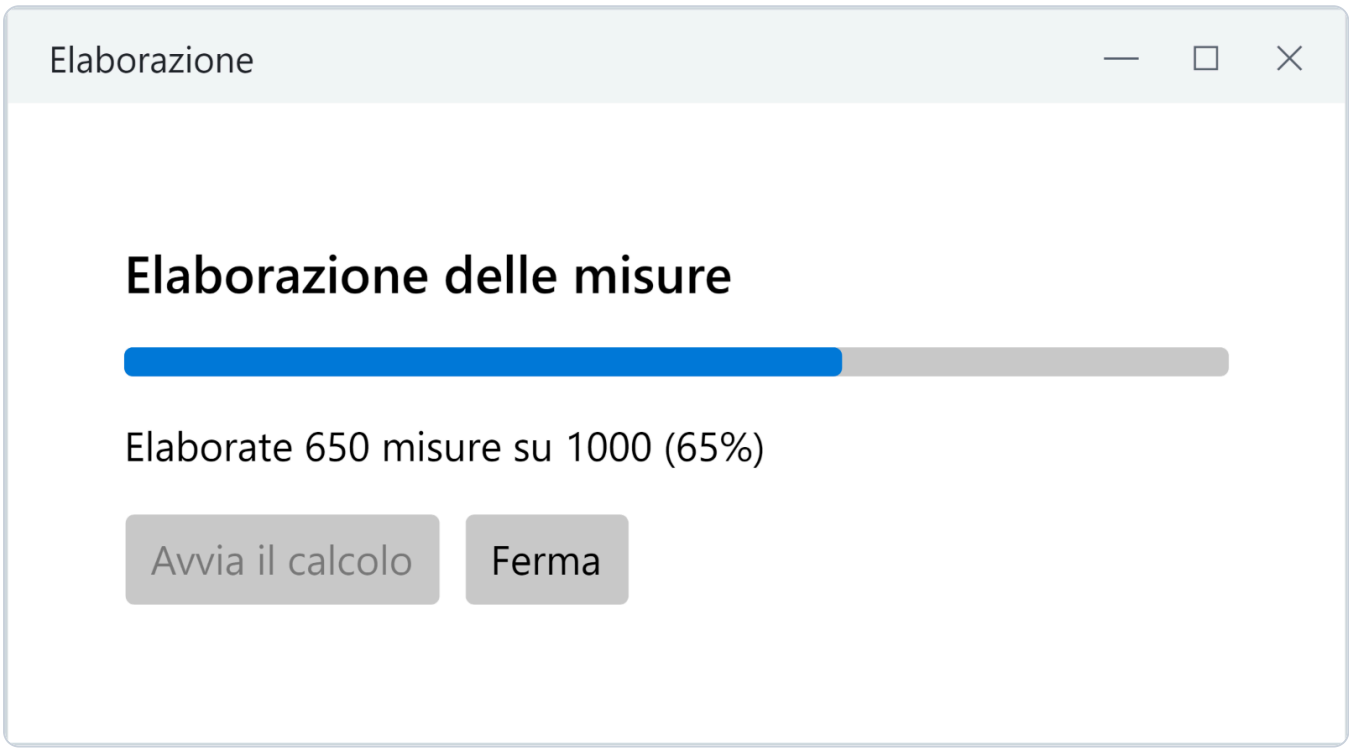
  // «Avvia»: il calcolo parte in un altro thread e qui si aspetta senza
bloccare la finestra
  private async void AvviaCliccato(object? mittente,
Avalonia.Interactivity.RoutedEventArgs e) {
    _annulla = new CancellationTokenSource();
    _avvia.IsEnabled = false;
    _ferma.IsEnabled = true;
    bool finito = await Task.Run(Calcola);                         // gruppo di
metodi
    _testo.Text = finito ? "Fatto: 1000 misure elaborate" : "Fermato
dall'utente";
    _avvia.IsEnabled = true;
    _ferma.IsEnabled = false;
  }

  // «Ferma»: chiede al calcolo di smettere
  private void FermaCliccato(object? mittente,
Avalonia.Interactivity.RoutedEventArgs e) => _annulla?.Cancel();

  // il lavoro lungo, nel thread del pool: niente controlli qui dentro
  private bool Calcola() {
    for (int i = 1; i <= 1000; i++) {
      if (_annulla!.IsCancellationRequested) return false;
      Thread.Sleep(5);                                             // finta
misura
      if (i % 10 == 0) {
        _percento = i / 10;
        Dispatcher.UIThread.Post(Mostra);                          // Mostra
girera' sul thread dell'interfaccia
      }
```

```
    }
    return true;
  }

  // sul thread dell'interfaccia: la barra e il testo seguono _percento
  private void Mostra() {
    _barra.Value = _percento;
    _testo.Text = $"Elaborate {_percento * 10} misure su 1000
({_percento}%)";
  }
}
```



Le regole, da ricordare a memoria:

- **dentro `Calcola` niente controlli**: solo dati. Il risultato si passa con un campo (`_percento`) o con il valore restituito;
- dopo `await Task.Run(...)` si è **di nuovo sul thread dell'interfaccia**: lì si possono toccare i controlli;
- `Post` mette in coda e torna subito; `Dispatcher.UIThread.InvokeAsync` fa lo stesso ma si può aspettare;
- per fermare un lavoro si usa un `CancellationTokenSource` : il lavoro controlla `IsCancellationRequested` e smette da sé. Non esiste un modo sano di «uccidere» un thread.
- `volatile` garantisce che il thread dell'interfaccia legga il valore aggiornato.

Esercizi

1. Mostrate anche il tempo stimato alla fine (`Stopwatch`).
2. Sostituite `Post(Mostra)` con un `IProgress<int>` (`new Progress<int>(MostraPercento)`): chi chiama il metodo di `Progress` sul thread giusto?

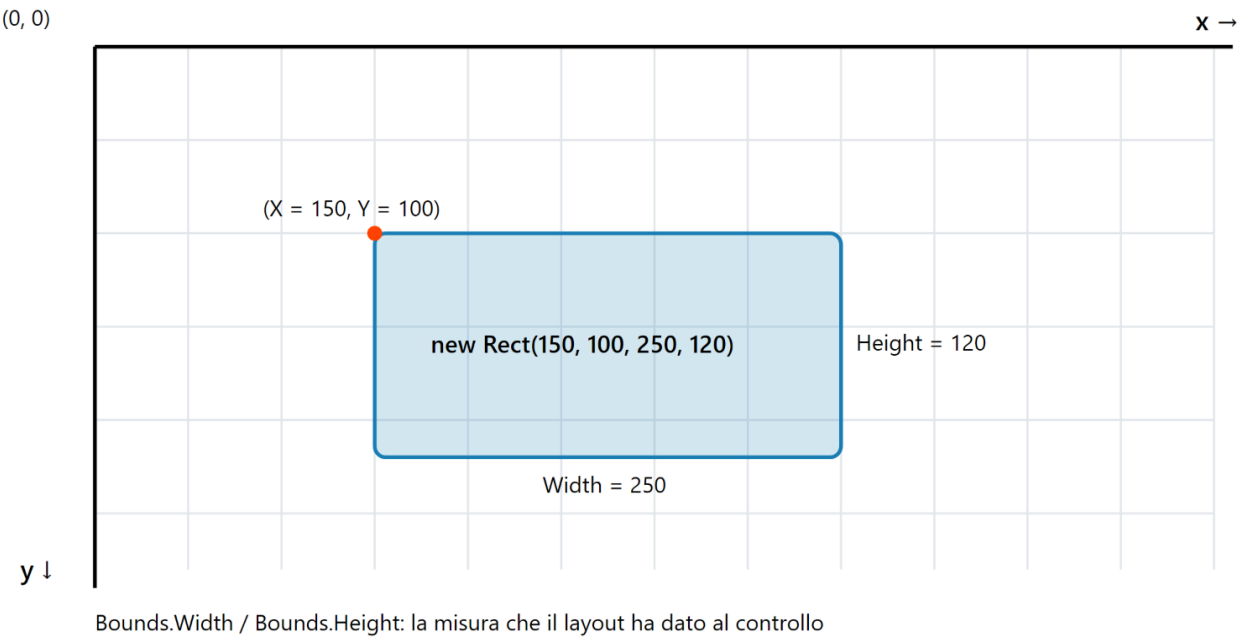
Parte III — Disegnare

11. Grafica disegnata a mano

> **Cosa imparerai:** a scrivere un controllo che disegna da sé, il sistema di coordinate, i pennelli, le penne, le geometrie, le sfumature e il testo; poi cinque controlli completi: barre, linee con area sfumata, indicatore circolare, torta e orologio.

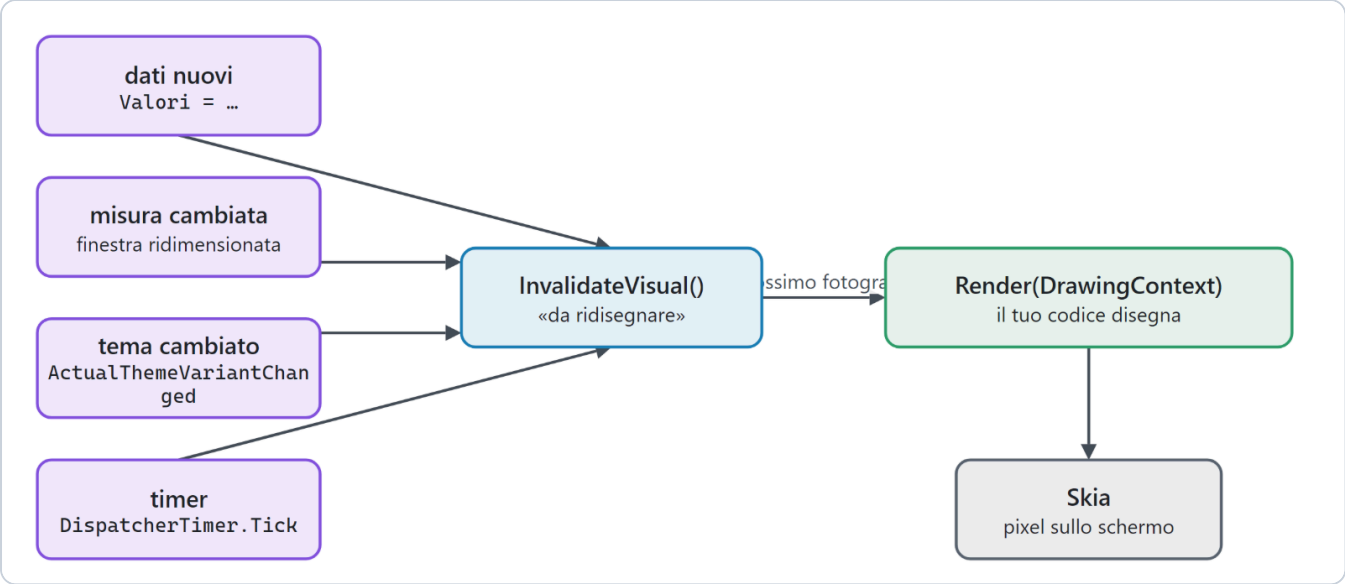
Quando un grafico, un indicatore o uno strumento non esistono fra i controlli pronti, si disegnano. Non serve una libreria di grafici: bastano una classe che eredita da `Control` e il metodo `Render`.

11.1 Coordinate



- L'origine `(0, 0)` è **in alto a sinistra** del controllo; la `x` cresce verso destra, la `y` **verso il basso**.
- Le misure sono in **pixel indipendenti dal dispositivo**: su uno schermo con scala 200% un pixel di Avalonia sono due pixel veri, e il disegno resta nitido.
- `Bounds.Width` e `Bounds.Height` sono la misura che il layout ha dato al controllo: il disegno deve adattarsi a quella, non a numeri fissi.

11.2 Quando viene chiamato Render



`Render` non si chiama mai direttamente: si chiede ad Avalonia di ridisegnare con `InvalidateVisual()`, e lei chiamerà `Render` al prossimo fotogramma (anche cento richieste diventano un disegno solo). Lo fa da sé quando cambia la misura. Il cambio di tema, invece, va segnalato: è il lavoro della nostra classe base.

11.3 La classe base Disegno

```
public abstract class Disegno : Control {

  /// <summary>Al cambio di tema si ridisegna, perche' i colori vengono
dalle risorse.</summary>
  protected Disegno() {
    ActualThemeVariantChanged += TemaCambiato;
  }

  // ActualThemeVariantChanged
  private void TemaCambiato(object? mittente, EventArgs e) =>
InvalidateVisual();

  /// <summary>Il pennello di una risorsa del tema (RAva.PennelloTema), con
un colore di ripiego.</summary>
  protected IBrush Tema(string chiave, uint ripiego) =>
this.ThemeBrush(chiave, new SolidColorBrush(Color.FromUInt32(ripiego)));

  /// <summary>Un testo pronto da disegnare con DrawText.</summary>
  protected static FormattedText Scritta(string testo, double dimensione,
IBrush colore, bool grassetto = false) =>
    new(testo, CultureInfo.CurrentCulture, FlowDirection.LeftToRight,
        new Typeface(FontFamily.Default, FontStyle.Normal, grassetto ?
FontWeight.SemiBold : FontWeight.Normal), dimensione, colore);

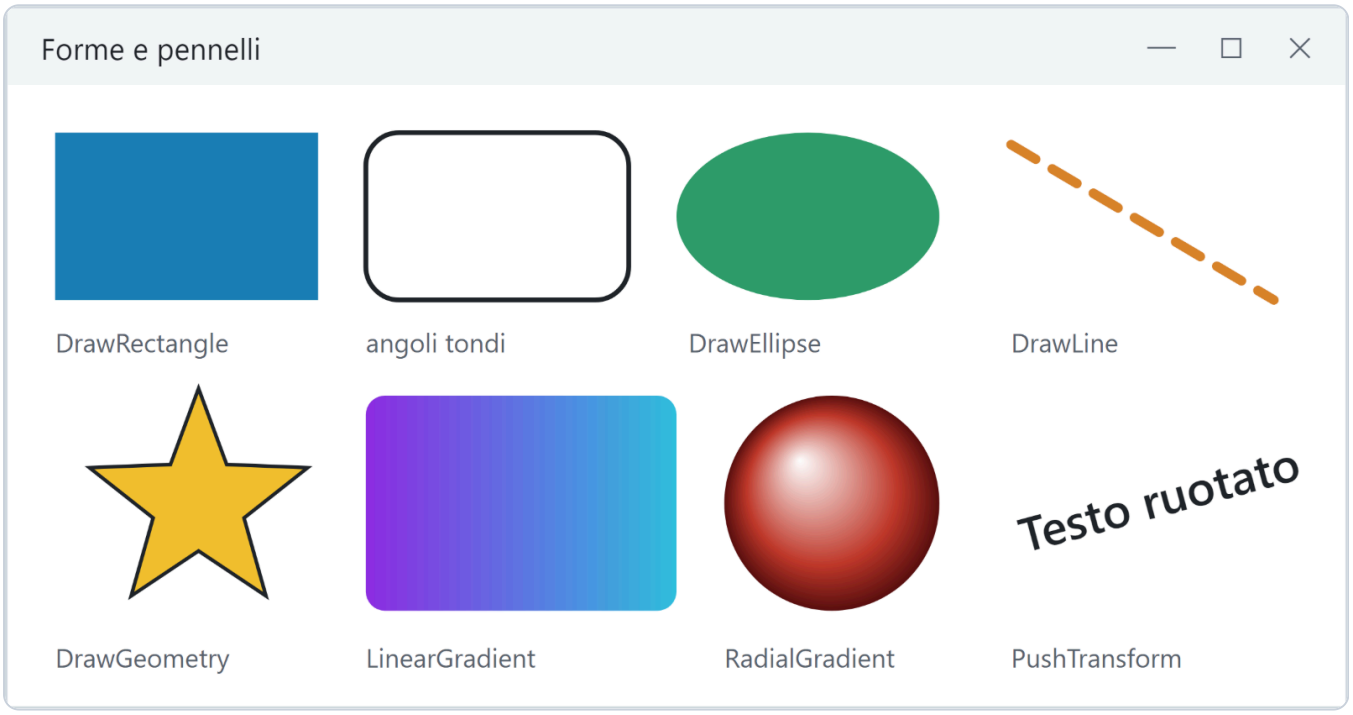
  /// <summary>Il punto sulla circonferenza: angolo in gradi, 0 in alto, in
senso orario (come le lancette).</summary>
  protected static Point SulCerchio(Point centro, double raggio, double
gradi) {
    double r = gradi * Math.PI / 180;
    return new Point(centro.X + raggio * Math.Sin(r), centro.Y - raggio *
Math.Cos(r));
  }

  /// <summary>Un arco di circonferenza da un angolo all'altro (gradi, 0 in
alto, senso orario).</summary>
  protected static StreamGeometry Arco(Point centro, double raggio, double
da, double a) {
    var g = new StreamGeometry();
    using (StreamGeometryContext c = g.Open()) {
      c.BeginFigure(SulCerchio(centro, raggio, da), false);
      c.ArcTo(SulCerchio(centro, raggio, a), new Size(raggio, raggio), 0, a
- da > 180, SweepDirection.Clockwise);
      c.EndFigure(false);
```

```
    }
    return g;
  }
}
```

- `ThemeBrush` di RAva cerca una risorsa nel **tema attuale del controllo**: in `Render` non si può usare `Resource` (che lega una proprietà), serve il pennello del momento.
- `SulCerchio` è un po' di trigonometria che tornerà utile in tre controlli: con l'angolo 0 in alto e il senso orario, gli angoli sono quelli dell'orologio.
- Una `StreamGeometry` è una **forma qualunque** fatta di segmenti e archi: si apre, si descrive con `BeginFigure`, `LineTo`, `ArcTo`, `EndFigure`, e si disegna con `DrawGeometry`.

11.4 Forme e pennelli



```
/// <summary>Disegna una forma per tipo, ciascuna con la sua didascalia.
</summary>
public override void Render(DrawingContext dc) {
  IBrush testo = Tema("Testo", 0xFF1F2328);
  var blu = new SolidColorBrush(Color.FromRgb(0x1A, 0x7F, 0xB5));
  var contorno = new Pen(testo, 2);

  dc.DrawRectangle(blu, null, new Rect(20, 20, 110, 70));
// rettangolo pieno
  dc.DrawRectangle(null, contorno, new Rect(150, 20, 110, 70), 14, 14);
// angoli tondi, solo bordo
  dc.DrawEllipse(new SolidColorBrush(Color.FromRgb(0x2E, 0x9E, 0x6A)), null,
new Point(335, 55), 55, 35);

  var tratteggio = new Pen(new SolidColorBrush(Color.FromRgb(0xD9, 0x82,
0x2B)), 4, new DashStyle(new double[] { 3, 2 }, 0), PenLineCap.Round);
  dc.DrawLine(tratteggio, new Point(420, 25), new Point(530, 90));
// linea tratteggiata

  var stella = new StreamGeometry();
// poligono: una stella a 5 punte
  using (StreamGeometryContext c = stella.Open()) {
    var centro = new Point(80, 175);
    for (int i = 0; i < 10; i++) {
      Point p = SulCerchio(centro, i % 2 == 0 ? 48 : 20, i * 36);
      if (i == 0) c.BeginFigure(p, true); else c.LineTo(p);
    }
    c.EndFigure(true);
  }
  dc.DrawGeometry(new SolidColorBrush(Color.FromRgb(0xF2, 0xC1, 0x2E)), new
Pen(testo, 1.5), stella);

  var lineare = new LinearGradientBrush {
// sfumatura da sinistra a destra
    StartPoint = new RelativePoint(0, 0, RelativeUnit.Relative),
    EndPoint = new RelativePoint(1, 0, RelativeUnit.Relative),
    GradientStops = { new GradientStop(Color.FromRgb(0x8E, 0x2D, 0xE2), 0),
new GradientStop(Color.FromRgb(0x30, 0xC0, 0xE0), 1) },
  };
  dc.DrawRectangle(lineare, null, new Rect(150, 130, 130, 90), 8, 8);

  var radiale = new RadialGradientBrush {
// sfumatura dal centro: una sfera
    GradientOrigin = new RelativePoint(0.35, 0.3, RelativeUnit.Relative),
    GradientStops = { new GradientStop(Colors.White, 0), new
```

```
GradientStop(Color.FromRgb(0xC0, 0x39, 0x2B), 0.7), new
GradientStop(Color.FromRgb(0x5A, 0x10, 0x10), 1) },
  };
  dc.DrawEllipse(radiale, null, new Point(345, 175), 45, 45);

  using (dc.PushTransform(Matrix.CreateRotation(-Math.PI / 12) *
Matrix.CreateTranslation(420, 175))) {   // testo ruotato
    dc.DrawText(Scritta("Testo ruotato", 20, testo, grassetto: true), new
Point(0, 0));
  }
  // ... le didascalie sotto le forme
}
```

Il vocabolario del disegno:

Oggetto	Cos'è	Esempio
`IBrush` (pennello)	come si **riempie**	`SolidColorBrush`, `LinearGradientBrush`, `RadialGradientBrush`
`Pen` (penna)	come si **traccia** un contorno	spessore, tratteggio (`DashStyle`), estremità (`PenLineCap`), giunti
`Rect`, `Point`, `Size`	posizioni e misure	`new Rect(x, y, larghezza, altezza)`
`Geometry`	una forma	`StreamGeometry`, `EllipseGeometry`, `RectangleGeometry`
`FormattedText`	un testo misurato	`Width`, `Height`, `Baseline` per allinearlo
`PushTransform`, `PushClip`, `PushOpacity`	stati temporanei	si chiudono uscendo dal `using`

Ogni metodo `Draw...` riceve **prima il pennello e poi la penna**: `null` per l'uno o per l'altra vuol dire «niente riempimento» o «niente contorno».

11.5 Il grafico a barre

```
public sealed class GraficoBarre : Disegno {

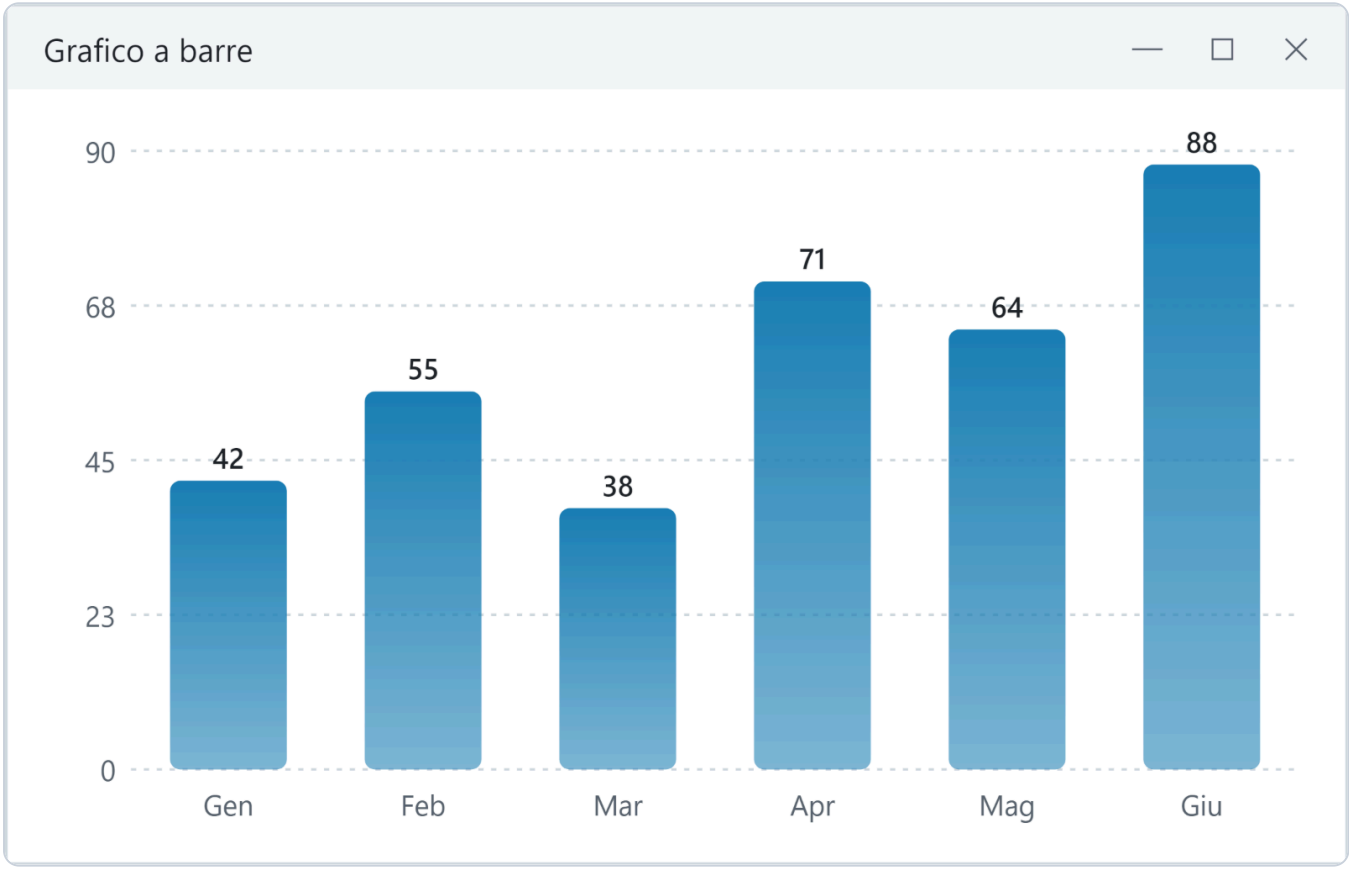
  private double[] _valori = Array.Empty<double>();

  /// <summary>Le etichette sotto le barre.</summary>
  public string[] Etichette { get; set; } = Array.Empty<string>();

  /// <summary>I valori: assegnarli ridisegna il grafico.</summary>
  public double[] Valori { get => _valori; set { _valori = value;
InvalidateVisual(); } }

  /// <summary>Griglia, barre sfumate, valori ed etichette.</summary>
  public override void Render(DrawingContext dc) {
    if (_valori.Length == 0) return;
    IBrush testo = Tema("Testo", 0xFF1F2328), tenue = Tema("TestoTenue",
0xFF59636E), bordo = Tema("Bordo", 0xFFD1D9E0);
    Color accento = Tema("Accento", 0xFF1A7FB5) is ISolidColorBrush s ?
s.Color : Colors.SteelBlue;
    const double sinistra = 40, alto = 16, basso = 28;
    double l = Bounds.Width - sinistra - 10, a = Bounds.Height - alto -
basso;
    double massimo = Math.Ceiling(_valori.Max() / 10) * 10;
    var griglia = new Pen(bordo, 1, new DashStyle(new double[] { 2, 3 },
0));
    for (int i = 0; i <= 4; i++) {
// linee orizzontali e scala
      double y = alto + a - a * i / 4;
      dc.DrawLine(griglia, new Point(sinistra, y), new Point(sinistra + l,
y));
      FormattedText scala = Scritta((massimo * i / 4).ToString("0"), 11,
tenue);
      dc.DrawText(scala, new Point(sinistra - 6 - scala.Width, y -
scala.Height / 2));
    }
    var pieno = new LinearGradientBrush {
      StartPoint = new RelativePoint(0, 0, RelativeUnit.Relative), EndPoint
= new RelativePoint(0, 1, RelativeUnit.Relative),
      GradientStops = { new GradientStop(accento, 0), new
GradientStop(Color.FromArgb(0x90, accento.R, accento.G, accento.B), 1) },
    };
    double passo = l / _valori.Length, barra = passo * 0.6;
    for (int i = 0; i < _valori.Length; i++) {
      double h = a * _valori[i] / massimo;
```

```
      double x = sinistra + i * passo + (passo - barra) / 2;
      dc.DrawRectangle(pieno, null, new Rect(x, alto + a - h, barra, h), 4,
4);
      FormattedText v = Scritta(_valori[i].ToString("0"), 11, testo,
grassetto: true);
      dc.DrawText(v, new Point(x + (barra - v.Width) / 2, alto + a - h -
v.Height - 2));
      if (i < Etichette.Length) {
        FormattedText e = Scritta(Etichette[i], 11, tenue);
        dc.DrawText(e, new Point(x + (barra - e.Width) / 2, alto + a + 6));
      }
    }
  }
}
```



Il disegno si ragiona in **tre misure**: lo spazio disponibile (`Bounds`), i margini per scala ed etichette, e la **scala** dei valori (qui il massimo arrotondato alla decina). Ogni valore diventa un'altezza con una proporzione: `h = a * valore / massimo` . Poiché la `y` cresce verso il basso, la barra parte da `alto + a - h` . Il setter di `Valori` chiama `InvalidateVisual` : cambiando i dati il grafico si ridisegna.

Si usa come un controllo qualunque:

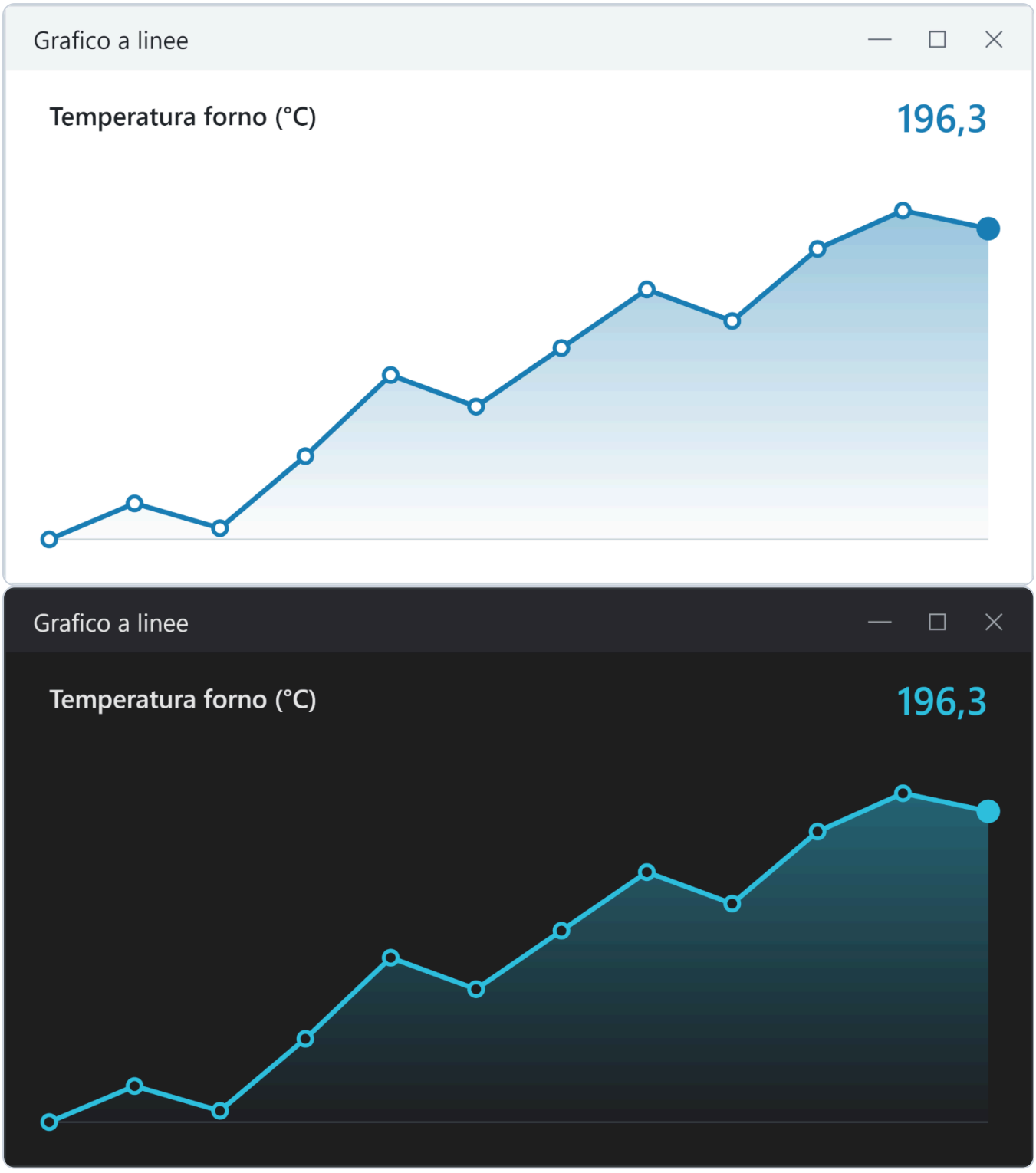
```
/// <summary>Le vendite per mese.</summary>
public static Control EsempioBarre() => new GraficoBarre {
  Etichette = new[] { "Gen", "Feb", "Mar", "Apr", "Mag", "Giu" },
  Valori = new double[] { 42, 55, 38, 71, 64, 88 },
}.Margin(8);
```

11.6 Linee con l’area sfumata

Il trucco dell'area sfumata è disegnare **due** geometrie: prima la linea chiusa verso il fondo, riempita con una sfumatura dall'accento al trasparente; poi la linea sola, sopra.

```
var area = new StreamGeometry();
// la linea chiusa verso il fondo
using (StreamGeometryContext c = area.Open()) {
  c.BeginFigure(new Point(punti[0].X, alto + a), true);
  foreach (Point p in punti) c.LineTo(p);
  c.LineTo(new Point(punti[^1].X, alto + a));
  c.EndFigure(true);
}
var sfumata = new LinearGradientBrush {
  StartPoint = new RelativePoint(0, 0, RelativeUnit.Relative), EndPoint =
new RelativePoint(0, 1, RelativeUnit.Relative),
  GradientStops = { new GradientStop(Color.FromArgb(0x70, accento.R,
accento.G, accento.B), 0), new GradientStop(Color.FromArgb(0x00, accento.R,
accento.G, accento.B), 1) },
};
dc.DrawGeometry(sfumata, null, area);

var linea = new StreamGeometry();
// la linea sola, sopra l'area
using (StreamGeometryContext c = linea.Open()) {
  c.BeginFigure(punti[0], false);
  for (int i = 1; i < punti.Length; i++) c.LineTo(punti[i]);
  c.EndFigure(false);
}
dc.DrawGeometry(null, new Pen(new SolidColorBrush(accento), 2.5, lineJoin:
PenLineJoin.Round), linea);
IBrush sfondo = Tema("Sfondo", 0xFFFFFFFF);
foreach (Point p in punti) dc.DrawEllipse(sfondo, new Pen(new
SolidColorBrush(accento), 2), p, 3.5, 3.5);
dc.DrawEllipse(new SolidColorBrush(accento), null, punti[^1], 6, 6);
```



`punti[^1]` è l'**ultimo** elemento dell'array (l'indice `^1` conta dalla fine). I tondini dei punti sono riempiti con il colore di sfondo del tema: sembrano «bucati» in entrambi i temi.

11.7 L’indicatore circolare

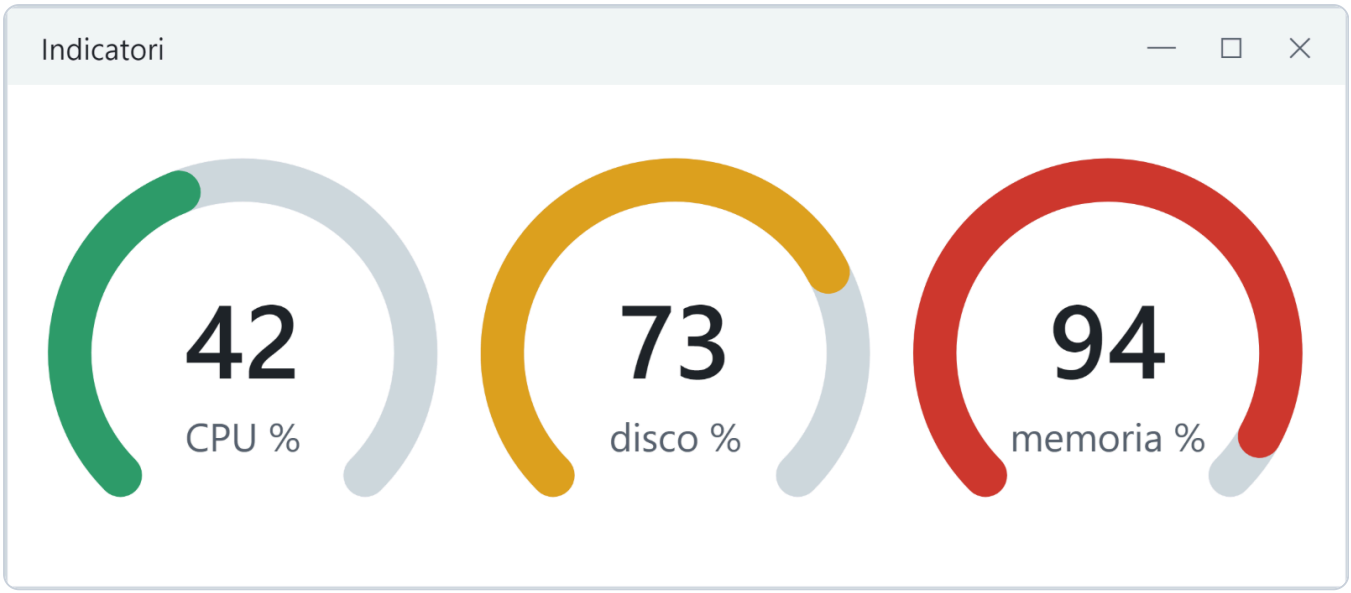
```
public sealed class Indicatore : Disegno {

  private double _valore;

  /// <summary>Il valore da 0 a 100.</summary>
  public double Valore { get => _valore; set { _valore = Math.Clamp(value,
0, 100); InvalidateVisual(); } }

  /// <summary>La scritta sotto il numero.</summary>
  public string Unita { get; set; } = "%";

  /// <summary>Arco di fondo, arco del valore (verde, giallo o rosso),
numero e unita'.</summary>
  public override void Render(DrawingContext dc) {
    double lato = Math.Min(Bounds.Width, Bounds.Height);
    var centro = new Point(Bounds.Width / 2, Bounds.Height / 2 + lato *
0.04);
    double spessore = lato * 0.1, raggio = lato / 2 - spessore;
    var fondo = new Pen(Tema("Bordo", 0xFFD1D9E0), spessore, lineCap:
PenLineCap.Round);
    dc.DrawGeometry(null, fondo, Arco(centro, raggio, -135, 135));
    Color colore = _valore < 60 ? Color.FromRgb(0x2E, 0x9E, 0x6A) : _valore
< 85 ? Color.FromRgb(0xE0, 0xA0, 0x20) : Color.FromRgb(0xD0, 0x3A, 0x2F);
    if (_valore > 0.5) dc.DrawGeometry(null, new Pen(new
SolidColorBrush(colore), spessore, lineCap: PenLineCap.Round), Arco(centro,
raggio, -135, -135 + 270 * _valore / 100));
    FormattedText numero = Scritta(_valore.ToString("0"), lato * 0.24,
Tema("Testo", 0xFF1F2328), grassetto: true);
    dc.DrawText(numero, new Point(centro.X - numero.Width / 2, centro.Y -
numero.Height / 2 - lato * 0.04));
    FormattedText unita = Scritta(Unita, lato * 0.09, Tema("TestoTenue",
0xFF59636E));
    dc.DrawText(unita, new Point(centro.X - unita.Width / 2, centro.Y + lato
* 0.13));
  }
}
```



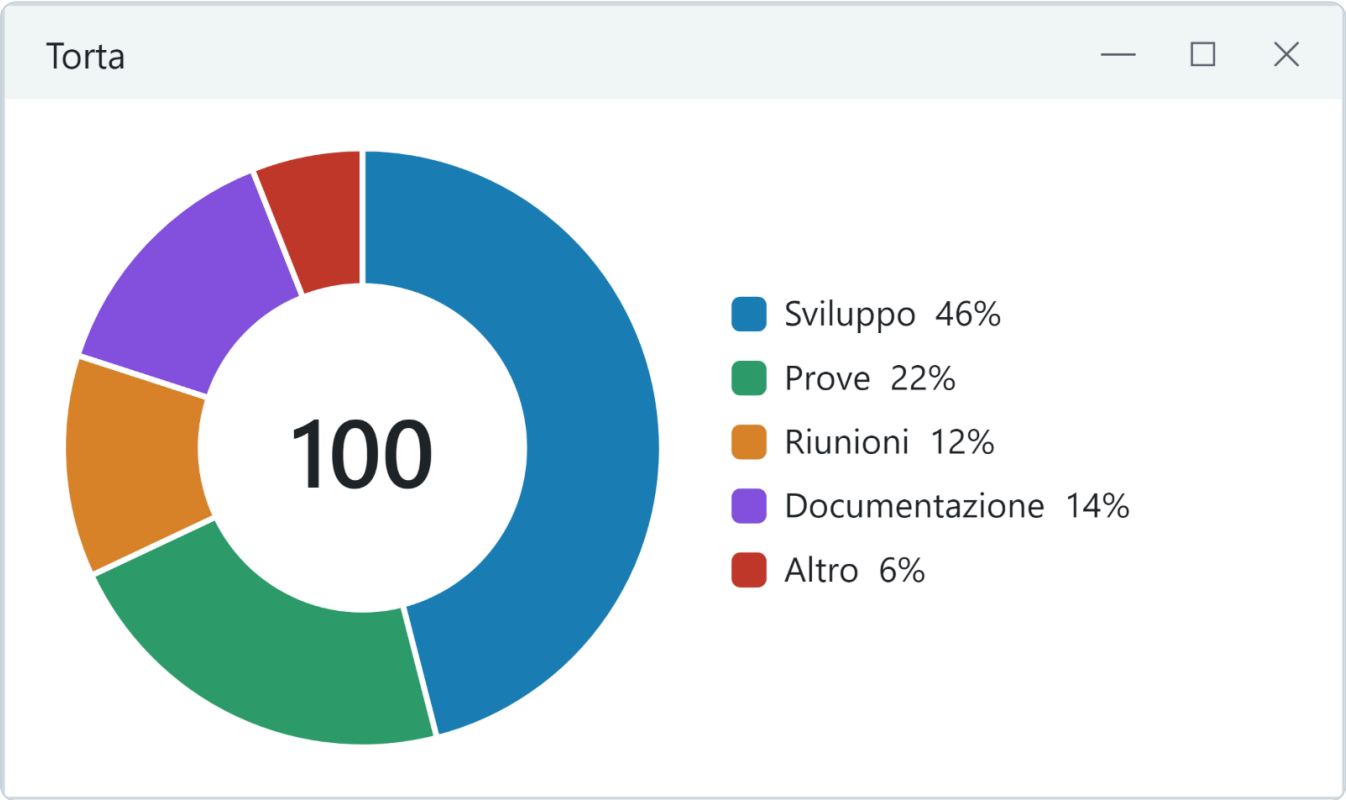
L'arco va da −135° a +135° (270° in tutto, con l'apertura in basso); il valore ne riempie una parte proporzionale. Le misure sono **frazioni del lato** (`lato * 0.1`, `lato * 0.24`): l'indicatore resta proporzionato a qualunque misura. Per mettere tre controlli uguali in fila si usa `UniformGrid`:

```
new UniformGrid { Columns = 3 }.Margin(8).Children(
  new Indicatore { Valore = 42, Unita = "CPU %" },
  new Indicatore { Valore = 73, Unita = "disco %" },
  new Indicatore { Valore = 94, Unita = "memoria %" });
```

11.8 La torta

Ogni spicchio è una geometria: dal centro al bordo, un arco, e di nuovo al centro.

```
double inizio = 0;
for (int i = 0; i < Voci.Length; i++) {
  double ampiezza = 360 * Voci[i].Valore / totale;
  var spicchio = new StreamGeometry();
  using (StreamGeometryContext c = spicchio.Open()) {
    c.BeginFigure(centro, true);
    c.LineTo(SulCerchio(centro, raggio, inizio));
    c.ArcTo(SulCerchio(centro, raggio, inizio + ampiezza), new Size(raggio,
raggio), 0, ampiezza > 180, SweepDirection.Clockwise);
    c.EndFigure(true);
  }
  dc.DrawGeometry(new SolidColorBrush(Color.FromUInt32(Palette[i %
Palette.Length])), new Pen(sfondo, 2), spicchio);
  inizio += ampiezza;
}
dc.DrawEllipse(sfondo, null, centro, raggio * 0.55, raggio * 0.55);
// il buco della ciambella
```



Il parametro `isLargeArc` di `ArcTo` (`ampiezza > 180`) dice quale dei due archi possibili disegnare fra due punti: senza, uno spicchio di più di mezza torta verrebbe disegnato dalla parte sbagliata. Il contorno degli spicchi ha il colore dello sfondo: li separa con una riga sottile in entrambi i temi. Le voci sono una **tupla** con nomi, `(string Nome, double Valore)[]`, comoda per dati piccoli.

11.9 L'orologio, con un timer

```
public sealed class Orologio : Disegno {

  private readonly Avalonia.Threading.DispatcherTimer _timer = new() {
Interval = TimeSpan.FromSeconds(1) };

  /// <summary>L'ora mostrata; null = l'ora vera, aggiornata ogni secondo.
</summary>
  public TimeSpan? OraFissa { get; set; }

  /// <summary>Il timer che fa avanzare le lancette.</summary>
  public Orologio() {
    _timer.Tick += Scatta;
  }

  /// <summary>Il timer parte quando l'orologio entra in una finestra...
</summary>
  protected override void
OnAttachedToVisualTree(VisualTreeAttachmentEventArgs e) {
    base.OnAttachedToVisualTree(e);
    if (OraFissa == null) _timer.Start();
  }

  /// <summary>...e si ferma quando ne esce, altrimenti continuerebbe a
girare per niente.</summary>
  protected override void
OnDetachedFromVisualTree(VisualTreeAttachmentEventArgs e) {
    _timer.Stop();
    base.OnDetachedFromVisualTree(e);
  }

  // DispatcherTimer.Tick: un secondo in piu'
  private void Scatta(object? mittente, EventArgs e) => InvalidateVisual();

  /// <summary>Quadrante, tacche, numeri, lancette.</summary>
  public override void Render(DrawingContext dc) {
    TimeSpan ora = OraFissa ?? DateTime.Now.TimeOfDay;
    double raggio = Math.Min(Bounds.Width, Bounds.Height) / 2 - 4;
    var c = new Point(Bounds.Width / 2, Bounds.Height / 2);
    IBrush testo = Tema("Testo", 0xFF1F2328);
    dc.DrawEllipse(Tema("Pannello", 0xFFF3F5F8), new Pen(testo, 3), c,
raggio, raggio);
    for (int i = 0; i < 60; i++) {
// tacche: grosse ogni 5 minuti
```

```
      bool ora5 = i % 5 == 0;
      dc.DrawLine(new Pen(testo, ora5 ? 3 : 1), SulCerchio(c, raggio * (ora5
? 0.84 : 0.9), i * 6), SulCerchio(c, raggio * 0.95, i * 6));
    }
    for (int h = 1; h <= 12; h++) {
      FormattedText n = Scritta(h.ToString(), raggio * 0.15, testo,
grassetto: true);
      Point p = SulCerchio(c, raggio * 0.68, h * 30);
      dc.DrawText(n, new Point(p.X - n.Width / 2, p.Y - n.Height / 2));
    }
    double secondi = ora.Seconds, minuti = ora.Minutes + secondi / 60, ore =
ora.Hours % 12 + minuti / 60;
    Lancetta(dc, c, raggio * 0.5, ore * 30, 6, testo);
    Lancetta(dc, c, raggio * 0.78, minuti * 6, 4, testo);
    Lancetta(dc, c, raggio * 0.85, secondi * 6, 1.5, Brushes.OrangeRed);
    dc.DrawEllipse(Brushes.OrangeRed, null, c, 5, 5);
  }

  // una lancetta dal centro (un po' oltre, dall'altra parte) verso l'angolo
dato
  private static void Lancetta(DrawingContext dc, Point c, double lunghezza,
double gradi, double spessore, IBrush colore) =>
    dc.DrawLine(new Pen(colore, spessore, lineCap: PenLineCap.Round),
SulCerchio(c, -lunghezza * 0.15, gradi), SulCerchio(c, lunghezza, gradi));
}
```


Orologio
12
1
2
3
4
5
6
7
8
9
10
11



- Un `DispatcherTimer` chiama il suo `Tick` **sul thread dell'interfaccia**: dentro si possono toccare i controlli senza Dispatcher.
- `OnAttachedToVisualTree` e `OnDetachedFromVisualTree` arrivano quando il controllo entra o esce da una finestra: è il posto giusto per avviare e fermare le cose che consumano.
- Le ore avanzano anche con i minuti (`ore = h + minuti / 60`): la lancetta corta alle 10:08 non sta esattamente sul 10.

Esercizi

1. Aggiungete al grafico a barre una linea orizzontale tratteggiata con la media dei valori.
2. Fate un `Termometro`: un rettangolo arrotondato che si riempie dal basso, con la scala a sinistra.
3. Rendete cliccabile la torta: in `OnPointerPressed` calcolate l'angolo del punto (`Math.Atan2`) e trovate lo spicchio.

Parte IV — I componenti di RAVAL

12. Le estensioni di RAva

> **Cosa imparerai:** tutte le estensioni di `RAVAL.RAva` (disposizione, testo e cornici, pannelli pronti, eventi senza lambda, colori e temi), e perché si chiamano come le proprietà di Avalonia.

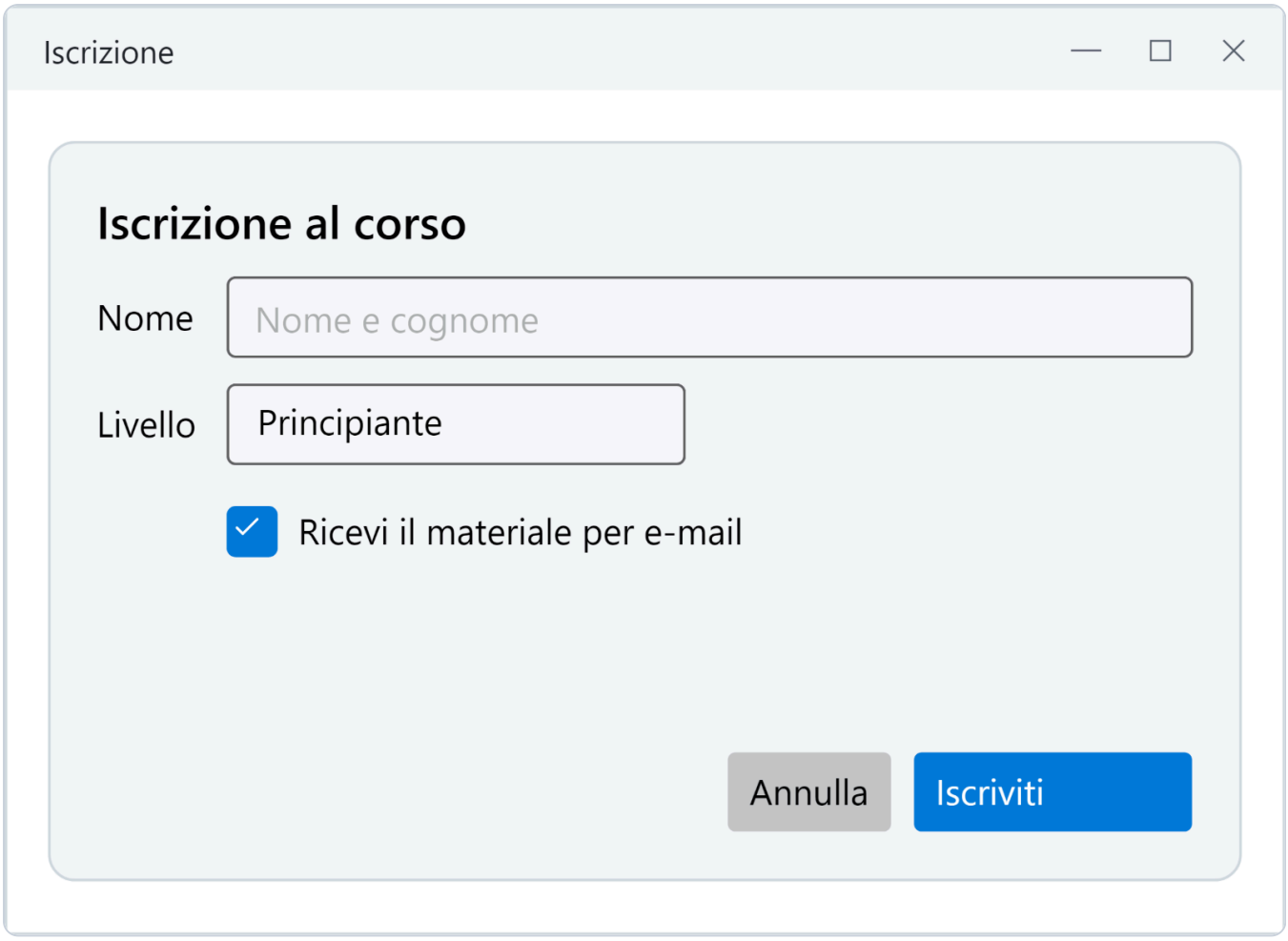
Le estensioni stanno in `RAVAL\UI\`: `RAva.cs` e, per gruppo, `RAvaText.cs`, `RAvaPanels.cs`, `RAvaEvents.cs`, `RAvaColors.cs`; i colori dei temi in `RThemes.cs`. Ognuna assegna una proprietà o aggiunge qualcosa al controllo e **restituisce il controllo stesso**, del suo tipo vero (sono generiche, `T Margine<T>(this T c, ...)`): dopo `new TextBox().Margin(4)` si ha ancora una `TextBox`.

Estensione	Vale per	Fa	Equivale a
`Margin(tutti)`	ogni `Layoutable`	stesso margine sui 4 lati	`Margin = new Thickness(t)`
`Margin(orizz, vert)`	ogni `Layoutable`	margine orizzontale e verticale	`new Thickness(o, v)`
`Margin(s, a, d, b)`	ogni `Layoutable`	sinistra, alto, destra, basso	`new Thickness(s, a, d, b)`
`Align(o, v)`	ogni `Layoutable`	allineamento orizzontale e verticale	`HorizontalAlignment`, `VerticalAlignment`
`VAlign(v)`	ogni `Layoutable`	solo verticale	`VerticalAlignment`
`Width(l)`, `Height(a)`	ogni `Layoutable`	misura fissa	`Width`, `Height`
`Column(c, span)`	ogni `Control`	colonna della `Grid` (e quante)	`Grid.SetColumn`, `Grid.SetColumnSpan`
`Row(r, span)`	ogni `Control`	riga della `Grid` (e quante)	`Grid.SetRow`, `Grid.SetRowSpan`
`Dock(lato)`	ogni `Control`	lato del `DockPanel`	`DockPanel.SetDock`
`Children(...)`	ogni `Panel`	aggiunge i figli	`Children.AddRange`
`Text(s)`	`TextBlock`	il testo	`Text`
`Content(o)`	`ContentControl` (Button, CheckBox…)	il contenuto; con «_Prenota»	`Content`

Estensione	Vale per	Fa	Equivale a
		anche il nome accessibile «Prenota»	
`Child(c)`	`Border`, ogni `Decorator`	l'unico figlio	`Child`
`ToolTip(s)`	ogni `Control`	il testo al passaggio del mouse	`ToolTip.SetTip`
`AutomationName(s)`	ogni `Control`	il nome che annuncia il lettore di schermo (per i pulsanti con la sola icona)	`AutomationProperties.SetName`
`OnClick(azione)`	`Button`	un metodo senza argomenti al clic	`Click += ...`
`Resource(prop, chiave)`	ogni `Control`	lega la proprietà a una risorsa del tema	`Bind(prop, GetResourceObservable(chiave)`
`ThemeBrush(chiave, ripiego)`	ogni `Control`	il pennello della risorsa nel tema attuale (per `Render`)	`TryFindResource(chiave, ActualThemeVariant, ...)`

Un modulo che le usa tutte:

```
/// <summary>Un modulo di iscrizione che usa ogni estensione di RAva.
</summary>
public static Control ModuloRAva() {
  var g = new Grid { ColumnDefinitions = new ColumnDefinitions("Auto,*"),
RowDefinitions = new RowDefinitions("Auto,Auto,Auto,Auto,*"), ColumnSpacing
= 12, RowSpacing = 10 };
  g.Children(
    new TextBlock { FontSize = 18, FontWeight = FontWeight.SemiBold
}.Text("Iscrizione al corso").Row(0).Column(0, span: 2),
    new TextBlock().Text("Nome").VAlign(VerticalAlignment.Center).Row(1),
    new TextBox { PlaceholderText = "Nome e cognome"
}.Row(1).Column(1).ToolTip("Come compare sull'attestato"),
    new TextBlock().Text("Livello").VAlign(VerticalAlignment.Center).Row(2),
    new ComboBox { ItemsSource = new[] { "Principiante", "Intermedio",
"Esperto" }, SelectedIndex = 0 }.Width(180).Row(2).Column(1),
    new CheckBox { IsChecked = true }.Content("Ricevi il materiale per e-
mail").Row(3).Column(1),
    new StackPanel { Orientation = Orientation.Horizontal, Spacing = 8 }
      .Align(HorizontalAlignment.Right,
VerticalAlignment.Bottom).Row(4).Column(0, span: 2)
      .Children(
        new Button().Content("Annulla").OnClick(Annulla),
        new Button { Classes = { "accent" }
}.Content("Iscriviti").Width(110).OnClick(Iscrivi)));
  return new Border { CornerRadius = new CornerRadius(10), BorderThickness =
new Thickness(1), Padding = new Thickness(18) }
    .Resource(Border.BackgroundProperty,
"Pannello").Resource(Border.BorderBrushProperty, "Bordo")
    .Margin(16).Height(290).Child(g);
}
```



Testo e cornici

In Avalonia le stesse proprietà stanno su classi diverse: `FontSize` c'è nel `TextBlock` e nei controlli con modello (`Button` , `TextBox` , `CheckBox` … cioè i `TemplatedControl`), `Padding` nel `Border` , nel `TextBlock` e nei controlli con modello ma **non** nei pannelli. Queste estensioni valgono per ogni `Control` e guardano il tipo vero quando le si chiama. Se il controllo non ha la proprietà, si fermano con un errore che dice per chi valgono: un'impostazione ignorata in silenzio si scoprirebbe solo guardando la finestra.

Estensione	Vale per	Fa
`FontSize(px)`	`TextBlock`, controlli con modello, pannelli	dimensione del carattere; su un pannello la ereditano i figli
`FontWeight(p)`, `Bold()`, `Italic()`	come sopra	peso e stile del carattere
`FontFamily("Consolas")`	come sopra	la famiglia del carattere
`Foreground(pennello)`, `Foreground("#1a7fb5")`	come sopra	il colore del testo
`Wrap()`, `TextAlign(a)`	`TextBlock`	il testo va a capo; allineamento delle righe
`Border(spessore, raggio)`, `CornerRadius(r)`	`Border`, controlli con modello	spessore del bordo e angoli arrotondati
`BorderBrush(...)`	`Border`, controlli con modello	il colore del bordo
`Padding(...)`	`Border`, `TextBlock`, controlli con modello	lo spazio fra il bordo e il contenuto (1, 2 o 4 valori, come `Margin`)
`Background(...)`	pannelli, `Border`, `TextBlock`, controlli con modello	lo sfondo
`Visible(b)`, `Enabled(b)`, `Opacity(o)`	ogni `Control`	`IsVisible`, `IsEnabled`, `Opacity`
`Classes("accent", ...)`	ogni `Control`	aggiunge classi di stile

I colori si possono scrivere come in HTML: `"#1a7fb5"`, `"#801a7fb5"` (le prime due cifre sono l'opacità), `"Red"`. Per un colore fisso c'è anche `RAva.Brush("#1a7fb5")` o `RAva.Brush(0xFF1A7FB5)`; per un colore che segue il tema si usa sempre `Resource` (capitolo 7).

Pannelli pronti

Metodo	Crea
`RAva.HStack(spazio, figli...)`	uno `StackPanel` orizzontale
`RAva.VStack(spazio, figli...)`	uno `StackPanel` verticale
`RAva.Grid("Auto,*", "Auto,Auto,*")`	una `Grid` con colonne e righe (le righe si possono omettere)
`.Spacing(s)` / `.Spacing(colonne, righe)`	lo spazio fra i figli di uno `StackPanel` / di una `Grid`

Non si chiamano `Row` e `Column` perché quei nomi sono già le estensioni che mettono un controllo in una riga o colonna della `Grid`.

Eventi senza lambda

Come `OnClick`, si passa un **metodo con un nome**, che riceve già il valore utile invece di mittente ed `EventArgs` da convertire. Dietro ogni estensione c'è una piccola classe che ricorda il controllo e il metodo: la stessa tecnica del capitolo 6, scritta una volta per tutte.

Estensione	Vale per	Il metodo riceve
`OnTextChanged(metodo)`	`TextBox`	il testo nuovo, a ogni lettera (`void Filtra(string testo)`)
`OnEnter(metodo)`	`TextBox`	niente: è stato premuto Invio (ascoltato in fase Tunnel, prima della casella: vale anche con `AcceptsReturn` ; capitolo 6)
`OnChecked(metodo)`	`CheckBox` , `ToggleSwitch` , `RadioButton`	lo stato nuovo (`void Attiva(bool spuntato)`)
`OnSelectionChanged(metodo)`	`ListBox` , `ComboBox`	l'elemento scelto, o null (`void Mostra(object? scelto)`)
`OnThemeChanged(metodo)`	ogni `Control`	niente: il tema è cambiato (per chi disegna in `Render` , capitolo 11)
`RAva.Every(intervallo, metodo)`	—	niente: è passato l'intervallo; restituisce il `DispatcherTimer` già avviato
`OnClick(metodo)`	`MenuItem`	niente: la voce è stata scelta, col mouse o con il suo tasto

Per le voci dei menu c'è anche `HotKey("Ctrl+S")` , che scrive la scorciatoia a destra della voce (`InputGesture`) e la fa funzionare (`HotKey`), anche a menu chiuso. Le voci vanno messe con `Items.Add` : con `ItemsSource` quelle di un sottomenu chiuso non esistono ancora e il tasto non scatta.

Colori e temi

I due dizionari del capitolo 7 si possono scrivere a coppie, una riga per colore, con `RThemes` :

```
/// <summary>Tema Fluent e colori dei due temi.</summary>
public override void Initialize() {
  Styles.Add(new FluentTheme());
  new RThemes()
    .Color("Sfondo", light: "#ffffff", dark: "#1e1f22")
    .Color("Pannello", light: "#f3f5f8", dark: "#2a2c30")
    .Color("Testo", light: "#1f2328", dark: "#e6e6e6")
    .Apply(this);
}
```

`Apply(this)` in `Application.Initialize` vale per tutte le finestre; `Apply(finestra)` solo dentro una finestra. I controlli poi si legano alla chiave con `Resource`, come prima.

Una rubrica che si filtra

Una finestra che le usa: si scrive nella casella e l'elenco si filtra a ogni lettera, la spunta mostra solo i clienti attivi, Invio sceglie il primo nome, e la scelta riempie la scheda a destra. Ogni evento arriva a un metodo con un nome della classe `Rubrica`.

```
/// <summary>Una rubrica che si filtra mentre si scrive: pannelli pronti,
testo, cornici ed eventi con un nome.</summary>
public static Control RubricaRAva() {
  var r = new Rubrica();
  var cerca = new TextBox { PlaceholderText = "Cerca un nome…"
}.OnTextChanged(r.Filtra).OnEnter(r.ScegliPrimo);
  var soloAttivi = new CheckBox().Content("Solo i clienti
attivi").OnChecked(r.SoloAttivi);
  r.Elenco.OnSelectionChanged(r.Mostra);
  // un pannello non ha imbottitura ne' bordo: la cornice e' un Border
attorno
  var scheda = new Border().Padding(16).Border(1, radius: 8)
    .Resource(Border.BackgroundProperty,
"Sfondo").Resource(Border.BorderBrushProperty, "Bordo")
    .Child(RAva.VStack(6,
      new
TextBlock().Text("Scheda").FontSize(12).Bold().Resource(TextBlock.Foreground
Property, "TestoTenue"),
      r.Nome.FontSize(20).FontWeight(FontWeight.SemiBold),
      r.Stato.FontSize(13).Wrap()));
  var g = RAva.Grid("220,*", "Auto,Auto,*").Spacing(14, 10).Children(
    cerca.Row(0),
    soloAttivi.Row(1),
    r.Elenco.Row(2),
    scheda.Row(0, span: 3).Column(1));
  r.Elenco.SelectedIndex = 1;
// per la figura: una scelta gia' fatta
  return new Border().Padding(16).Resource(Border.BackgroundProperty,
"Pannello").Child(g);
}

// la rubrica dell'esempio: i dati, i controlli da aggiornare e i metodi
chiamati dagli eventi
private sealed class Rubrica {
  private static readonly string[] Tutti = { "Anna Bianchi", "Bruno Rossi",
"Carla Verdi", "Dario Neri", "Elena Gallo" };
  private static readonly string[] Attivi = { "Anna Bianchi", "Bruno Rossi",
"Elena Gallo" };
  private string _filtro = "";
// il testo cercato
  private bool _soloAttivi;
// solo i clienti attivi

  /// <summary>L'elenco dei nomi.</summary>
  public ListBox Elenco { get; } = new ListBox();
```

```
  /// <summary>Il nome scelto, grande, nella scheda.</summary>
  public TextBlock Nome { get; } = new TextBlock().Text("—");
  /// <summary>Lo stato del cliente scelto.</summary>
  public TextBlock Stato { get; } = new TextBlock();

  // all'inizio l'elenco completo
  public Rubrica() { Aggiorna(); }

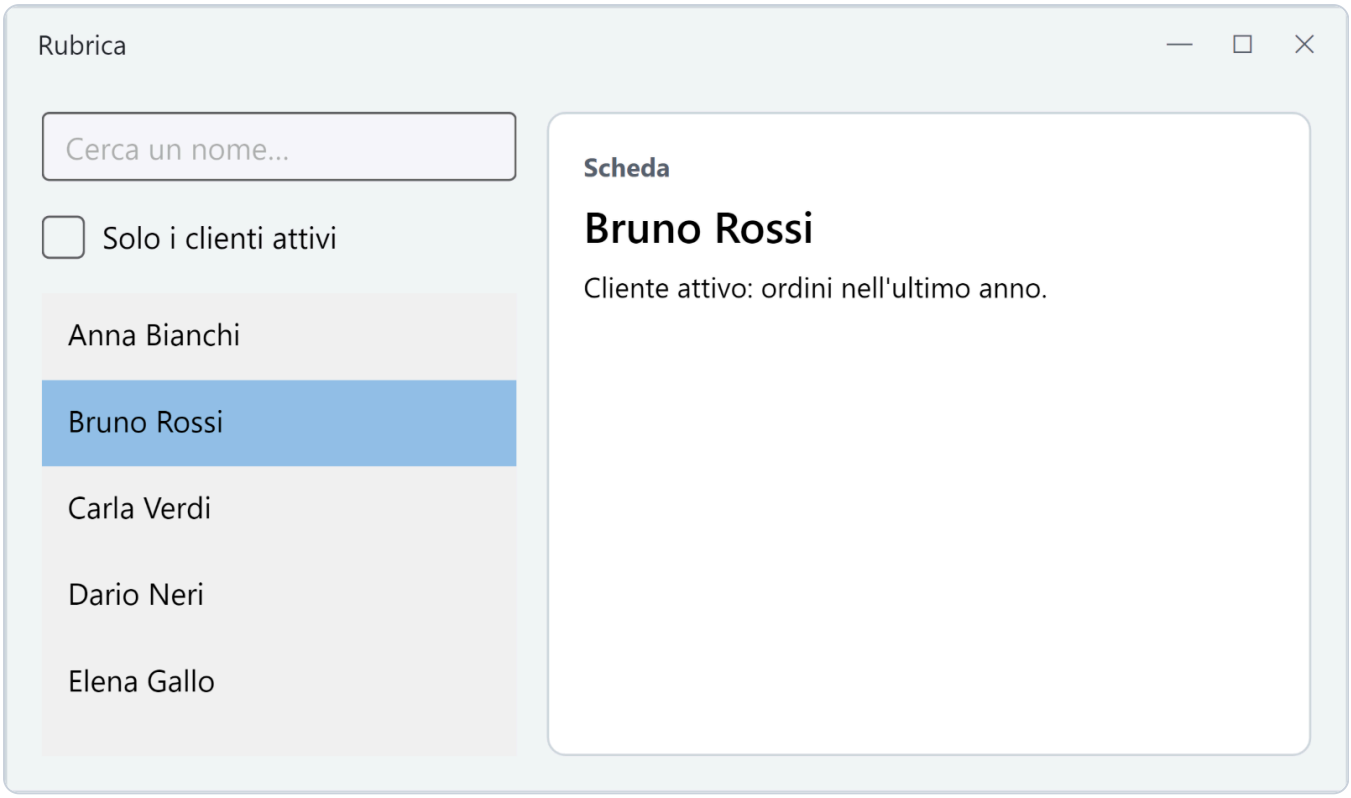
  // AlCambioTesto: il filtro cambia a ogni lettera
  public void Filtra(string testo) { _filtro = testo.Trim(); Aggiorna(); }

  // AlSpunta: mostra tutti o solo gli attivi
  public void SoloAttivi(bool spuntato) { _soloAttivi = spuntato;
Aggiorna(); }

  // OnEnter: sceglie il primo nome dell'elenco filtrato
  public void ScegliPrimo() { if (Elenco.ItemCount > 0) Elenco.SelectedIndex
= 0; }

  // AlCambioSelezione: la scheda del nome scelto
  public void Mostra(object? scelto) {
    string? nome = scelto as string;
    Nome.Text = nome ?? "—";
    Stato.Text = nome == null ? "" : Array.IndexOf(Attivi, nome) >= 0 ?
"Cliente attivo: ordini nell'ultimo anno." : "Nessun ordine nell'ultimo
anno.";
  }

  // l'elenco secondo il filtro e la spunta (foreach, non LINQ con lambda)
  private void Aggiorna() {
    var nomi = new List<string>();
    foreach (string n in Tutti) {
      if (_soloAttivi && Array.IndexOf(Attivi, n) < 0) continue;
      if (_filtro.Length > 0 && !n.Contains(_filtro,
StringComparison.OrdinalIgnoreCase)) continue;
      nomi.Add(n);
    }
    Elenco.ItemsSource = nomi;
  }
}
```



Notate la cornice della scheda: le righe stanno in un `VStack`, ma imbottitura e bordo li ha il `Border` attorno. Scrivere `RAva.VStack(...).Padding(16)` si ferma con l'errore «Imbottitura: StackPanel non ha questa proprietà»: un pannello non ha imbottitura.

Perché si chiamano come le proprietà

L'estensione che imposta `Margin` si chiama `Margin`, quella che imposta `Background` si chiama `Background`: chi conosce Avalonia non deve imparare nomi nuovi. `controllo.Margin` (senza parentesi) è la proprietà, `controllo.Margin(8)` è l'estensione. Funziona perché, quando un nome è **chiamato** con le parentesi, il compilatore scarta i membri che non si possono chiamare, come una proprietà, e va a cercare i metodi d'estensione. Misurato con Avalonia 12.1.4: `new Button().Margin(8)` imposta il margine a 8 su tutti i lati.

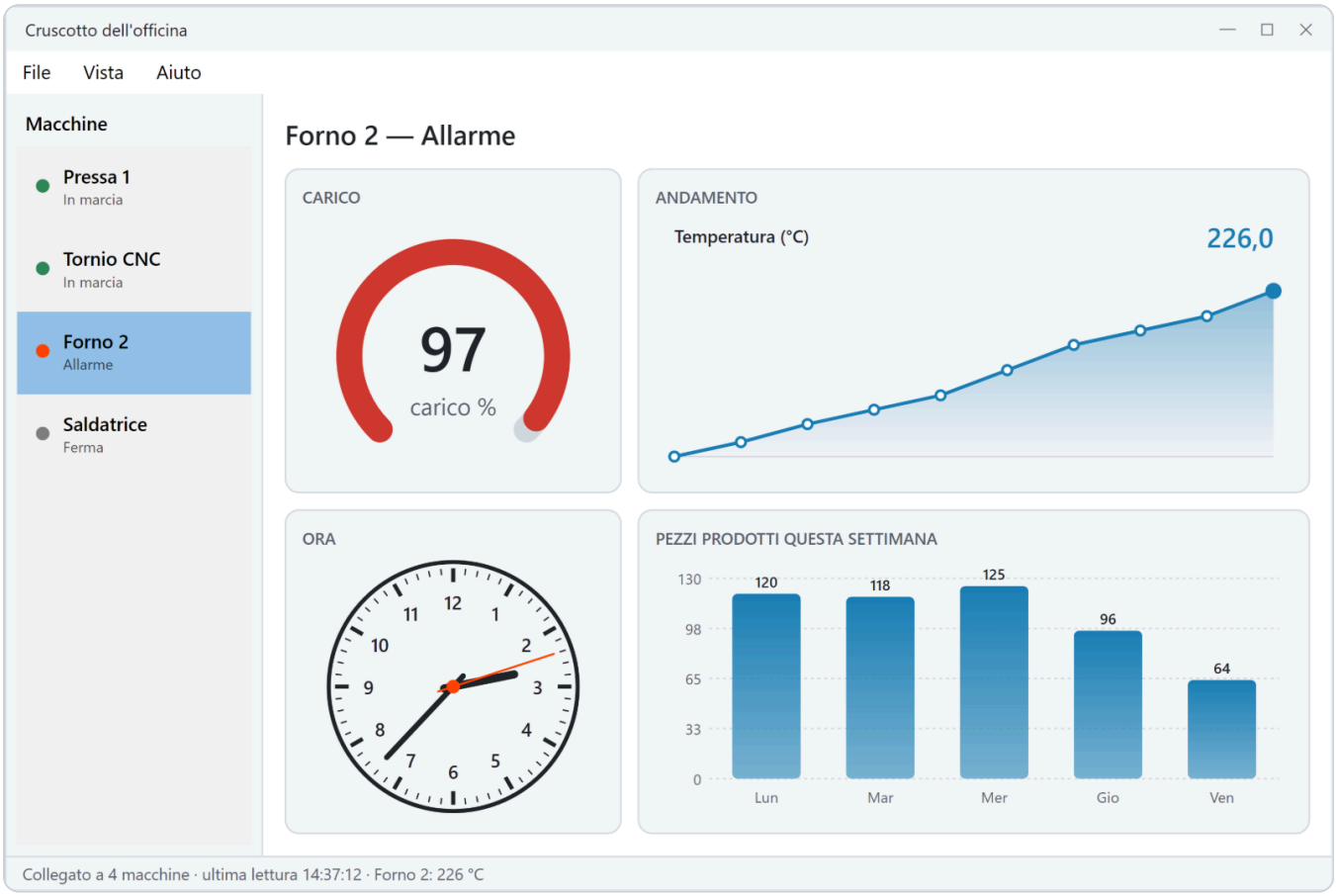
Le eccezioni sono poche e hanno un motivo:

- `AutomationName`, non `Name`: in Avalonia `Name` è il nome del controllo nell'albero, un'altra cosa;
- `OnClick`, `OnTextChanged`...: gli eventi non sono proprietà da impostare, e `On` dice che si collega un metodo;
- `RAva.HStack` e `RAva.VStack`: `Row` e `Column` sono già le estensioni della `Grid`.

Per lo stesso motivo **non serve** `Avalonia.Markup.Declarative` o un'altra libreria «fluente»: queste poche estensioni bastano, e restano codice nostro.

13. Il progetto finale: un cruscotto

> **Cosa imparerai:** a mettere insieme tutto: cornice con DockPanel, menu, elenco con un modello, schede in una Grid, i controlli disegnati del capitolo 11, i colori del tema e la selezione che aggiorna le schede.



13.1 I dati

```
/// <summary>Una macchina dell'officina: nome, stato e le sue misure.
</summary>
public sealed record Macchina(string Nome, string Stato, double Carico,
double[] Temperature, double[] Pezzi);
```

13.2 La riga dell’elenco

```
/// <summary>Un pallino colorato secondo lo stato, il nome e lo stato
scritto.</summary>
public sealed class ModelloMacchina : IDataTemplate {

  /// <summary>Il modello vale per le Macchina.</summary>
  public bool Match(object? dati) => dati is Macchina;

  /// <summary>I controlli di una riga.</summary>
  public Control? Build(object? dati) {
    if (dati is not Macchina m) return null;
    IBrush colore = m.Stato == "In marcia" ? Brushes.SeaGreen : m.Stato ==
"Ferma" ? Brushes.Gray : Brushes.OrangeRed;
    return new StackPanel { Orientation = Orientation.Horizontal, Spacing =
10 }.Margin(2, 3).Children(
      new Border { Width = 10, Height = 10, CornerRadius = new
CornerRadius(5), Background = colore }.VAlign(VerticalAlignment.Center),
      new StackPanel().Children(
        new TextBlock { FontWeight = FontWeight.SemiBold }.Text(m.Nome),
        new TextBlock { FontSize = 11, Opacity = 0.7 }.Text(m.Stato)));
  }
}
```

13.3 La finestra

I controlli che cambiano sono **campi** della classe: il gestore della selezione li ritrova lì.

```
public sealed class Cruscotto {

  private readonly ListBox _elenco = new() { ItemTemplate = new
ModelloMacchina() };
  private readonly Indicatore _carico = new() { Unita = "carico %" };
  private readonly GraficoLinee _temperatura = new() { Titolo = "Temperatura
(°C)" };
  private readonly GraficoBarre _pezzi = new() { Etichette = new[] { "Lun",
"Mar", "Mer", "Gio", "Ven" } };
  private readonly TextBlock _titolo = new() { FontSize = 20, FontWeight =
FontWeight.SemiBold };
  private readonly TextBlock _stato = new() { FontSize = 12 };

  /// <summary>La finestra completa.</summary>
  public Control Vista { get; }

  /// <summary>Costruisce la cornice e le schede, poi seleziona la prima
macchina.</summary>
  public Cruscotto() {
    var macchine = new List<Macchina> { /* quattro macchine con le loro
misure */ };
    _elenco.ItemsSource = macchine;
    _elenco.SelectionChanged += MacchinaScelta;

    var menu = new Menu().Dock(Dock.Top);
    menu.Items.Add(new MenuItem { Header = "_File" });
    menu.Items.Add(new MenuItem { Header = "_Vista" });
    menu.Items.Add(new MenuItem { Header = "_Aiuto" });

    var schede = new Grid { ColumnDefinitions = new
ColumnDefinitions("*,2*"), RowDefinitions = new RowDefinitions("*,*"),
ColumnSpacing = 12, RowSpacing = 12 }.Children(
      Scheda("Carico", _carico).Row(0).Column(0),
      Scheda("Andamento", _temperatura).Row(0).Column(1),
      Scheda("Ora", new Orologio { OraFissa = new TimeSpan(14, 37, 12)
}).Row(1).Column(0),
      Scheda("Pezzi prodotti questa settimana", _pezzi).Row(1).Column(1));

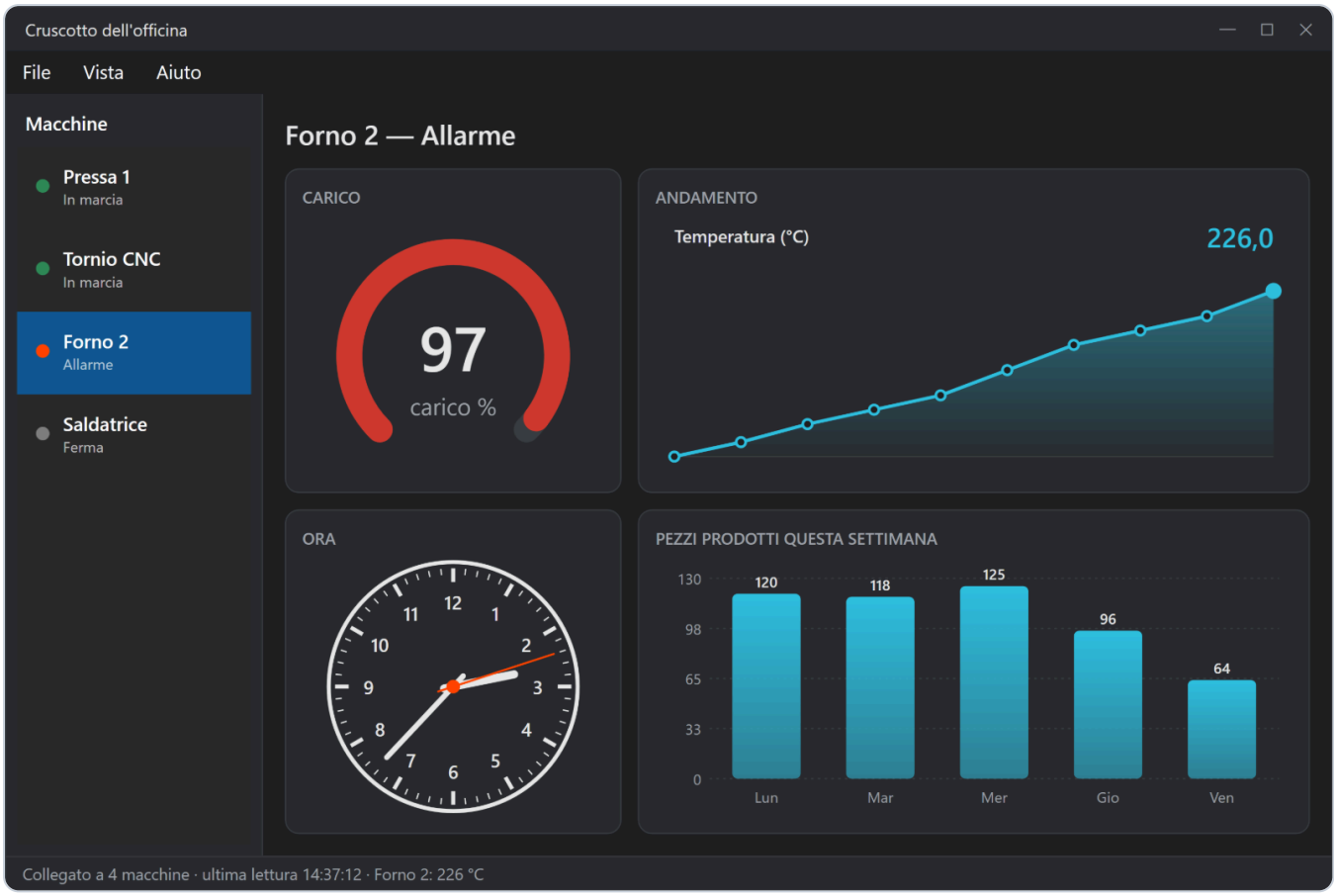
    Vista = new DockPanel().Children(
      menu,
      new Border { BorderThickness = new Thickness(0, 1, 0, 0), Padding =
new Thickness(12, 4) }
        .Resource(Border.BorderBrushProperty,
"Bordo").Resource(Border.BackgroundProperty, "Pannello").Dock(Dock.Bottom)
        .Child(_stato.Resource(TextBlock.ForegroundProperty, "TestoTenue")),
```

```
      new Border { BorderThickness = new Thickness(0, 0, 1, 0), Width = 190
}
        .Resource(Border.BorderBrushProperty,
"Bordo").Resource(Border.BackgroundProperty, "Pannello").Dock(Dock.Left)
        .Child(new DockPanel().Margin(8).Children(
          new TextBlock { FontWeight = FontWeight.SemiBold
}.Text("Macchine").Margin(6, 4, 0, 8).Dock(Dock.Top), _elenco)),
      new DockPanel().Margin(16).Children(_titolo.Margin(0, 0, 0,
12).Dock(Dock.Top).Resource(TextBlock.ForegroundProperty, "Testo"),
schede));
    _elenco.SelectedIndex = 2;
  }

  // ListBox.SelectionChanged: le schede mostrano la macchina scelta
  private void MacchinaScelta(object? mittente, SelectionChangedEventArgs e)
{
    if (_elenco.SelectedItem is not Macchina m) return;
    _titolo.Text = m.Nome + " — " + m.Stato;
    _carico.Valore = m.Carico;
    _temperatura.Valori = m.Temperature;
    _pezzi.Valori = m.Pezzi;
    _stato.Text = $"Collegato a 4 macchine · ultima lettura 14:37:12 ·
{m.Nome}: {m.Temperature[^1]:0} °C";
  }

  // una scheda: bordo arrotondato col colore del pannello, un titolo e il
contenuto
  private static Border Scheda(string titolo, Control contenuto) =>
    new Border { CornerRadius = new CornerRadius(10), BorderThickness = new
Thickness(1), Padding = new Thickness(12) }
      .Resource(Border.BackgroundProperty,
"Pannello").Resource(Border.BorderBrushProperty, "Bordo")
      .Child(new DockPanel().Children(
        new TextBlock { FontSize = 12, FontWeight = FontWeight.SemiBold
}.Text(titolo.ToUpperInvariant())
          .Resource(TextBlock.ForegroundProperty, "TestoTenue").Margin(0, 0,
0, 6).Dock(Dock.Top),
        contenuto));
}
```

Lo stesso cruscotto nel tema scuro, senza una riga di codice in più:



Ripercorriamo cosa c'è dentro: la cornice è un `DockPanel` (capitolo 4) con menu (9), barra di stato ed elenco; l'elenco usa un `IDataTemplate` (8); le schede sono `Border` in una `Grid` con colonne `*` e `2*` (4); i colori sono tutti risorse del tema (7); i grafici sono i controlli disegnati (11), e i loro setter chiamano `InvalidateVisual`, così `MacchinaScelta` deve solo assegnare i dati nuovi (6). Per i dati veri, letti da una macchina in un altro thread, si aggiunge il Dispatcher (capitoli 2 e 10).

Esercizi

1. Aggiungete il menu Vista → Tema scuro, che chiama `Temi.CambiaTema`.
2. Con un `DispatcherTimer` aggiungete ogni secondo una temperatura finta alla macchina scelta, togliendo la più vecchia: il grafico deve scorrere.
3. Aggiungete una quinta scheda con la `Torta` dei pezzi buoni e degli scarti.

14. Un server di API JSON

> **Cosa imparerai:** a far rispondere il tuo programma alla rete con `RServer`: percorsi collegati a metodi con un nome, oggetti che diventano JSON, codici di errore, e i tre modi di avviarlo

> (senza `await`, con un thread, con `await`). Chi lo chiama può essere un browser, un telefono, un altro programma.

Un programma con finestre spesso deve anche **farsi chiamare**: il cruscotto del capitolo 13 letto da un telefono, una macchina che manda le sue misure, un altro programma che chiede un dato. Il modo più semplice oggi è un'**API JSON** su HTTP: l'altro manda una richiesta a un indirizzo (`GET /api/macchine/3`), il programma risponde con un testo JSON (`{ "id": 3, "nome": "Pressa" }`).

`RServer` di RAVAL lo fa sopra **Kestrel**, il server web di ASP.NET Core (gratuito, MIT, già nell'SDK di .NET), senza il resto del framework: niente controller, niente attributi, niente lambda. Ogni percorso è collegato a un **metodo con un nome**, che riceve la richiesta e restituisce l'oggetto da mandare.

14.1 La bacheca

Una finestra con un elenco di messaggi. Chiunque nella rete può leggerli (`GET /api/messaggi`) e aggiungerne uno (`POST /api/messaggi` con `{ "autore": "...", "testo": "..." }`).

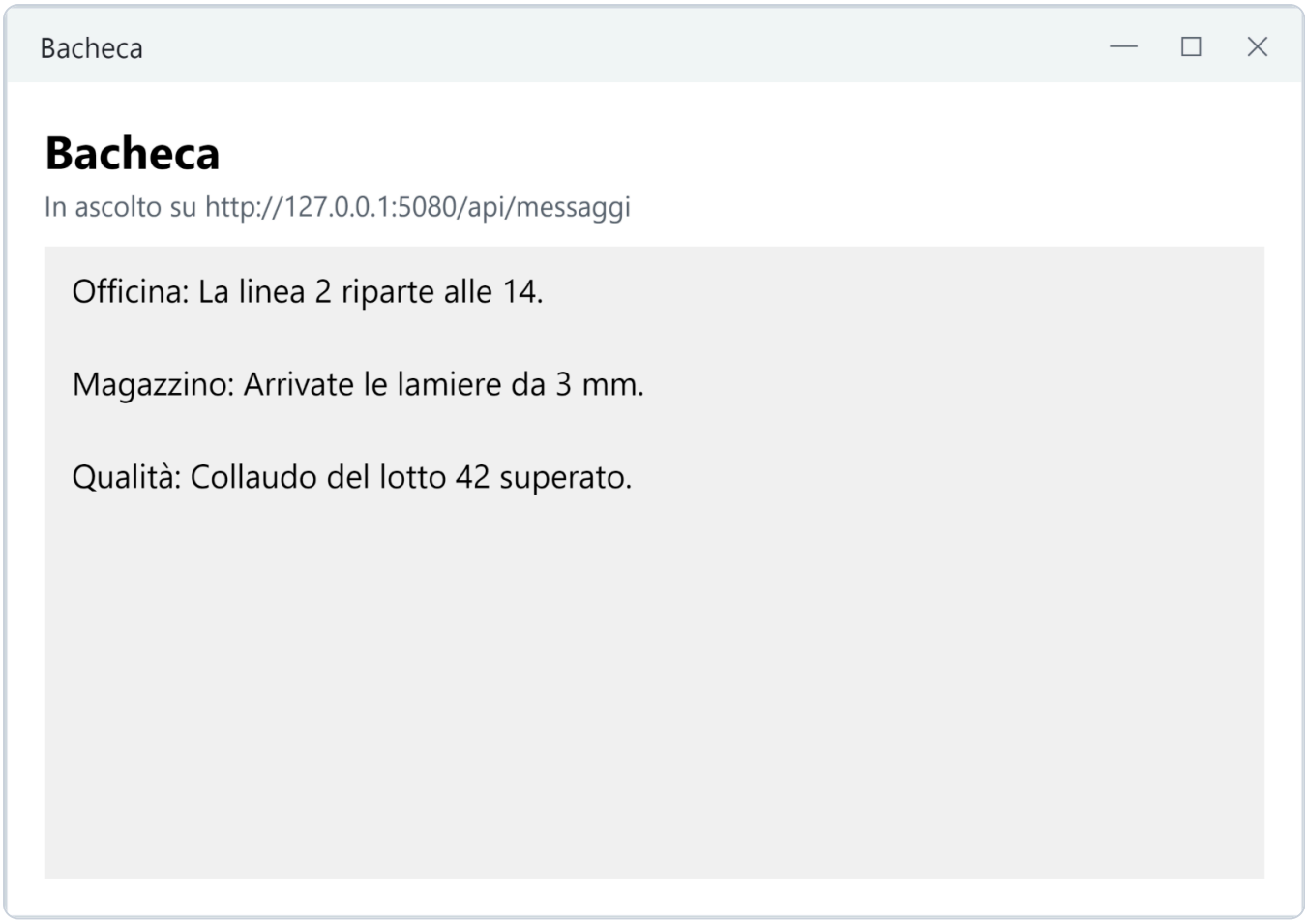
```
/// <summary>Un messaggio della bacheca: diventa e viene dal JSON {
"autore": ..., "testo": ... }.</summary>
public sealed class MessaggioBacheca {
  /// <summary>Chi scrive.</summary>
  public string Autore { get; set; } = "";
  /// <summary>Il messaggio.</summary>
  public string Testo { get; set; } = "";
  /// <summary>Come compare nell'elenco.</summary>
  public override string ToString() => Autore + ": " + Testo;
}
```

```
/// <summary>La bacheca con il suo server sulla porta data (0 = una porta
libera).</summary>
public Bacheca(int porta) {
  _server = new RServer(porta)
    .Get("/api/messaggi", Elenco)
    .Post("/api/messaggi", Aggiungi)
    .OnUiThread();
// Aggiungi tocca l'elenco
  _server.Start();
// niente await: torna quando ascolta
  _stato.Text = "In ascolto su " + _server.Addresses[0] + "/api/messaggi";
  var titolo = new TextBlock().Text("Bacheca").FontSize(20).Bold();
  var elenco = new ListBox { ItemsSource = _messaggi };
  Vista = new DockPanel().Margin(16).Children(
    RAva.VStack(2, titolo, _stato).Margin(0, 0, 0, 10).Dock(Dock.Top),
    elenco);
  Vista.DetachedFromVisualTree += Chiusa;
}

// GET /api/messaggi: l'elenco attuale
private object? Elenco(RRequest r) => _messaggi;

// POST /api/messaggi: un messaggio nuovo, letto dal JSON del corpo
private object? Aggiungi(RRequest r) {
  MessaggioBacheca m = r.Body<MessaggioBacheca>();
  if (m.Testo.Trim().Length == 0) throw new RHttpError(400, "il testo e'
vuoto");
  _messaggi.Add(m);
// siamo sul thread dell'interfaccia
  return RResponse.Created(m);
}

// la bacheca e' uscita dalla finestra: il server si ferma e libera la porta
private void Chiusa(object? mittente, VisualTreeAttachmentEventArgs e) =>
_server.Stop();
```

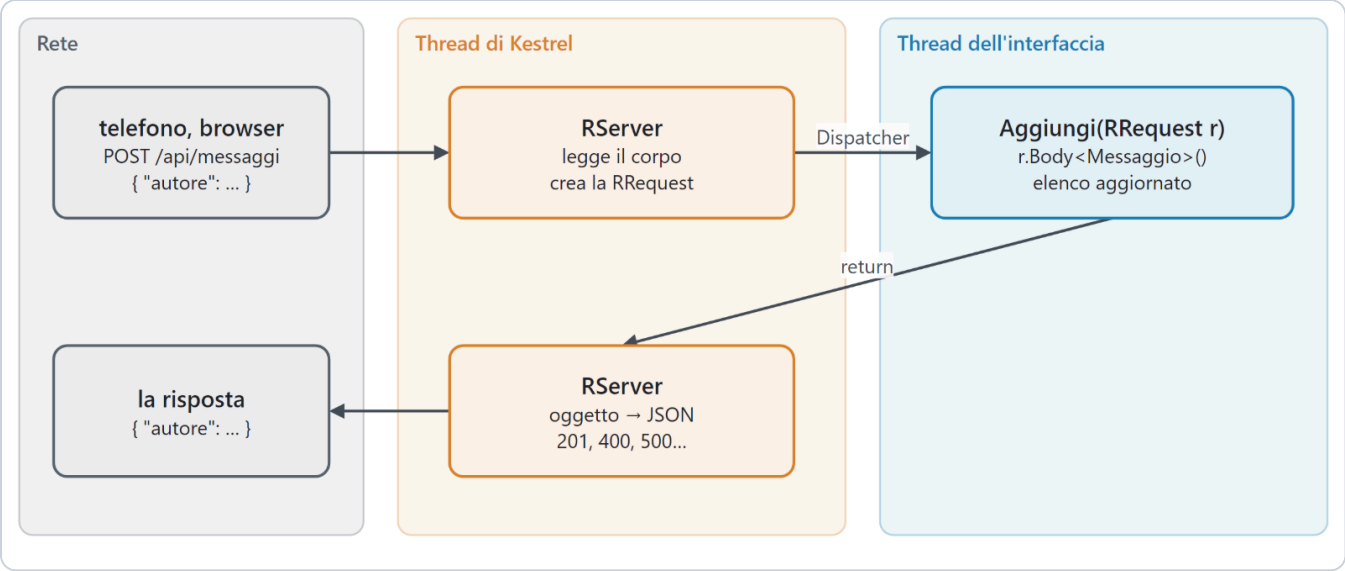


I tre messaggi della figura sono arrivati davvero via HTTP: il programma che disegna le figure li manda con un `HttpClient` da un altro thread, mentre la finestra è aperta. Per provarlo a mano, con il programma avviato su una porta fissa (per esempio `new Bacheca(5080)`), da un terminale:

```
curl http://127.0.0.1:5080/api/messaggi
curl -X POST http://127.0.0.1:5080/api/messaggi -H "Content-Type:
application/json" -d "{\"autore\":\"Anna\",\"testo\":\"ciao\"}"
```

Oppure si apre `http://127.0.0.1:5080/api/messaggi` nel browser: i GET si leggono così.

14.2 Il viaggio di una richiesta



1. Kestrel riceve la richiesta su un **suo** thread, legge il corpo e cerca il percorso;
2. `RServer` costruisce la `RRequest` e chiama il tuo metodo. Con `OnUiThread()` lo chiama sul thread dell'interfaccia attraverso il Dispatcher (capitolo 2, §2.8), quindi il metodo può toccare i controlli; senza, lo chiama sul thread di Kestrel;
3. l'oggetto restituito diventa JSON e torna indietro.

Le regole della risposta:

Il metodo…	La risposta è
restituisce un oggetto	200 con l'oggetto in JSON (nomi in camelCase: `Autore` diventa `"autore"`)
restituisce `null`	204, senza corpo
restituisce `RResponse.Created(x)`, `new RResponse(202, x)`	quel codice, con `x` in JSON
lancia `new RHttpError(404, "non c'è")`	404 con `{ "error": "non c'è" }`
lancia un'eccezione qualsiasi	500 con `{ "error": messaggio }`: il server non cade

E quello che il metodo trova nella `RRequest`:

Membro	Cosa dà
`Param("id")`	un parametro del percorso (`/api/macchine/{id}`) o dopo il `?` (`?filtro=rossi`); null se manca
`Int("id")`	lo stesso, come intero; se non è un numero, la risposta è già un 400
`Body<T>()`	il corpo JSON letto come oggetto; se manca o è sbagliato, la risposta è già un 400
`Text`	il corpo come testo
`Header("Authorization")`	un'intestazione della richiesta
`Metodo` , `Percorso` , `Contesto`	GET/POST..., il percorso, l' `HttpContext` di ASP.NET per il resto

Un percorso con un parametro:

```
// GET /api/macchine/{id}: 404 se non c'e'
public object? Macchina(RRequest r) {
  int id = r.Int("id");
  if (id < 1 || id >= Macchine.Length) throw new RHttpError(404, "la
macchina " + id + " non c'e'");
  return new Scheda { Id = id, Nome = Macchine[id] };
}
```

14.3 Tre modi di avviarlo

Come per tutto quello che aspetta (capitolo 2), ci sono tre modi, e vanno bene tutti:

```
// 1. come un metodo normale: Avvia torna quando il server ascolta
server.Start();
...
server.Stop();

// 2. con un thread: Esegui avvia il server e resta fermo finche' qualcuno
chiama Ferma
var filo = new Thread(server.Run) { IsBackground = true, Name = "server" };
filo.Start();
server.WaitStarted(TimeSpan.FromSeconds(10));
...
server.Stop();
// libera il thread di Esegui
filo.Join();

// 3. con await
await server.StartAsync();
...
await server.StopAsync();
```

Il modo 2 è quello di un programma console che fa solo da server: `server.Run();` alla fine di `Main`, e il programma vive finché qualcuno (un comando, un altro percorso dell'API) chiama `Stop`. `Start` e `Stop` si possono chiamare dal thread dell'interfaccia senza rischi: dentro fanno il lavoro su un thread del pool, per il motivo spiegato in §2.9.

Anche i metodi dei percorsi possono essere asincroni, se devono aspettare qualcosa (un archivio, un altro servizio): `GetAsync` e `PostAsync` accettano un `async Task<object?> Metodo(RRequest r)`.

14.4 Con i thread invece di SulThreadInterfaccia

Senza `OnUiThread()` i metodi girano sui thread di Kestrel: sono più veloci (non aspettano il thread dell'interfaccia) ma non possono toccare i controlli. È la scelta giusta quando l'API legge dati che non stanno nei controlli, e si usano le regole del capitolo 2: i dati condivisi con la finestra si proteggono con `lock`, e per aggiornare lo schermo si chiede al Dispatcher.

```
private readonly object _chiave = new();
// protegge _ultimaMisura
private double _ultimaMisura;

// POST /api/misura, sul thread di Kestrel
private object? NuovaMisura(RRequest r) {
  Misura m = r.Body<Misura>();
  lock (_chiave) _ultimaMisura = m.Valore;
  Dispatcher.UIThread.Post(Mostra);
// lo schermo si aggiorna dal suo thread
  return null;
// 204
}

// sul thread dell'interfaccia
private void Mostra() {
  double v;
  lock (_chiave) v = _ultimaMisura;
  valore.Text = v.ToString("0.00");
}
```

14.5 Dalla rete

`new RServer(5080)` ascolta solo su questo computer (`127.0.0.1`): è il default, ed è il più sicuro. Per farsi raggiungere da un telefono o da un altro computer si aggiunge `.OnAllInterfaces()`, e la prima volta il sistema operativo chiede il permesso nel firewall. A quel punto **chiunque nella rete** può chiamare l'API: un'API che cambia qualcosa (`POST`, `DELETE`) va protetta, per esempio controllando un'intestazione con una chiave (`r.Header("X-Chiave")`) e rispondendo 401 se manca.

Una nota per chi distribuisce il programma: `RServer` usa il framework condiviso di ASP.NET Core, che c'è nell'SDK di .NET e nel «.NET Runtime» completo per i server. Un programma che usa RAVAL **senza** `RServer` non lo carica mai.

Esercizi

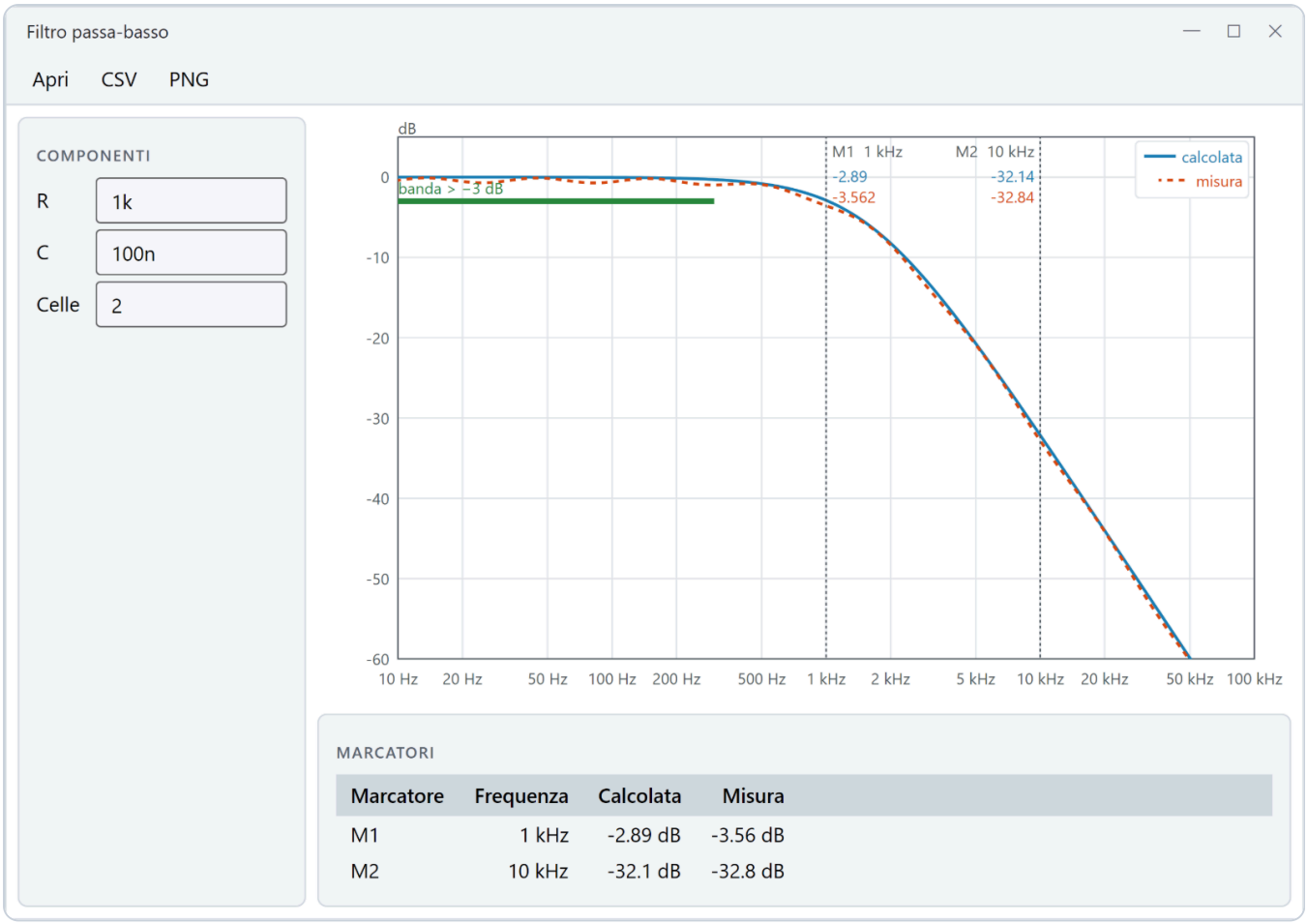
1. Aggiungete alla bacheca `DELETE /api/messaggi/{n}`, che cancella il messaggio numero `n` e risponde 404 se non c'è.
2. Aggiungete `GET /api/messaggi?autore=Anna`, che restituisce solo i messaggi di un autore (`r.Param("autore")`, un ciclo `foreach`, nessuna lambda).
3. Rifate la bacheca senza `OnUiThread()`: i messaggi in una `List` protetta da `lock`, e l'elenco aggiornato con `Dispatcher.UIThread.Post`.

4. Scrivete un programma console che fa solo da server con `server.Run()`, e un percorso `POST /api/ferma` che chiama `Stop` da un altro thread (attenzione: chi chiama `Stop` non deve essere il thread che sta rispondendo; usate un `new Thread(server.Ferma).Start()`).

15. Strumenti di misura: grafici, tabelle e stili pronti

> **Cosa imparerai:** a costruire la finestra di uno strumento con i pezzi pronti di RAVAL: il grafico a curve `RChart` con i marcatori, la tabella `RTable`, l'elenco di valori `RPropertyList`, i numeri con i prefissi SI di `RFormat`, le curve in CSV con `RCsv`, il grafico salvato a 300 dpi con `RImage` e l'aspetto pulito di `RStyles`.

Nel capitolo 11 abbiamo disegnato i grafici a mano, punto per punto, per capire come si fa. Un programma di misura o di progetto però ha sempre bisogno delle stesse cose: un grafico con gli assi tondi e la legenda, i marcatori da spostare con il mouse, una tabella che si ordina, i valori dei componenti da cambiare, i numeri scritti come li legge un tecnico («1 kHz», non «1000»), le curve da salvare per un foglio di calcolo e le figure da stampare. RAVAL le ha già: in questo capitolo le mettiamo insieme nella finestra che misura un filtro passa-basso.



Il codice è in `ESEMPI_RAVAL\Esempi\Strumento.cs`. Le classi sono tutte nel namespace `RAVAL`, come le estensioni di RAva; i loro membri, uno per uno, sono nel riferimento (`DOC\API.md`).

15.1 I numeri come li legge un tecnico: RFormat

Un asse che dice «1000000000» non si legge, uno che dice «1 GHz» sì. `RFormat.Si` sceglie il prefisso che lascia da 1 a 999 prima del punto e tiene al più quattro cifre significative; `RFormat.Parse` fa il contrario, e accetta anche il modo in cui si scrivono i valori sugli schemi, con il prefisso al posto della virgola:

```
RFormat.Si(1.296e9, "Hz")        // «1.296 GHz»
RFormat.Si(2.2e-8, "F", 2)       // «22 nF»: due cifre significative
RFormat.Parse("4k7")             // 4700
RFormat.Parse("10 MHz", "Hz")    // 10 000 000: l'unità dopo il prefisso si salta
RFormat.Parse("5%")              // 0,05
```

I numeri scritti da `RFormat` hanno sempre il punto decimale, qualunque sia la lingua del computer: un valore copiato da una tabella si rilegge uguale. Il grafico, la tabella e l'elenco di proprietà usano `RFormat` da soli: la tabella, per esempio, ordina «470p» prima di «10n» perché ne legge il valore.

15.2 Il grafico: RChart, RSeries, RLimit

`RChart` è un controllo: si mette in un pannello come un `TextBlock`. Gli si danno le curve (`RSeries`, un nome e due array di ascisse e ordinate), i limiti (`RLimit`, un segmento orizzontale spesso, per esempio la banda che il filtro deve rispettare) e le ascisse dei marcatori:

```
private readonly RChart _grafico = new() { TitleY = "dB", UnitX = "Hz", LogX
= true, MinY = -60, MaxY = 5 };

_grafico.Series.Add(new RSeries("calcolata", _f, calcolata));
_grafico.Series.Add(new RSeries("misura", _f, misura, 1) { Dashed = true });
// il secondo colore, tratteggiata
_grafico.Limits.Add(new RLimit(10, 300, -3, RAva.Brush("#2b8a3e")) { Label =
"banda > -3 dB" });
_grafico.Markers.Add(1000);
_grafico.MarkersChanged += Marcatori;
```

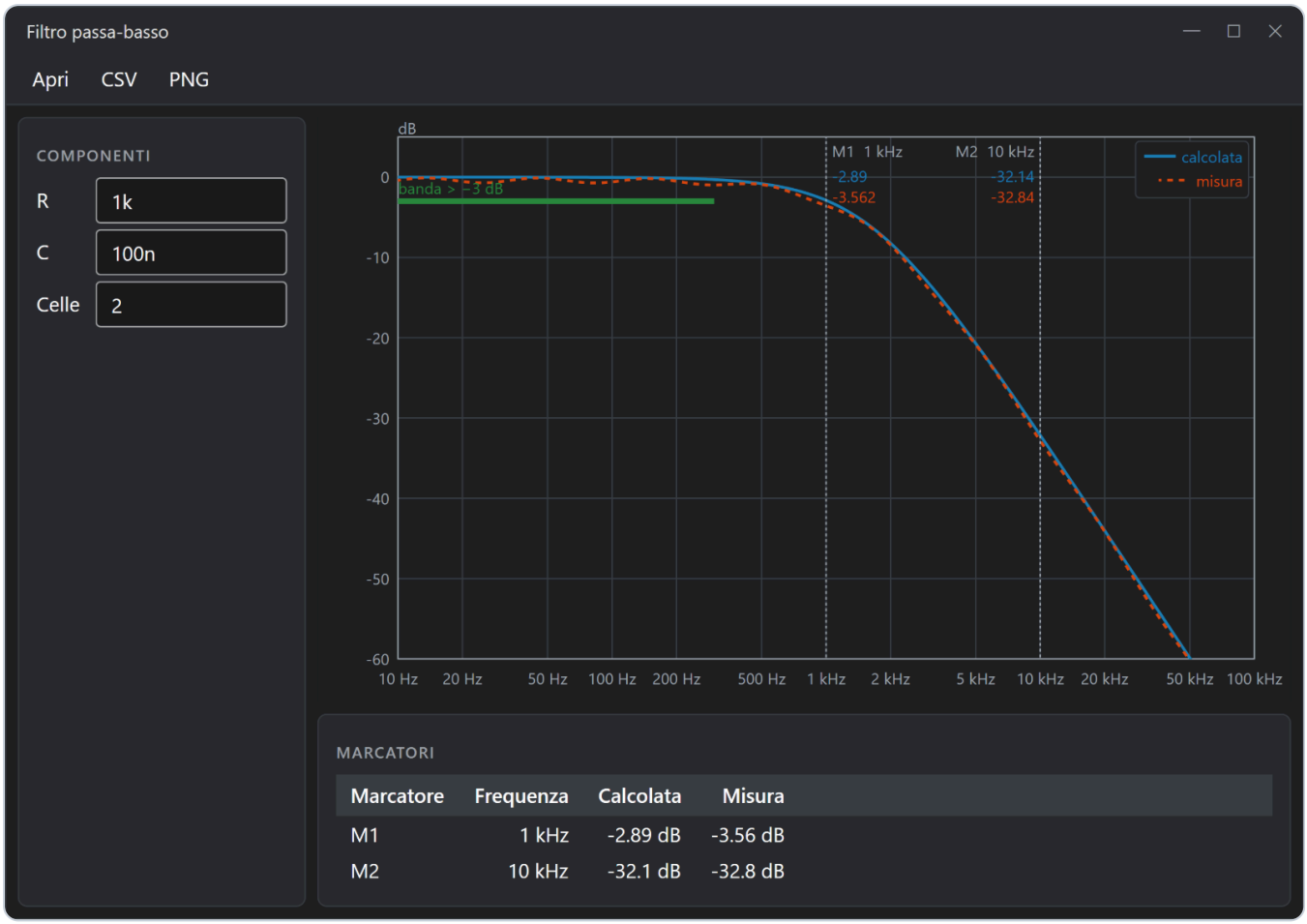
Le divisioni degli assi sono «tonde» (1, 2 o 5 per una potenza di dieci) e, con `LogX`, l'ascissa è logaritmica con le divisioni 1-2-5 di ogni decade. Le ascisse si scrivono con `RFormat.Si` e l'unità `UnitX`; se servono scritte diverse (gradi, date), `FormatX` prende un metodo con un nome che riceve il numero e restituisce il testo. `MinY` e `MaxY` fissano l'asse sinistro; lasciati vuoti, lo decidono i dati. Una curva con `RightAxis = true` usa l'asse destro, che compare da solo con il titolo `TitleY2`; una curva con il nome vuoto si disegna ma resta fuori dalla legenda: serve quando le curve sono cento, come le prove di un'analisi statistica.

Il mouse lavora da solo: **un clic** mette il primo marcatore, **Maiusc + clic** ne aggiunge un altro, **il tasto destro** li toglie tutti. Ogni marcatore scrive in alto il valore di ogni curva, interpolato fra i due campioni più vicini (`RSeries.At`). Dopo ogni clic arriva l'evento `MarkersChanged`, un `Action` senza argomenti: il nostro metodo `Marcatori` rifà la tabella.

Quando le curve cambiano, si rifà la lista e si chiama `Refresh`:

```
_grafico.Series.Clear();
_grafico.Series.Add(new RSeries("calcolata", _f, calcolata));
_grafico.Series.Add(new RSeries("misura", _f, misura, 1) { Dashed = true });
_grafico.Refresh();
```

I colori delle curve vengono dalla tavolozza `RColor.Palette`, otto colori che si leggono sia sul bianco sia sul quasi nero; lo sfondo, la griglia e le scritte vengono dalle risorse del tema «ChartBack», «ChartGrid» e «ChartInk». Un'applicazione che non le definisce, come la nostra, vede comunque un grafico chiaro nel tema chiaro e scuro nel tema scuro: ogni risorsa ha il suo colore di riserva.



15.3 La tabella: RTable

`RTable` mostra righe di testo, si ordina e si sceglie. Le colonne si danno con le intestazioni; «:R» in fondo allinea a destra, «:C» al centro:

```
private readonly RTable _marcatori = new() { HeaderKey = "Bordo", StripeKey
= "Pannello" };

_marcatori.Columns("Marcatore", "Frequenza:R", "Calcolata:R", "Misura:R");

// la tabella dei marcatori, una riga per marcatore
private void Marcatori() {
  _marcatori.Clear();
  for (int i = 0; i < _grafico.Markers.Count; i++) {
    double f = _grafico.Markers[i];
    _marcatori.AddRow("M" + (i + 1), RFormat.Si(f, "Hz"),
                      RFormat.Si(_grafico.Series[0].At(f), "dB", 3),
                      RFormat.Si(_grafico.Series[1].At(f), "dB", 3));
  }
}
```

Un clic sull'intestazione ordina per quella colonna (per valore, se ogni cella è un numero, anche con un prefisso SI; se no in ordine alfabetico), un secondo clic rovescia l'ordine. Un clic su una riga la sceglie e manda l'evento `SelectionChanged` con il numero della riga. `HeaderKey` e `StripeKey` sono le risorse del tema per lo sfondo dell'intestazione e delle righe alterne: qui usiamo i nomi italiani della nostra applicazione.

`RTable` è una `Grid` di `TextBlock`: è semplice e si disegna bene anche nelle figure, ma costruisce tutte le righe. Va bene per decine o centinaia di righe; per decine di migliaia serve un elenco virtualizzato (capitolo 8).

15.4 I valori da cambiare: RPropertyList

A sinistra, i valori dei componenti: nomi a sinistra, caselle a destra.

```
private readonly RPropertyList _valori = new();

_valori.Add("R", "1k").Add("C", "100n").Add("Celle", "2");
_valori.Changed += ValoreCambiato;

// un valore confermato nell'elenco: si ricalcola tutto
private void ValoreCambiato(string nome, string valore) => Ricalcola();
```

`Changed` non arriva a ogni tasto, che ricalcolerebbe il filtro con un valore scritto a metà: arriva quando si preme Invio o la casella perde il fuoco, e solo se il testo è cambiato davvero. Il metodo riceve il nome e il valore nuovo; `Value("R")` legge un valore in qualunque momento, e `RFormat.Parse` lo trasforma in numero.

15.5 L'aspetto: Card, Caption, Bar, ToolbarButton

La finestra è un `DockPanel`: la barra in alto, i valori a sinistra, i marcatori in basso e il grafico al centro, che prende lo spazio che resta (capitolo 4).

```
var barra = RAva.Bar(RAva.HStack(2,
  RAva.ToolbarButton("Apri", "Apri una misura", Apri),
  RAva.ToolbarButton("CSV", "Salva le curve in CSV", SalvaCsv),
  RAva.ToolbarButton("PNG", "Salva il grafico a 300 dpi", SalvaPng)), true,
"Pannello", "Bordo");
var sinistra = RAva.Card(RAva.VStack(6, RAva.Caption("Componenti",
"TestoTenue"), _valori),
                         "Pannello", "Bordo").Width(200).Margin(8, 8, 0, 8);
var sotto = RAva.Card(RAva.VStack(6, RAva.Caption("Marcatori",
"TestoTenue"), _marcatori),
                      "Pannello", "Bordo").Margin(8, 0, 8, 8);
Vista = new DockPanel().Children(
  barra.Dock(Dock.Top), sinistra.Dock(Dock.Left), sotto.Dock(Dock.Bottom),
_grafico.Margin(8));
```

- `RAva.Card` è un riquadro con gli angoli tondi e il bordo sottile; `RAva.Caption` il titolo di una sezione, piccolo e in maiuscolo; `RAva.Bar` una barra degli strumenti (`true`, con la linea sotto) o di stato (`false`, con la linea sopra); `RAva.ToolbarButton` un bottone con il suggerimento e il metodo da chiamare. `RAva.CodeBox(16)` è una casella di testo a spaziatura fissa alta sedici righe, per un sorgente o dei numeri in colonna.
- I colori vengono dalle risorse del tema: gli ultimi parametri sono i nomi delle chiavi. Senza, valgono i nomi inglesi «Panel», «Border», «Bar», «InkSoft»; noi passiamo quelli della nostra applicazione.

L'aspetto vero lo danno gli stili di `RStyles.Clean()`, che si applicano una volta sola all'applicazione:

```
public override void Initialize() {
  Styles.Add(new FluentTheme { DensityStyle = DensityStyle.Compact });
  RStyles.Clean().Apply(this);
  RStyles.Accent(this, Color.Parse("#1f5fa8"), Color.Parse("#5b9be0"));
}
```

`Clean` rende piatti i bottoni con la classe `toolbar` (quelli di `ToolbarButton`), dà alle schede una scritta di 13 punti invece di 24, mette la spaziatura fissa alle caselle con la classe `code` e compatta alberi ed elenchi. `Accent` cambia il colore d'accento di Fluent nei due temi, con le sei tinte più chiare e più scure che Fluent usa al passaggio del mouse e alla pressione. Le figure di questo libro sono disegnate tutte dalla stessa applicazione, che non applica `Clean` per non cambiare l'aspetto degli altri capitoli: per questo nella figura i bottoni della barra hanno ancora lo sfondo grigio di Fluent.

Gli stili di `RStyles` si scrivono anche da sé, senza la lambda che `new Style(x => ...)` chiederebbe:

```
new RStyles()
  .For<Button>("toolbar").Set(Button.BackgroundProperty,
Brushes.Transparent)
  .For<TextBlock>("caption").Set(TextBlock.FontSizeProperty, 11.0)
  .Apply(this);
```

15.6 Salvare: le curve in CSV, il grafico a 300 dpi

Il bottone «CSV» salva le due curve in un testo che si apre con qualunque foglio di calcolo:

```
// «CSV»: le due curve, sulle frequenze della prima
private void SalvaCsv() {
  using var w = new StreamWriter("filtro.csv");
  w.Write(RCsv.Write("f (Hz)", _grafico.Series));
}
```

La prima colonna sono le ascisse della prima curva; le altre curve, se hanno ascisse diverse, si interpolano su quelle. I numeri sono scritti nel formato che si rilegge identico, bit per bit, e una cella che contiene una virgola o le virgolette va fra virgolette, come vuole lo standard del CSV (RFC 4180). `RCsv.Write` accetta anche colonne di numeri qualunque e una `RTable`; `RCsv.Read` restituisce le righe di un testo CSV.

Il bottone «PNG» salva il grafico per un documento stampato:

```
// «PNG»: un grafico nuovo con le stesse curve, largo 16 cm a 300 dpi
private void SalvaPng() {
  var copia = new RChart { TitleY = "dB", UnitX = "Hz", LogX = true, MinY =
-60, MaxY = 5 };
  copia.Series.AddRange(_grafico.Series);
  copia.Limits.AddRange(_grafico.Limits);
  copia.Markers.AddRange(_grafico.Markers);
  RImage.SavePng(copia, "filtro.png", RImage.Dip(16), RImage.Dip(10), 300);
}
```

Una figura stampata deve avere almeno 300 punti per pollice alla misura in cui va sulla pagina, se no si vede sgranata. `RImage.Dip(16)` dice quanti dip (le unità di Avalonia, 96 per pollice) è largo un oggetto di 16 cm; `SavePng` disegna il controllo, anche senza metterlo in una finestra, a 300 dpi: 1890 pixel di larghezza. E scrive la risoluzione dentro il PNG, così il programma di

impaginazione mette la figura alla misura giusta (`RImage.Dpi` la rilegge). Si disegna una copia del grafico, e non quello della finestra, perché un controllo ha un solo genitore e una sola misura alla volta (capitolo 22).

15.7 Perché sono fatti così

- **Disegnati a mano**, come i grafici del capitolo 11: niente librerie di grafici da installare, ogni pixel si spiega, e senza schermo si provano con il mouse finto (`TEST_RAVAL\ProveGrafica.cs`).
- **Eventi con il valore utile**: `MarkersChanged` , `SelectionChanged(int)` , `Changed(string, string)` si collegano a metodi con un nome, come gli eventi di RAva del capitolo 6.
- **I colori dalle risorse del tema, sempre con una riserva**: lo stesso codice va bene nel tema chiaro e nello scuro, e anche in un'applicazione che quelle risorse non le ha.

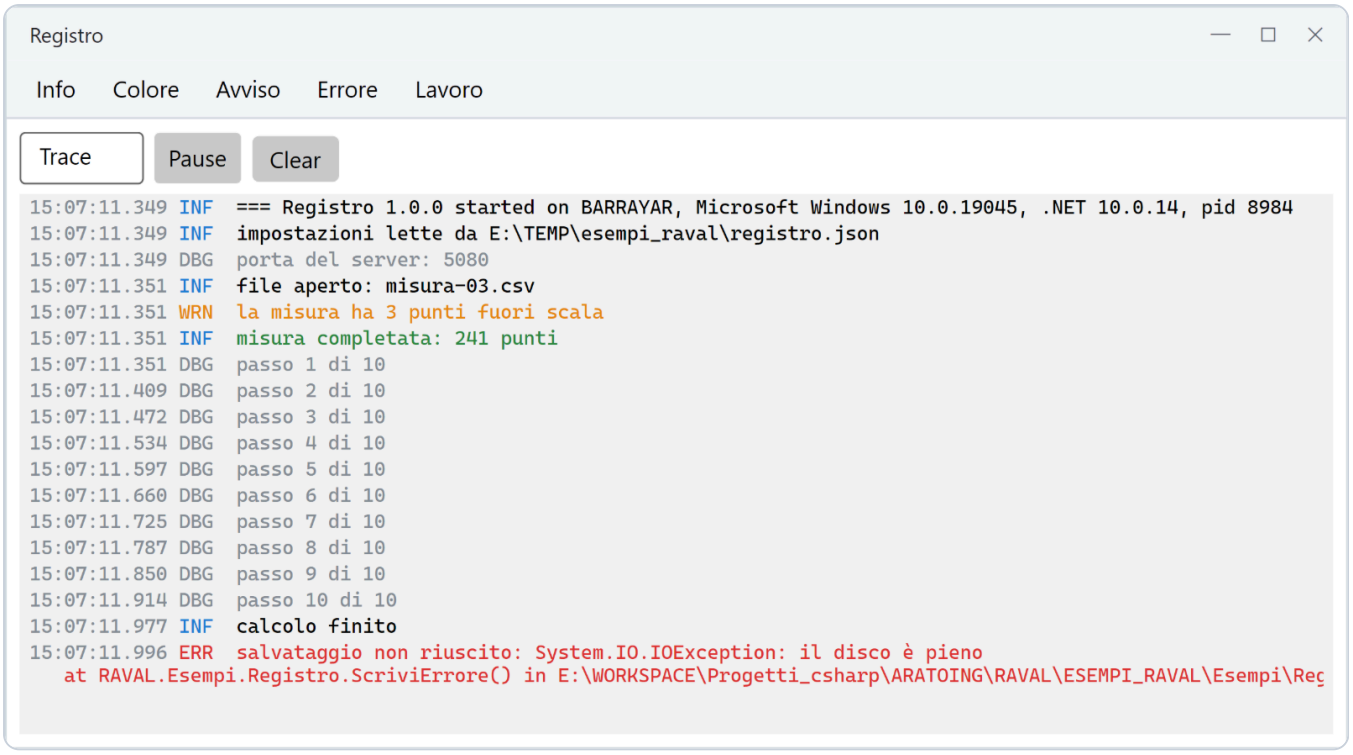
Esercizi

1. Aggiungete una terza riga all'elenco, «Guadagno» in dB, e sommatelo alle due curve.
2. Mettete la fase del filtro (`-celle · atan(ωRC)` , in gradi) sull'asse destro, con `RightAxis = true` e `TitleY2 = "°"` .
3. Collegate `SelectionChanged` della tabella a un metodo che sposta in cima la riga scelta (`Select` , `Rows`).
4. Al bottone «Apri», leggete con `RCsv.Read` un file salvato da «CSV» e mostratelo come terza curva.

16. Il log e le impostazioni

> **Cosa imparerai:** a far raccontare al programma cosa fa, con `RLog` e la finestra `RLogView` , e a fargli ricordare le scelte dell'utente in un file JSON o INI con `RSettings` : la finestra che riapre dov'era e il menu dei file recenti.

Finché il programma gira sul vostro computer, il debugger vi dice tutto. Quando gira su quello di un cliente, l'unica cosa che vi resta è quello che il programma ha scritto: il **log**. E un programma che a ogni avvio ha dimenticato la cartella dell'ultimo progetto, la misura della finestra e la porta del server non piace a nessuno: servono le **impostazioni**. Sono le prime due cose che un programma vero si scrive, e in RAVAL ci sono già.



Il codice è in `ESEMPI_RAVAL\Esempi\Registro.cs`; la scheda completa delle classi in `DOC\RApp.md`.

16.1 Scrivere il log

```
RLog.Start("Registro", "1.0.0", "log/registro.log");
RLog.Info("impostazioni lette da " + _impostazioni.Path);
RLog.Debug("porta del server: " + porta);
RLog.Warning("la misura ha 3 punti fuori scala");
```

`Start` apre il file (e la sua cartella, se manca) e scrive la prima riga: nome e versione del programma, nome del computer, sistema operativo, versione di .NET e numero del processo. Quando un cliente vi manda un log, la prima cosa da sapere è quale versione girava e dove: è scritta lì, a ogni avvio.

Ogni riga ha un **livello**: `Error` (qualcosa non è riuscito), `Warning` (qualcosa di strano che il programma ha saputo gestire), `Info` (cosa fa: avvio, file aperti, collegamenti), `Debug` e `Trace` (i dettagli per chi cerca un difetto). `RLog.Level` dice il livello meno importante che si scrive: con `Info`, il valore predefinito, le righe `Debug` e `Trace` non costano niente. Nel file una riga è così:

```
2026-11-06 10:58:46.988 ERR [1] salvataggio non riuscito:
System.IO.IOException: il disco è pieno
```

data e ora al millesimo, livello, numero del thread che l'ha scritta, testo. Un errore si scrive con la sua eccezione, che porta con sé il tipo, il messaggio e lo stack:

```
// «Errore»: l'eccezione va nel log con il suo tipo e il messaggio
private static void ScriviErrore() {
  try {
    throw new IOException("il disco è pieno");
  } catch (IOException e) {
    RLog.Error("salvataggio non riuscito", e);
  }
}
```

Sulla console le righe sono **colorate**: l'ora attenuata, la sigla del livello nel suo colore (errori rossi, avvisi arancio, Info blu, dettagli grigi) e il testo nel colore del livello. Una riga si può anche colorare a mano, per farla trovare a colpo d'occhio fra cento altre:

```
// «Colore»: una riga Info nel colore scelto, per farla trovare a colpo
d'occhio
private static void ScriviColore() => RLog.Info("misura completata: 241
punti", "#2b8a3e");
```

In Windows Terminal, e nei terminali che lo dichiarano (`COLORTERM=truecolor`), i colori sono quelli esatti; nella vecchia console di Windows sono i più vicini fra i suoi sedici. Quando l'uscita del programma va in un file, i colori spariscono da soli: sarebbero solo caratteri strani in mezzo al testo. Anche il file del log è testo semplice, da leggere con qualunque programma.

`RLog.CatchUnhandled()`, chiamato una volta all'avvio, scrive anche le eccezioni che nessuno prende: quelle che chiudono il programma, che altrimenti sparirebbero con lui. Alla fine si chiama `RLog.Stop()`, che scrive le righe rimaste e chiude il file.

16.2 Perché il log ha un thread suo

Chi chiama `RLog.Info` non scrive sul disco: mette la riga in una **coda** e torna subito. Un thread del log prende le righe una alla volta e le scrive sulla console e nel file. È lo schema **produttore e consumatore**: molti thread producono righe, uno solo le consuma, e fra loro c'è una coda sicura fra thread (`BlockingCollection`, che fa aspettare il consumatore quando è vuota). Ha tre vantaggi:

- una riga di log costa uguale sul thread dell'interfaccia e su un thread di lavoro: il disco lento non blocca la finestra;
- le righe di thread diversi non si mescolano mai a metà, perché le scrive un thread solo;

- il bottone «Lavoro» dell'esempio fa partire un thread che scrive dieci righe, e non serve nessun `lock`.

Il file **ruota**: quando supera `MaxBytes` (10 MB) diventa `registro.1.log`, il vecchio `.1` diventa `.2`, e così via fino a `MaxFiles` (5). Un programma che resta acceso per mesi non riempie il disco.

16.3 Il log in una finestra: RLogView

```
Vista = new DockPanel().Children(barra.Dock(Dock.Top), new
RLogView().Margin(8));
```

`RLogView` mostra il log dal vivo, con gli stessi colori della console e quelli scelti a mano. Il menu a sinistra sceglie il livello mostrato, «Pause» smette di seguire l'ultima riga, «Clear» svuota la vista (il file resta). Quando la vista entra in una finestra mostra anche le ultime righe scritte prima (`RLog.Recent()`, le ultime 2000): si può aprire la finestra del log quando c'è un problema e vedere cosa è successo.

Le righe arrivano sul thread del log, e i controlli si toccano solo dal thread dell'interfaccia (capitolo 10). La vista mette le righe in una coda sua e chiede al Dispatcher **un solo** lavoro, che sposta tutte quelle arrivate nel frattempo: mille righe in un colpo costano un aggiornamento, non mille. L'elenco è virtualizzato (capitolo 8): si costruiscono solo le righe visibili.

16.4 Le impostazioni

```
_impostazioni = RSettings.Open(Path.Join(cartella, "registro.json"));
int porta = _impostazioni.GetInt("server.port", 5080);
bool dettagli = _impostazioni.GetBool("log.debug", true);
```

`RSettings.Open` legge il file; con l'estensione «.ini» è un file INI, con le altre un file JSON. Le chiamate sono le stesse per i due formati. Ogni lettura ha il suo **predefinito**: se il file non c'è, se la chiave manca, se il valore non è un numero, il programma va avanti con il valore che ha scritto il programmatore.

Le chiavi sono percorsi con i punti. Dopo il primo avvio il file JSON è così:

```
{
  "server": {
    "port": 5080
  },
  "log": {
    "debug": true
  }
}
```

La porta non l'abbiamo scritta noi: una chiave letta e non trovata **si aggiunge al file** con il suo predefinito (`AddMissing`). Dopo il primo avvio il file elenca tutte le opzioni che il programma legge, e chi lo apre per cambiarle le trova tutte. Lo stesso programma con un file INI scriverebbe:

```
[server]
port = 5080

[log]
debug = true
```

e la sezione è tutto quello che sta prima dell'ultimo punto della chiave. Nel file INI i commenti (`;` e `#`), le righe vuote e l'ordine delle chiavi restano dove li ha messi l'utente: cambiare un valore riscrive solo la sua riga.

Per scrivere si usa `Set`:

```
_impostazioni.Set("server.port", 5081);
```

`Set` non scrive il file subito: lo salva mezzo secondo dopo l'ultima modifica (`SaveDelayMs`). Una finestra trascinata manda centinaia di posizioni al secondo, e il file si scrive una volta sola. Il salvataggio scrive un file temporaneo e poi lo rinomina: se il programma si chiude a metà, resta il file vecchio, non uno spezzato. E un file che non si riesce a leggere non si butta: diventa `registro.json.bad`, il log lo dice, e il programma parte con i predefiniti.

Dove si mettono le impostazioni di un programma installato? Non nella sua cartella, che spesso è di sola lettura: `RSettings.Folder("Registro")` dà la cartella giusta per l'utente (`%APPDATA%\Registro` su Windows, `~/.config/Registro` su Linux e macOS).

16.5 La finestra che si ricorda, i file recenti

```
public FinestraPrincipale(RSettings s) {
  s.Remember(this, "window.main");                    // prima di Show
  _recenti = new RRecentFiles(s, "files.recent");
  _recenti.Opened += Apri;                            // un metodo con il
percorso
  _recenti.Fill(_menuRecenti);                        // le voci di File →
Recenti
}
```

`Remember` rimette la finestra dove l'utente l'aveva lasciata, con la sua misura, e se era ingrandita la riapre ingrandita. Ricorda la misura di quando la finestra era normale: ridotta, torna al suo posto e non alla misura dello schermo. Se lo schermo su cui stava non c'è più (il portatile staccato dal monitor), la finestra si apre al centro dello schermo principale invece che fuori dalla vista.

`RRecentFiles` tiene gli ultimi file aperti, il più recente in cima; un file riaperto torna in cima invece di comparire due volte. `Fill` riempie un menu con una voce per file, e lo tiene aggiornato; dopo ogni apertura o salvataggio si chiama `_recenti.Add(percorso)`.

16.6 Il tema: chiaro, scuro o come il sistema

Il capitolo 7 spiega come funzionano i temi di Avalonia. In un programma vero servono anche tre cose in più: lasciare scegliere all'utente, ricordare la sua scelta, e seguire il sistema operativo se l'utente preferisce così.

```
public override void Initialize() {
  Styles.Add(new FluentTheme());
  RThemes.Standard().Color("Marchio", light: "#8f4a1c", dark:
"#e39a5c").Apply(this);
  RTheme.Remember(impostazioni);                  // «ui.theme», applicato
subito
}

RTheme.Fill(menuTema);                            // Chiaro / Scuro / Come il
sistema
barra.Children.Add(RTheme.Button());              // il sole e la luna
```

`RThemes.Standard()` dà i colori che usano i controlli di RAVAL (`Back`, `Panel`, `Ink`, `InkSoft`, `Border`, `Bar`, `ChartBack`...), nei due temi. È la tavolozza del simulatore ASTRIP. Lo scuro è una grafite con una punta di blu, su tre livelli di luce: la barra più scura, poi i documenti, poi i pannelli

più chiari. Così l'occhio vede dove finisce una zona senza bisogno di bordi pesanti. Il testo non è bianco puro, che sul fondo scuro abbaglia.

`RTheme.Mode` cambia il tema di tutta l'applicazione mentre il programma gira, finestre già aperte comprese. Con `System` decide il sistema operativo, e il programma lo segue quando cambia. `Remember` legge la scelta dalle impostazioni e la riscrive a ogni cambio. `Changed` avvisa chi deve rifare qualcosa a mano.

Un controllo disegnato a mano cambia con il tema solo se chiede i colori mentre disegna (`ThemeBrush` dentro `Render`) e si ridisegna al cambio (`OnThemeChanged`). Un pennello preso all'avvio e tenuto in un campo resta del colore del primo tema: è l'inciampo che ASTRIP ha trovato sulle sue carte.

16.7 Le lingue: RLang

```
RLang.Open(RSettings.Folder("Misure") + "/lingue", "it");    // «»: la
lingua del sistema
var file = new MenuItem().Tr("menu.file", "_File");         // cambia con la
lingua
string s = RLang.T("misura.salvata", "Measure saved");
string n = RLang.T("file.aperti", "{0} files open", 3);
RLang.Use("en");                                             // tutti i
testi legati cambiano
```

Una lingua è un file nella cartella, «it.json» o «it.ini», letto con `RSettings`. Le chiavi sono percorsi con i punti, come nelle impostazioni. Il testo scritto nel codice è il predefinito, e per la prima lingua non serve nessun file. Una chiave chiesta e non trovata viene scritta nel file con il suo testo predefinito: dopo una prova del programma, il file di una lingua nuova elenca tutti i testi, pronti da tradurre. Il file nasce mezzo secondo dopo la prima chiave aggiunta, oppure con `RLang.Save()`.

`Tr` lega un controllo a una chiave, scegliendo la proprietà giusta in base al controllo:

- `Text` per un `TextBlock`;
- `Header` per una voce di menu;
- `Title` per una finestra;
- `PlaceholderText` per una casella;
- `Content` per un bottone.

Quando la lingua cambia, ogni controllo legato prende il testo nuovo. I legami sono deboli: un controllo buttato via non resta in memoria per colpa loro. I testi di RAVAL stessa hanno le chiavi

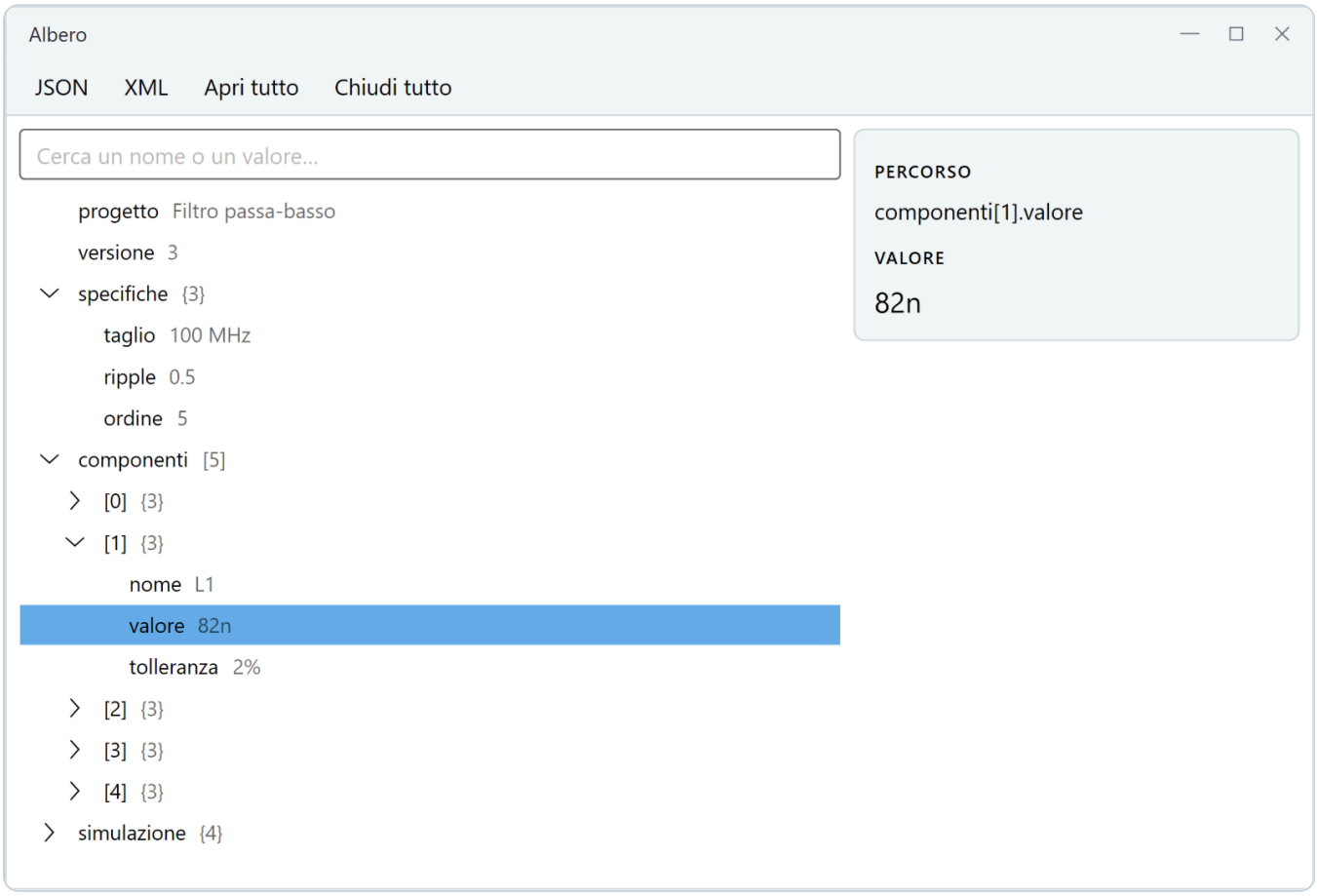
«raval.*» e l'italiano già pronto: con `Use("it")` anche i bottoni e i titoli di `RDialog` e il menu del tema sono in italiano.

Esercizi

1. Aggiungete all'esempio un `CheckBox` «Dettagli» che cambia `RLog.Level` fra `Info` e `Debug` e salva la scelta in `log.debug`.
2. Cambiate il file in `registro.ini` e guardate come lo scrive `RSettings`; aggiungete a mano un commento e controllate che resti al suo posto dopo un `Set`.
3. Chiamate `RLog.CatchUnhandled()` e fate un bottone che lancia un'eccezione da un thread: cercatela nel file.
4. Scrivete un programma con un menu File → Recenti fatto con `RRecentFiles`, e controllate che dopo un riavvio la lista ci sia ancora.

17. Gli alberi: JSON, XML e cartelle

Un file di progetto, una risposta JSON di un server, la configurazione di un apparato in XML, le cartelle di un disco: sono tutti alberi, e il modo naturale di guardarli è un `TreeView`. In Avalonia il `TreeView` si riempie di solito con un modello di dati e un `TreeDataTemplate` scritto in XAML. `RTree` fa lo stesso lavoro in codice: si danno i nodi, e il controllo pensa alle voci sullo schermo.



L'esempio completo è `ESEMPI_RAVAL/Esempi/Albero.cs`: a sinistra il progetto di un filtro, in JSON o in XML, con la casella di ricerca; a destra il percorso e il valore del nodo scelto.

17.1 I nodi: RTreeNode

Un nodo ha un testo, un valore che si vede più tenue accanto al testo, e i figli:

```
var server = new RTreeNode("server").Add(
  new RTreeNode("host", "localhost"),
  new RTreeNode("port", "5080"));
```

`Tag` tiene quello che serve al programma (l'oggetto da cui il nodo è nato), `Color` colora il testo, `Key` è il pezzo di percorso del nodo (il testo, se non lo si cambia). `Path(".")` mette insieme le chiavi dalla radice: «server.port», e per gli elementi di una lista «componenti[1].valore», senza punto davanti alla parentesi. Sono le stesse chiavi che legge `RSettings` (capitolo 16).

I nodi sono i dati, non le voci sullo schermo: si possono costruire, cercare e percorrere (`Children`, `Child("port")`, `Descendants()`) anche senza una finestra.

17.2 Da JSON e da XML

```
RTree albero = RTree.FromJson(testo);        // i membri dell'oggetto in
cima
RTree albero = RTree.FromXml(testo);         // l'elemento radice in cima
albero.LoadJson(altroTesto);                 // lo stesso controllo, un
altro file
```

Nel JSON gli oggetti e le liste diventano nodi con i membri come figli, e il valore dice quanti sono: «{3}» per un oggetto, «[5]» per una lista; numeri, testi e `true` / `false` sono foglie con il loro valore. I commenti e la virgola prima di una parentesi chiusa sono accettati, come nei file di impostazioni. Nell'XML ogni elemento è un nodo, con gli attributi come primi figli («@nome») e il testo come valore. Il testo lo legge `RNode`, del paragrafo che segue: ogni nodo dell'albero tiene nel suo `Tag` l'`RNode` da cui è nato.

17.3 I dati senza l'albero: RNode

Spesso un file JSON o XML non si deve guardare, si deve leggere: la porta di un server, il valore di un componente, la lista dei file recenti. Per questo c'è `RNode`, l'albero dei dati senza niente di grafico. Non usa nessun tipo di Avalonia, quindi va bene anche in un servizio, in un programma a riga di comando o in una prova.

```
RNode p = RNode.Load("progetto.json");           // «.xml»: XML; il resto:
JSON
string taglio = p.GetString("specifiche.taglio", "100 MHz");
int punti = p.GetInt("simulazione.punti", 201);
foreach (RNode c in p["componenti"]!.Children)
  Console.WriteLine(c.GetString("nome", "?") + " " + c.GetString("valore",
"));

p.Find("simulazione.punti")!.Set(801);           // un numero
p["componenti"]!.Add(RNode.Object().Add(RNode.Text("nome", "C4"),
RNode.Text("valore", "10p")));
p.Save("progetto.json");                         // un file temporaneo, poi
rinominato
```

Le letture hanno un predefinito, come quelle di `RSettings` (capitolo 16): una chiave che manca non ferma il programma. Il percorso è lo stesso dappertutto: «specifiche.taglio», «componenti[1].valore», e nell'XML «server/@port» con il separatore «/» (`p.GetInt("server/@port", 80, "/")`). La radice di un documento JSON non ha nome e nel percorso non compare.

Ogni nodo ha un tipo (`Kind` : oggetto, lista, valore, elemento, attributo), un nome, un valore come testo e i figli. `Source` tiene il `JsonNode` o l' `XElement` da cui è stato letto, per chi ha bisogno del resto. Un nodo aggiunto a una lista prende il nome «[n]» dalla sua posizione. Un testo cambiato con `Set` si riscrive fra virgolette, un numero o un sì/no senza.

Si può anche passare da un formato all'altro. `ToJson()` scrive anche un albero letto dall'XML: gli attributi diventano membri «@nome», il testo diventa «#text» e gli elementi che si ripetono diventano una lista. `ToXml()` scrive un JSON come XML: la radice si chiama «root» e gli elementi di una lista «item».

Per vedere un `RNode` nella finestra: `albero.LoadNode(p)` , oppure `RTreeNode.FromNode(p)` per metterlo sotto un altro nodo.

17.4 Il controllo: scegliere, aprire, cercare

```
albero.Searchable = true;                    // la casella di ricerca sopra
albero.SelectionChanged += Scelto;           // void Scelto(RTreeNode? n)
albero.Opened += Apri;                       // doppio clic o Invio
albero.Select(albero.Find("componenti[1].valore", "."));
```

`Find` legge un percorso scritto come lo scrive `Path` ; `Select` apre i nodi sopra quello scelto e lo porta in vista. `Expand` , `ExpandAll(livelli)` e `CollapseAll` aprono e chiudono. Il filtro

(`Filter` , o la casella di ricerca) tiene i nodi il cui testo o valore contiene le parole cercate, senza badare alle maiuscole, con i nodi sopra (aperti, così quello che si è trovato è in vista) e quelli sotto; `Found` dice quanti sono.

17.5 Le voci nascono quando servono

La voce di un nodo (`TreeViewItem`) si costruisce solo quando il nodo sopra viene aperto. Finché resta chiuso, al suo posto c'è una voce finta che serve solo a far vedere la freccia. Un JSON di centomila valori si apre in una frazione di secondo, perché sullo schermo c'è una voce sola: la prova (`TEST_RAVAL/ProveAlbero.cs`) lo misura.

Lo stesso vale per i dati, quando leggerli costa: un nodo può avere un *caricatore* al posto dei figli, un metodo che li legge la prima volta che servono.

```
var disco = RTreeNode.FromFolder("C:\\");    // le cartelle si leggono
quando si aprono
albero.Add(disco);

var ordini = new RTreeNode("ordini", LeggiOrdini);
// IEnumerable<RTreeNode> LeggiOrdini(RTreeNode nodo) { ... la query ... }
```

`FromFolder` mostra prima le cartelle e poi i file con la misura (`RFormat.Si` , «2.05 kB»). Un caricatore che fallisce (una cartella senza permessi) non rompe l'albero: il nodo mostra un figlio rosso con l'errore, e il log ne tiene traccia. `Reload()` dimentica i figli letti e li fa rileggere. Il filtro cerca solo fra i figli già letti: non legge un disco intero per una ricerca.

Un nodo cambiato si vede subito: `Add` , `Remove` e i nuovi `Text` o `Value` aggiornano le voci che ci sono già.

17.6 Perché è fatto così

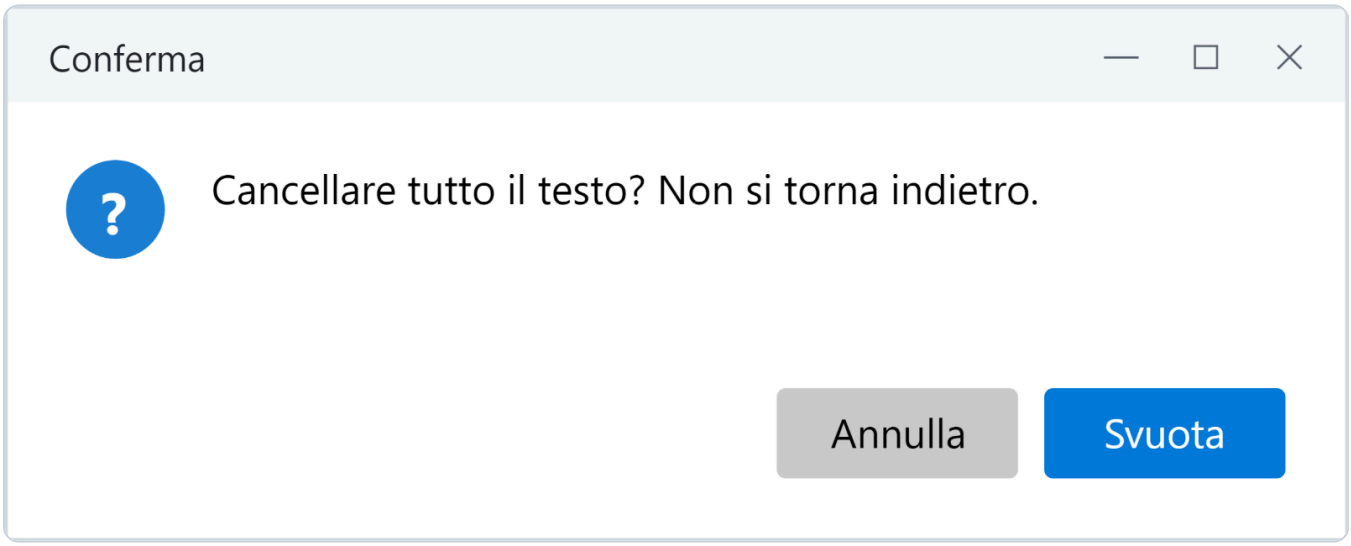
- **Niente modello di dati né modelli in XAML.** Con `ItemsSource` e un `TreeDataTemplate` servono una classe del modello, un modello per le voci e i binding fra le due; qui le voci si fanno a mano, in un metodo che si legge dall'alto in basso.
- **Tre strati: i dati, i nodi, le voci.** `RNode` sono i dati, senza Avalonia: si leggono in un altro thread (un file grande in un `Task`) e servono anche ai programmi senza finestre. `RTreeNode` è il nodo dell'albero visuale, con il colore e i caricatori. Le voci (`TreeViewItem`) esistono solo per la parte di albero che si vede.
- **Il valore tenue è trasparente, non grigio.** Un grigio fisso sulla riga scelta, azzurra, quasi sparisce; con l'opacità al 60% il valore prende il colore del testo della riga, chiaro o scuro, e si legge sempre.

Esercizi

1. Aggiungete all'esempio un bottone «Cartella» che mostra le cartelle dei documenti con un caricatore vostro, che scrive nel log una riga `Debug` ogni volta che viene chiamato: aprendo i nodi, guardate quando arriva.
2. Con `Opened` aprite un file scelto nell'albero delle cartelle e mostratene la misura in una `RPropertyList`.
3. Caricate nell'albero il file delle impostazioni del capitolo 16 e fate in modo che il percorso del nodo scelto sia proprio la chiave da passare a `GetString`.

18. Dialoghi e file in una riga

Nel capitolo 9 la finestra di conferma è una classe di trenta righe, e scegliere un file vuol dire costruire le opzioni, i tipi e leggere il percorso da un `IStorageFile`. Sono cose che un programma fa dieci volte, e ogni volta uguali. `RDialog` e `RFiles` le riducono a una riga ciascuna.



L'esempio completo è `ESEMPI_RAVAL/Esempi/Appunti.cs`, un blocco note in cui ogni bottone della barra è un metodo `async` di poche righe.

18.1 I dialoghi: RDialog

```
await RDialog.Info(Vista, "Salvato.");
if (!await RDialog.Confirm(Vista, "Cancellare tutto il testo?", "Svuota"))
return;
string? titolo = await RDialog.Ask(Vista, "La prima riga degli appunti:",
righe[0]);
int a = await RDialog.Choose(Vista, RDialogKind.Warning, "Non salvato",
"Salvare le modifiche?",
                             "Annulla", "Scarta", "Salva");
await RDialog.Error(Vista, "Gli appunti non sono stati salvati.", e);
```

Il primo argomento è un controllo qualunque della finestra: il dialogo si apre sopra quella finestra, al centro, e finché non si chiude il resto della finestra non risponde (è *modale*). Il controllo deve già essere in una finestra, se no la chiamata dà un errore chiaro.

I bottoni stanno a destra, nell'ordine dato. L'ultimo è quello che conferma: ha il colore d'accento e risponde a Invio. Esc e la crocetta della finestra non scelgono nessun bottone: `Confirm` dà `false`, `Ask` dà `null`, `Choose` dà −1. Il segno a sinistra dice di che si tratta (`RDialogKind`): «i» blu per le informazioni, «!» arancio per gli avvisi, «×» rosso per gli errori, «?» blu per le domande.

`Error` con un'eccezione fa tre cose. Scrive il messaggio dell'eccezione sotto il testo, mette l'eccezione intera sotto «Dettagli», dove si può copiare per chi deve correggere, e la registra nel log con `RLog.Error`: quando l'utente chiude il riquadro, l'errore non è perso.

I testi dei bottoni e dei titoli sono in inglese, come il resto della libreria. Un programma in italiano li cambia una volta, all'avvio:

```
RDialog.Cancel = "Annulla";
RDialog.ConfirmTitle = "Conferma";
RDialog.ErrorTitle = "Errore";
RDialog.Details = "Dettagli";
```

Chi vuole qualcosa di diverso sotto il testo (una lista, una casella di spunta «non chiedere più») costruisce un `RDialogBox` con il suo controllo e lo apre con `RDialog.Show`. L'evento `RDialog.Opening` consegna ogni dialogo prima che si apra: le prove (`TEST_RAVAL/ProveDialoghi.cs`) lo usano per premere i bottoni con `Pick`, come farebbe l'utente.

18.2 I file: RFiles

```
string? f = await RFiles.OpenAsync(Vista, "Apri un file di testo",
"Testo|*.txt;*.md", "Tutti|*.*");
if (f == null) return;                              // l'utente ha
rinunciato
_testo.Text = RFiles.ReadText(f);

string? s = await RFiles.SaveAsync(Vista, "Salva gli appunti",
"appunti.txt", "Testo|*.txt");
if (s != null) RFiles.WriteText(s, _testo.Text ?? "");
```

I tipi di file si scrivono come testo, «nome|*.a;*.b». La risposta è il percorso, che `StreamReader` e `StreamWriter` usano direttamente, oppure `null` se l'utente rinuncia. `OpenManyAsync` dà una lista di percorsi e `FolderAsync` una cartella. Il selettore successivo si apre nella cartella dell'ultimo file scelto (`RFiles.LastFolder`). `SaveAsync` chiede prima di sostituire un file che c'è già, e aggiunge l'estensione del primo tipo se l'utente non ne scrive una.

`ReadText` e `WriteText` leggono e scrivono un file di testo intero in UTF-8. `WriteText` crea la cartella se manca e scrive prima in un file temporaneo, che poi rinomina: se il programma si ferma a metà, resta il file vecchio, intero.

18.3 Perché sono fatti così

- **`await` e non una funzione da richiamare.** Il codice si legge dall'alto in basso: la domanda, la risposta, quello che si fa dopo. Con `async void` nei gestori dei bottoni (capitolo 2) la finestra resta viva mentre il dialogo aspetta.
- **La risposta è nel dialogo, non nel valore di `Close`.** `ShowDialog<int>` restituisce 0 quando la finestra si chiude con la crocetta, cioè lo stesso valore del primo bottone. `RDialogBox` tiene la risposta in `Answer`, che vale −1 finché non si preme un bottone: chiudere non è mai scambiato per una scelta.
- **I percorsi come testo.** `IStorageFile` serve dove i file non sono sul disco (un telefono, una sandbox); un programma da scrivania vuole un percorso. Un file senza percorso sul disco dà `null`, come una rinuncia.

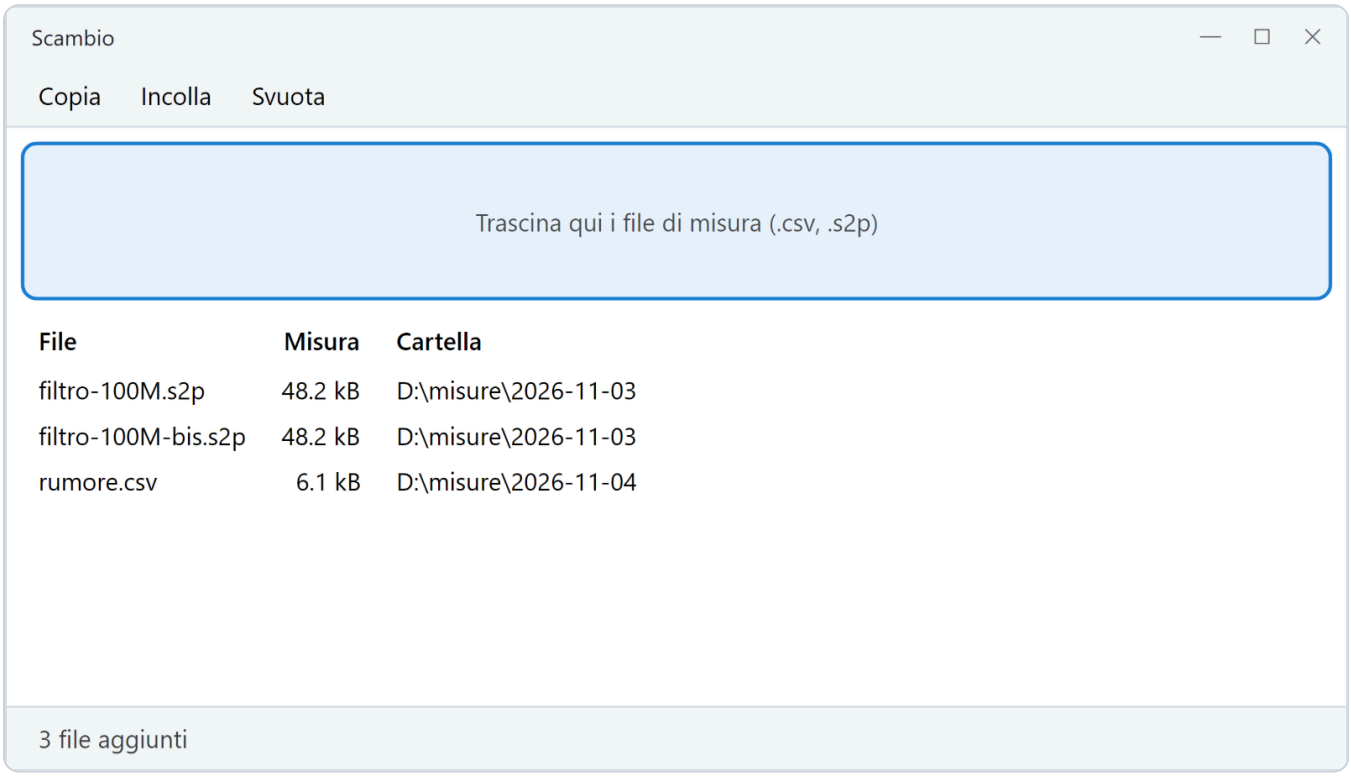
Esercizi

1. Aggiungete al blocco note la domanda «Salvare le modifiche?» con `Choose` e i tre bottoni Annulla, Scarta, Salva, quando si apre un altro file dopo aver cambiato il testo.
2. Con `OpenManyAsync` aprite più file e mettete i loro nomi in una `RTable` con la misura di ciascuno.

3. Costruite un `RDialogBox` con una `CheckBox` «Non chiedere più» sotto il testo, e salvate la scelta con `RSettings`.

19. Appunti e trascinamento

Due modi di far entrare e uscire i dati da un programma senza passare dai menu. Gli appunti servono a copiare un valore, o una tabella di risultati da incollare in un foglio di calcolo. Il trascinamento serve a lasciare i file di misura sulla finestra invece di cercarli con «Apri». `RClipboard` e `RDrop` li fanno in una riga.



L'esempio completo è `ESEMPI_RAVAL/Esempi/Scambio.cs`: i file lasciati sul riquadro diventano righe della tabella, e la tabella si copia e si incolla da e verso un foglio di calcolo.

19.1 Gli appunti: RClipboard

```
await RClipboard.SetText(Vista, "1.296 GHz");
string? t = await RClipboard.GetText(Vista);              // null se negli
appunti non c'è testo

await RClipboard.SetTable(Vista, righe);                  // si incolla in
un foglio, una cella per valore
List<string[]> incollate = await RClipboard.GetTable(Vista);

await RClipboard.SetImage(Vista, RImage.Render(grafico, 600, 400, 192));
List<string> file = await RClipboard.GetFiles(Vista);     // i file copiati
in Esplora risorse
```

In Avalonia 12 gli appunti appartengono alla finestra (`TopLevel.Clipboard`) e lavorano con formati e oggetti del disco; qui, come per i dialoghi, il primo argomento è un controllo qualunque della finestra e le risposte sono testi, righe e percorsi.

Una tabella va negli appunti come la copia un foglio di calcolo: una tabulazione fra le celle e un a capo dopo ogni riga. È il formato che tutti i fogli capiscono, nei due sensi: i risultati di una misura vanno in un foglio, e una colonna copiata da un foglio torna come righe (`Cells` fa il lavoro su un testo qualunque). Una tabulazione o un a capo dentro una cella diventano spazi, altrimenti la cella si spezzerebbe.

Un grafico si copia come immagine disegnandolo con `RImage.Render`: a 192 dpi, il doppio dello schermo, resta nitido quando lo si incolla in un documento.

19.2 Il trascinamento: RDrop

```
RDrop.Files(Area, Arrivati, "*.csv", "*.s2p", "*.s1p");   // void
Arrivati(List<string> file)
RDrop.Text(note, Incolla);                                // void
Incolla(string testo)
```

Il controllo accetta soltanto quello che sa usare. Mentre si trascinano dei file sopra di lui, il puntatore mostra «copia» se almeno uno corrisponde ai modelli e «vietato» altrimenti. Quando l'utente lascia andare, il metodo riceve soltanto i percorsi giusti. I modelli sono quelli dei nomi di file: `*` per qualunque sequenza di caratteri, `?` per un carattere solo, senza badare alle maiuscole. Senza modelli va bene ogni file.

Mentre sopra il controllo c'è qualcosa che accetta, il controllo ha la classe `drop`. Con `RStyles.Clean`, un `Border` con la classe `droparea` è un riquadro con il bordo grigio; con

`drop` il bordo diventa blu e lo sfondo azzurrino, come nella figura. `Stop()` toglie tutto e il controllo non accetta più niente.

19.3 Perché sono fatti così

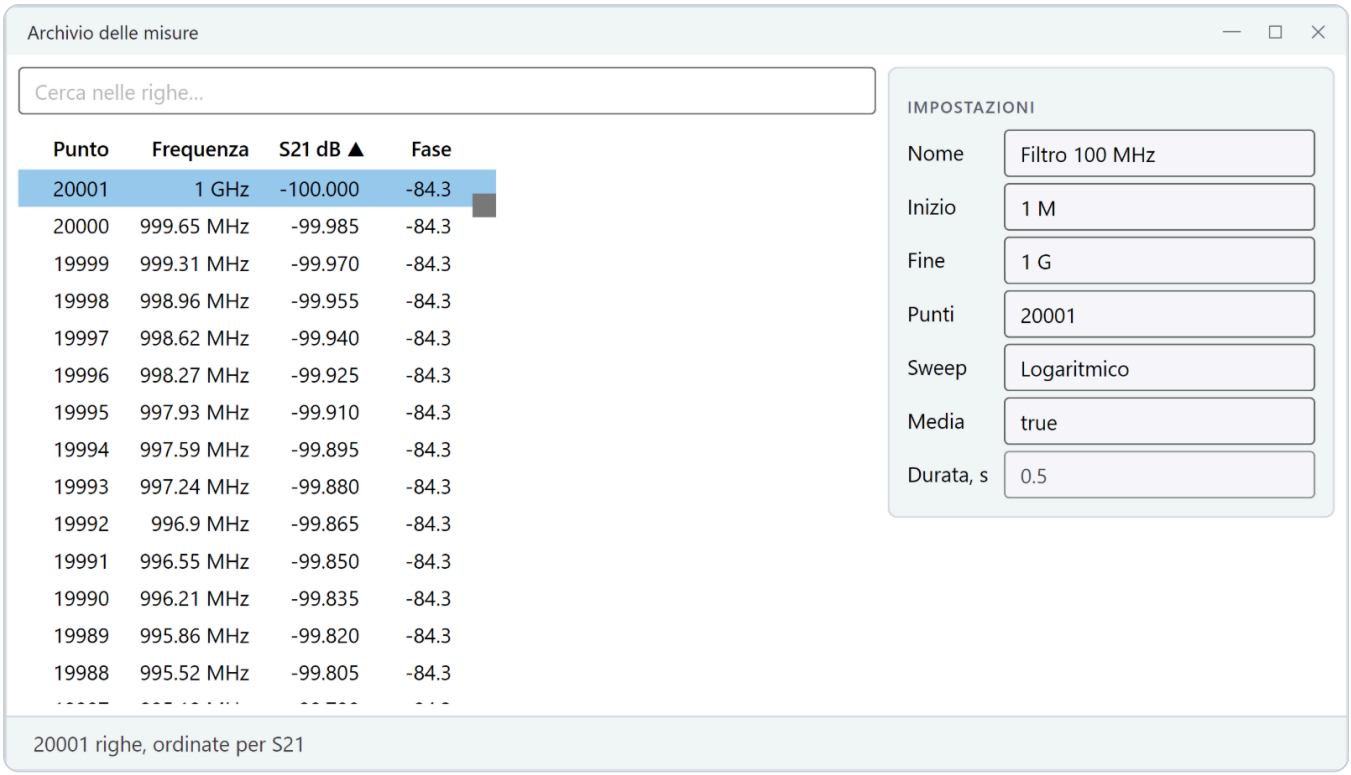
- **Il foglio di calcolo come formato di scambio.** Chi misura vuole i numeri in un foglio, e chi ha i dati in un foglio vuole incollarli nel programma; la tabulazione funziona senza aprire file, nei due sensi.
- **Il controllo dice subito se accetta.** Un file sbagliato è rifiutato mentre è ancora sotto il puntatore, non con un messaggio d'errore dopo averlo lasciato.
- **Il bordo dallo stile, non dal controllo.** Un valore scritto sul controllo (`BorderBrush = ...`) vince su ogni stile, anche su quello di `drop`: il riquadro dell'esempio, con il bordo messo a mano, si colorava dentro ma il bordo restava grigio. Per questo bordo e angoli vengono dalla classe `droparea`.

Esercizi

1. Aggiungete allo scambio un bottone «Copia grafico» che mette negli appunti un `RChart` disegnato con `RImage.Render`, e incollatelo in un documento.
2. Fate accettare al blocco note del capitolo 18 un file `.txt` trascinato sopra, che lo apre come «Apri».
3. Con `RClipboard.GetTable` incollate due colonne copiate da un foglio (frequenza e S21) e disegnatele in un `RChart`.

20. Tabelle grandi e oggetti da modificare

`RTable`, del capitolo 15, crea un `TextBlock` per ogni cella: va benissimo per cento righe, ma con diecimila diventa lenta, e con centomila la finestra non si apre più. Un archivio di misure, il registro di un apparato o uno sweep di ventimila punti chiedono un'altra tabella. Allo stesso modo `RPropertyList`, con le coppie nome e valore aggiunte a mano, costringe a ricopiare i valori in un oggetto ogni volta che cambiano. Il capitolo risolve tutte e due le cose.



L'esempio completo è `ESEMPI_RAVAL/Esempi/Archivio.cs`.

20.1 La tabella virtualizzata: RVirtualTable

```
var t = new RVirtualTable();
t.LoadCsv(RFiles.ReadText("sweep.csv"));      // la prima riga: le
intestazioni
t.SelectionChanged += Scelta;                 // void Scelta(int riga)
t.Filter = "C1";                              // solo le righe con C1 in una
cella
t.Sort(2, false);                             // per la terza colonna
```

Le righe sono le voci di un `ListBox` *virtualizzato*: sullo schermo si costruiscono solo quelle che si vedono, una quindicina, e mentre si scorre si rifanno quelle che entrano. Così un CSV di centomila righe si apre in un terzo di secondo: la prova (`TEST_RAVAL/ProveVirtuale.cs`) lo misura, e conta le righe costruite.

Le colonne hanno una larghezza fissa, misurata sul testo più lungo dell'intestazione e delle prime mille righe (`SetWidth` per sceglierla). L'intestazione scorre di lato insieme alle righe. `LoadCsv` allinea a destra le colonne di soli numeri. Un clic su un'intestazione ordina: per valore se ogni cella è un numero, anche con il prefisso SI, se no in ordine alfabetico; un secondo clic rovescia. Centomila righe si ordinano in meno di due decimi di secondo, perché ogni cella si legge una volta sola e le celle che non sono numeri si scartano senza eccezioni (`RFormat.TryParse`).

`Rows` tiene tutte le righe; `Shown` quelle che passano il filtro, nell'ordine in cui si vedono. `SelectedIndex` e l'evento indicano sempre una riga di `Rows`, comunque la tabella sia ordinata o filtrata. `Text()` dà le righe mostrate con le tabulazioni, pronte per `RClipboard.SetText`.

20.2 Le proprietà di un oggetto: RPropertyList.Show

```
public sealed class Impostazioni {
  public string Nome { get; set; } = "Filtro 100 MHz";
  [DisplayName("Inizio"), Description("Hz; si può scrivere 1 MHz")]
  public double Inizio { get; set; } = 1e6;
  public int Punti { get; set; } = 20001;
  public Sweep Sweep { get; set; } = Sweep.Logaritmico;
  public bool Media { get; set; } = true;
  public double Durata => Math.Round(Punti * 25e-6, 3);     // solo get: in
sola lettura
}

_impostazioni.Show(new Impostazioni());
_impostazioni.Changed += Cambiata;            // dopo che il valore è
nell'oggetto
_impostazioni.Rejected += Rifiutata;          // un testo che non si
converte, con il motivo
```

`Show` elenca le proprietà pubbliche che stanno in una casella: testi, numeri, sì/no ed `enum`, nell'ordine della classe. Quando si conferma un valore, con Invio o uscendo dalla casella, il testo torna nell'oggetto convertito nel tipo della proprietà:

- i numeri si leggono come li legge `RFormat`, quindi «1 MHz», «4k7» e «1.5p» sono numeri;
- i sì/no si scrivono come in `RSettings`;
- gli `enum` per nome, senza badare alle maiuscole.

Un testo che non si converte, come «4.5» per un intero, è rifiutato: la casella torna al valore dell'oggetto e `Rejected` dice il motivo. Gli attributi di `System.ComponentModel` cambiano quello che si vede:

- `DisplayName` per il nome mostrato;
- `Description` per il suggerimento;
- `ReadOnly(true)`, o una proprietà senza `set` pubblico, per una casella che non si cambia;
- `Browsable(false)` per non mostrarla.

`Reload()` rilegge l'oggetto quando è cambiato altrove: nell'esempio, la durata ricalcolata dopo che si sono cambiati i punti.

20.3 Perché sono fatti così

- **Righe come array di testo.** Una tabella che mostra deve sapere scrivere, non conoscere il modello dei dati: con `string[]` la stessa tabella mostra un CSV, il risultato di una query o un registro.
- **I numeri dei tecnici, anche con l'unità.** «10 MHz» scritto in una casella vale 1e7 anche quando nessuno dice che l'unità è «Hz»: `RFormat.Parse`, senza unità indicata, prova a tenere la prima lettera come prefisso e poi lascia cadere il resto («5 V» vale 5). I numeri molto grandi o molto piccoli si mostrano con il prefisso («1 M»), e si rileggono uguali.
- **L'oggetto è la verità.** Dopo una conferma la casella mostra quello che c'è nell'oggetto, non quello che si è scritto: se un `set` arrotonda o rifiuta, lo si vede subito.

Esercizi

1. Aggiungete all'archivio un bottone «Apri CSV» con `RFiles.OpenAsync`, e la possibilità di trascinare un CSV sulla tabella con `RDrop.Files`.
2. Fate un bottone «Copia» che mette negli appunti le righe filtrate con `t.Text()`, e incollatele in un foglio di calcolo.
3. Mostrate con `Show` un oggetto vostro con un `set` che rifiuta i valori negativi lanciando `ArgumentException`, e guardate il motivo che arriva a `Rejected`.

21. Altri grafici: barre, mappe, carta di Smith e polare

`RChart`, del capitolo 15, disegna curve su assi cartesiani. Un progetto a radiofrequenza ne chiede altri quattro:

- la **carta di Smith**, per i coefficienti di riflessione;
- il **grafico polare**, per le trasmissioni e i diagrammi d'antenna;
- le **barre**, per confrontare valori di componenti;
- la **mappa di colori**, per una grandezza su un piano, come la corrente su un circuito stampato.

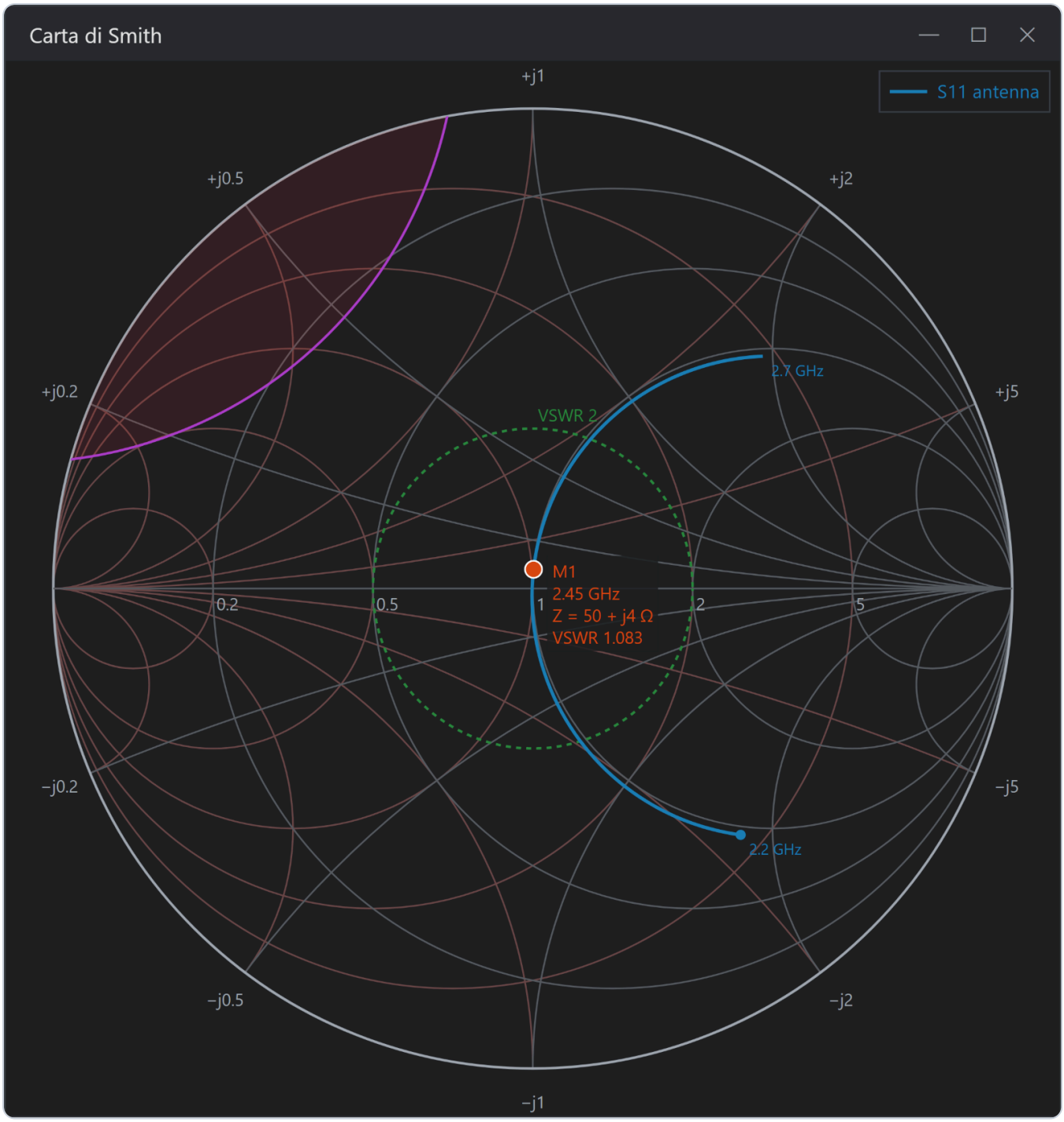
Sono disegnati a mano come `RChart`, prendono i colori dalle stesse risorse del tema e si salvano a 300 dpi con `SavePng`.



L'esempio è `ESEMPI_RAVAL/Esempi/GraficiRF.cs`. La carta di Smith e il grafico polare vengono dal simulatore ASTRIP, portati in RAVAL e completati: ASTRIP stessa ora usa questi. Il capitolo del libro che spiega come si costruisce una carta di Smith ne descrive una versione didattica, più semplice.

21.1 La carta di Smith: RSmithChart

```
var s = new RSmithChart { Z0 = 50, ShowAdmittance = true };
s.Series.Add(new RComplexSeries("S11", gamma, frequenze));
s.Circles.Add(new RSmithCircle("VSWR 2", RSmithMath.Vswr(2), pennello) {
Dashed = true });
s.Markers.Add(new RSmithMarker("M1", gamma[50], pennello, true) { Frequency
= frequenze[50] });
s.MarkerMoved += Accorda;                          // void
Accorda(RSmithMarker m)
```



Quello che la carta disegna:

- **le griglie**: quella d'impedenza (r e x costanti) e, se si vuole, quella d'ammettenza, in rosa;
- **le curve**: una per parametro, con le frequenze ai due capi;
- **i cerchi dell'utente**: VSWR e Q costanti, e i cerchi di stabilità, guadagno e rumore. `Fill` colora l'interno di un cerchio, ad esempio la regione instabile;
- **i marcatori**: ognuno scrive etichetta, frequenza, Z e VSWR. Quello trascinabile segue il mouse e avvisa con `MarkerMoved`: è il modo di accordare a mano;
- **la lettura sotto il puntatore**: Γ, Z, Y, VSWR e perdita di ritorno.

Per muoversi: la rotella ingrandisce intorno al puntatore, un trascinamento sulla carta vuota sposta la vista, un doppio clic torna alla carta intera. Con `MaxGamma = 2` la carta si allarga oltre |Γ| = 1, dove la resistenza è negativa: serve per i dispositivi instabili e gli oscillatori.

Le formule stanno in `RSmithMath` (cartella `RF`), che non usa Avalonia:

- le conversioni: `Gamma`, `Impedance`, `Admittance`, `Vswr`, `ReturnLoss`;
- i cerchi della griglia;
- `OutputStability` e `InputStability`, con `OutputStableInside` per sapere da che parte si è stabili;
- `Rollett`, che dà K e |Δ|;
- `UnilateralGain` per i cerchi di guadagno e `Noise` per quelli di rumore.

Le prove le controllano con la loro definizione: sui punti di un cerchio di stabilità |Γin| vale 1, su quelli di un cerchio di rumore la cifra di rumore è quella chiesta.

21.2 Il grafico polare: RPolarChart

```
var p = new RPolarChart { Decibels = true, Range = 40, ZeroAngle = 90,
Clockwise = true };
p.Series.Add(RComplexSeries.Pattern("Piano E", gradi, guadagnoDb, true));
```

Il modulo è la distanza dal centro e la fase è l'angolo. Di base lo zero sta a destra e gli angoli crescono in senso antiorario, come nel piano complesso: va bene per S21 in modulo e fase. Un diagramma d'antenna si disegna invece con lo zero in alto e in senso orario (`ZeroAngle = 90`, `Clockwise`), e in dB (`Decibels`). In dB il bordo è il massimo, arrotondato ai 5 dB, e il centro sta `Range` dB più giù, così si vedono anche i lobi a −30 dB. `RComplexSeries.Pattern` costruisce la curva da angoli e valori.

21.3 Le barre e la mappa di colori: RBarChart, RHeatMap

```
var b = new RBarChart { TitleY = "Q a 2.45 GHz" };
b.Categories.AddRange(new[] { "C1", "C2", "L1" });
b.Series.Add(new RBarSeries("misurato", new[] { 410, 380, 62.0 }));

var m = new RHeatMap { TitleX = "x, mm", TitleY = "y, mm", Unit = "A/m",
Smooth = true };
m.SetValues(corrente, 0, 40, 0, 25);              // [riga, colonna], la
riga 0 in basso
```

Le barre di una categoria stanno affiancate, una per serie, oppure impilate con `Stacked`. Partono da zero, quindi i valori negativi scendono, e scrivono il loro valore in cima.

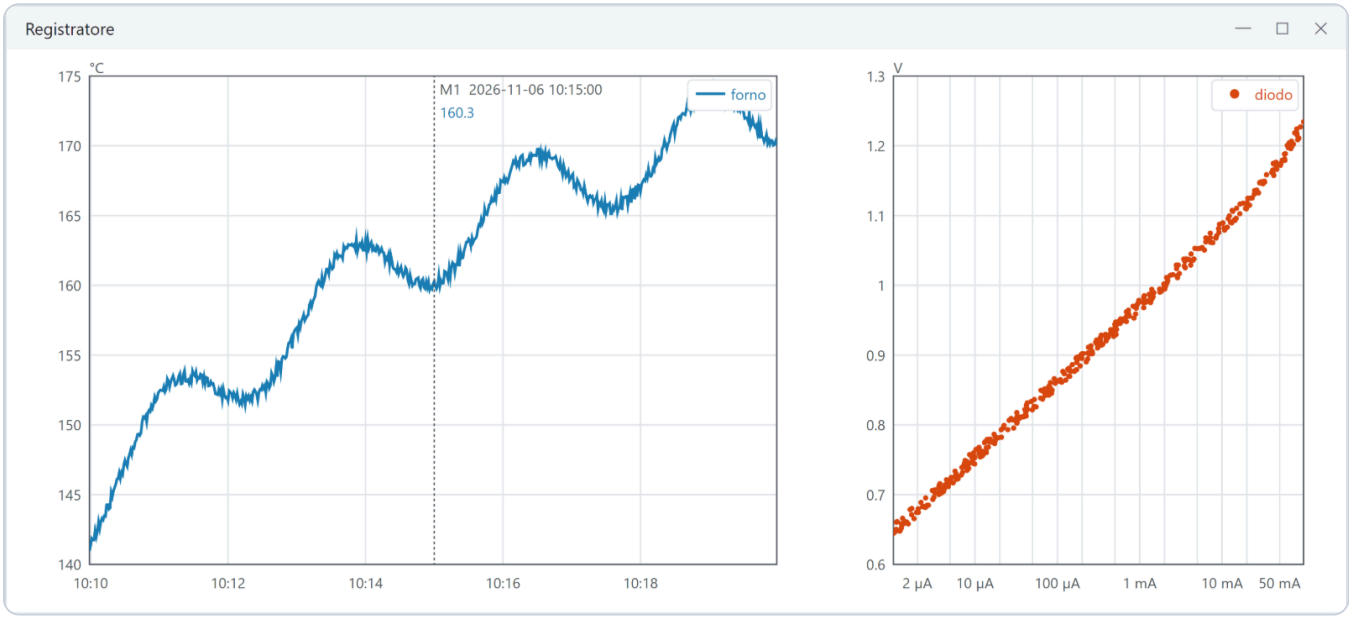
La mappa trasforma la matrice in un'immagine con un pixel per valore, poi la stira sull'area: a quadretti, o sfumata con `Smooth`. Le scale di colore sono quattro:

- `Viridis`, la predefinita: si legge bene anche stampata in grigio;
- `Heat`;
- `BlueRed`, per i valori intorno a un centro, come un errore o una fase;
- `Gray`.

A destra c'è la barra della scala, con l'unità. Il puntatore legge il valore sotto di sé (`ValueAt`), e i valori `NaN` restano trasparenti.

21.4 Le curve di RChart: punti, tempi, dati dal vivo, zoom

`RChart`, del capitolo 15, fa anche il plotter XY, il registratore e l'oscilloscopio lento.




```
var nuvola = new RSeries("diodo", corrente, tensione, 1) { Style =
RSeriesStyle.Points };

var c = new RChart { TimeAxis = true, TitleY = "°C" };
var forno = new RLiveSeries(c, "forno") { Window = 600 };     // gli ultimi
dieci minuti
forno.Add(DateTime.Now, lettura);                              // da qualunque
thread
```

L'esempio è `ESEMPI_RAVAL/Esempi/Registratore.cs`.

- **Punti e dispersione.** `Style` vale `Line`, `Points` o `LineAndPoints`. Le ascisse possono venire in qualunque ordine: un ciclo d'isteresi, una traiettoria o una nuvola di misure si disegnano come sono. Solo una curva con le ascisse crescenti (`Sorted`) viene letta dai

marcatori, perché lì a un'ascissa corrisponde un valore solo. `At` la legge per bisezione, quindi anche un milione di punti si legge subito.

- **L'asse dei tempi.** Con `TimeAxis` le ascisse sono tempi: secondi dall'anno 1, quelli che scrive `RTime.X(data)`. Le divisioni cadono su secondi, minuti, ore e giorni tondi, a partire dalla mezzanotte. Le etichette sono «10:15» o «10:15:30», e alla mezzanotte compare la data; un marcatore scrive data e ora per intero.
- **Dati dal vivo.** `RLiveSeries.Add` si chiama da qualunque thread, ad esempio quello della porta seriale. I punti aspettano in coda, e un solo passaggio al thread dell'interfaccia li prende tutti insieme, come fa la finestra del log: mille punti al secondo costano pochi ridisegni. `Window` tiene gli ultimi `Window` secondi (o unità d'ascissa) e il grafico li segue; `MaxPoints` mette un tetto ai punti tenuti.
- **Zoom e spostamento.** La rotella ingrandisce l'ascissa intorno al puntatore, con Maiusc l'ordinata. Un trascinamento sposta la vista e il doppio clic torna a tutti i dati, perché il clic fermo resta quello del marcatore. Mentre la vista è spostata, il registratore smette di seguire gli ultimi punti; da codice ci sono `ZoomX`, `ZoomY`, `SetView`, `ResetView` e i limiti fissi `MinX` e `MaxX`.
- **Etichette che non si toccano.** Un'etichetta d'ascissa che toccherebbe la precedente si salta, mentre la linea della griglia resta. Su un asse logaritmico stretto restano le decadi.

21.5 Perché sono fatti così

- **Una volta sola, bene.** La carta di Smith serve ad ASTRIP, agli strumenti di misura e ai libri: scritta una volta in RAVAL con le prove, non si corregge in tre posti.
- **Le formule separate dal disegno.** `RSmithMath` non conosce Avalonia: un progetto calcola i cerchi senza aprire finestre, e le prove li controllano senza disegnarli.
- **Il punto, non la virgola.** Come `RChart` (`Culture`): i numeri del grafico sono gli stessi della tabella e del CSV.

Esercizi

1. Trascinate il marcatore della carta e scrivete nella barra di stato l'impedenza letta in `MarkerMoved`.
2. Aggiungete il cerchio di rumore a 2 dB di un transistor con `RSmithMath.Noise` e il cerchio di guadagno con `UnilateralGain`, e cercate a mano il punto che li tocca tutti e due.
3. Disegnate in una `RHeatMap` con la scala `BlueRed` la differenza fra due misure, con `Min = -1` e `Max = 1`.

22. Gli inciampi di Avalonia 12

Cose che sono costate ore durante lo sviluppo di programmi veri e che non sono scritte nella documentazione di Avalonia. Sapendole prima, si risparmia il tempo.

Un metodo d'estensione può chiamarsi come una proprietà. Si legge spesso il contrario, e anche questo tutorial lo diceva: è falso. In una chiamata con le parentesi il compilatore scarta la proprietà, che non si può chiamare, e trova l'estensione: `c.Margin(8)` chiama `Margin(this T, double)` (misurato con Avalonia 12.1.4). Per questo le estensioni di RAva hanno i nomi delle proprietà che impostano (capitolo 12).

Dentro la classe delle estensioni, un metodo con il nome di un tipo nasconde il tipo. In `RAva` c'è il metodo `Border`, e allora `Border` scritto dentro `RAva` è il metodo, non la classe di Avalonia: `c is Border` dà l'errore CS0118. Lì dentro i tipi con lo stesso nome di un metodo si scrivono per intero (`Avalonia.Controls.Border`, `Avalonia.Controls.Grid`, `Avalonia.Media.FontWeight`). Fuori da `RAva`, nel codice di chi usa la libreria, il problema non c'è.

Non esiste `GetVisualRoot`. Per trovare la finestra di un controllo si usa `TopLevel.GetTopLevel(controllo)`, che restituisce la `Window` (o il `TopLevel` del browser o del telefono), oppure null se il controllo non è ancora in una finestra. Da lì si arriva a `StorageProvider`, `Clipboard`, `Screens`.

Un controllo che ha già un genitore non si rimette altrove. Ogni controllo ha un solo genitore visuale. Riusare lo stesso controllo in una riga nuova di un elenco o di una griglia (in un programma vero: un `CalendarDatePicker` riusato nella riga di una nuova attività) manda in crash il programma con un'eccezione sul genitore. Si crea un controllo nuovo ogni volta, oppure si toglie prima il controllo dal vecchio genitore.

Invio in una TextBox su più righe non arriva al KeyDown. Con `AcceptsReturn = true` la `TextBox` usa Invio per andare a capo e lo segna come gestito: un `KeyDown` normale non lo riceve mai. Si ascolta nella fase Tunnel, prima della casella: `AddHandler(InputElement.KeyDownEvent, Gestore, RoutingStrategies.Tunnel)` oppure `.OnEnter(Metodo)` (capitolo 6). Una TextBox di una riga invece Invio lo lascia passare, anche nella fase Bubble.

Sullo stesso controllo, i gestori della stessa fase vanno in ordine inverso. L'ultimo iscritto con `AddHandler` è chiamato per primo; se segna l'evento come gestito, quelli iscritti prima non lo ricevono, a meno di `handledEventsToo: true`. Chi aggiunge un gestore Tunnel a una casella che ha già `OnEnter` lo riceve; chi lo aveva aggiunto prima, no.

Una Bitmap usata da due Image non va liberata a mano. Se la stessa `Bitmap` è la `Source` di due controlli `Image` (una miniatura e un'anteprima) e la si libera con `Dispose` quando uno dei due sparisce, l'altro va in crash dentro `Image.Render` al ridisegno successivo. Le bitmap condivise si lasciano al garbage collector; si liberano a mano solo quelle che hanno un solo proprietario sicuro.

Senza schermo i menu non si vedono. Quando si disegnano le finestre senza schermo (per le figure o per i test), i menu e le tendine aprono un `Popup` in una finestra a parte, che non finisce nella cattura. Uno stile su tutti i `Popup` con `ShouldUseOverlayLayer = true` li fa disegnare **dentro** la finestra (appendice C).

Il testo su sfondo trasparente si sgrana. Disegnando un controllo con `RenderTargetBitmap` senza sfondo, il testo perde l'antialiasing e diventa illeggibile: mettete il controllo in un `Border` con sfondo pieno prima di fotografarlo.

Il tema cambia ma il controllo disegnato a mano no. `Render` legge i colori al momento del disegno, e Avalonia non lo richiama da sé quando cambia il tema: ci si iscrive ad `ActualThemeVariantChanged` e si chiama `InvalidateVisual` (capitolo 11).

Aspettare un metodo asincrono dal thread dell'interfaccia può bloccare per sempre. `MetodoAsync().Result` o `.GetAwaiter().GetResult()` nel thread dell'interfaccia si bloccano se dentro il metodo un `await` vuole tornare su quel thread (§2.9). È successo davvero con l'arresto di ASP.NET Core in `RServer.Stop()`: la cura è `Task.Run(FermaAsync).GetAwaiter().GetResult()`, che fa il lavoro sul pool.

Senza schermo i `DispatcherTimer` non scattano con `RunJobs()`. Nelle prove con `Avalonia.Headless`, `Dispatcher.UIThread.RunJobs()` esegue i lavori in coda ma non i timer: per provare un timer bisogna far girare il ciclo vero con `Dispatcher.UIThread.PushFrame(new DispatcherFrame())` e fermarlo da un altro thread (`TEST_RAVAL\Prove.cs`, classe `Attesa`).

A 300 dpi le scritte disegnate dopo un ritaglio spariscono. Una `RenderTargetBitmap` creata a 300 dpi perde ogni testo disegnato dopo la chiusura di un `PushClip`: il grafico salvato per la stampa non aveva né la legenda né i valori dei marcatori, che sullo schermo e a 96 dpi c'erano. La cura, in `RImage`, è creare la bitmap a 96 dpi ma già con i pixel della risoluzione voluta, e disegnare con la scala come trasformazione: stessi pixel, tutto disegnato. La risoluzione vera si scrive poi nel PNG (capitolo 15).

Una risorsa che manca toglie lo sfondo, e con lo sfondo il clic. Uno sfondo legato con `Resource` a una chiave che l'applicazione non ha resta nullo, e un `Border` senza sfondo non riceve il mouse: è trasparente anche al clic. Le righe di una tabella non si potevano scegliere in un'applicazione che chiamava i colori con nomi diversi. Un controllo che deve ricevere il clic ha sempre uno sfondo, anche solo `Brushes.Transparent`.

Appendici

A. Riferimento rapido

Voglio…	Scrivo
una finestra	`class MiaFinestra : Window` con `Title`, `Width`, `Height`, `Conte`
aprirla	`new MiaFinestra().Show()`; modale: `await f.ShowDialog<T>(proprietaria)`
chiuderla con un risultato	`Close(valore)`
dei figli in colonna (o in riga)	`RAva.VStack(8, a, b, c)`, `RAva.HStack(8, a, b)`
testo grande e in grassetto	`.Text("...").FontSize(18).Bold().Foreground("#1a7fb5")`
una cornice	`new Border().Border(1, radius: 8).Padding(16).Child(...)`
un modulo	`Grid` con `ColumnDefinitions("Auto,*")` e `.Row(r).Column(c)`
la cornice di un programma	`DockPanel` con `.Dock(Dock.Top)` … e l'ultimo figlio al centro
reagire a un clic	`b.Click += Metodo;` oppure `.OnClick(Metodo)`
Invio in una TextBox	`.OnEnter(Metodo)` (dietro: `AddHandler(InputElement.KeyDownEven ..., RoutingStrategies.Tunnel)`)
il testo che cambia	`.OnTextChanged(Metodo)` con `void Metodo(string testo)`
una spunta, una scelta	`.OnChecked(Metodo)`, `.OnSelectionChanged(Metodo)`
un colore che segue il tema	`new RThemes().Color("Chiave", light: ..., dark: ...).Apply(t` + `.Resource(Proprieta, "Chiave")`
un elenco di oggetti	`ListBox { ItemsSource = dati, ItemTemplate = new MioModello(`
la finestra di un controllo	`TopLevel.GetTopLevel(controllo)`
scegliere un file	`TopLevel.GetTopLevel(c)!.StorageProvider.OpenFilePickerAsync`

Voglio…	Scrivo
un lavoro lungo	`await Task.Run(Metodo)` e `Dispatcher.UIThread.Post(Mostra)` (capitolo 10)
un thread che lavora sempre	`new Thread(oggetto.Lavora) { IsBackground = true }.Start()` (
un dato condiviso fra thread	`lock (_chiave) { ... }` o `Interlocked.Increment(ref n)` (§2.4)
aspettare che un thread sia pronto	`ManualResetEventSlim`: `Set()` da una parte, `Wait()` dall'altra (§2.
un risultato da un altro thread, senza await	`Task<T> t = Task.Run(Metodo); T x = t.Result;` (§2.6)
due attese insieme	`await Task.WhenAll(a, b)` (§2.7)
fermare un lavoro	`CancellationTokenSource` + `token` passato alle attese (§2.11)
un server di API JSON	`new RServer(5080).Get("/api/x", Metodo).Start()` (capitolo 14)
scrivere nel log	`RLog.Start("App", "1.0", "log/app.log")`, poi `RLog.Info(...)`, `RLog.Error("...", ex)` (capitolo 16)
il log in una finestra	`new RLogView()`
un JSON o un XML da guardare	`RTree.FromJson(testo)`, `RTree.FromXml(testo)`; `SelectionChang` `Metodo` (capitolo 17)
leggere un JSON o un XML, senza finestra	`RNode.Load("f.json").GetInt("server.port", 5080)`; `Set`, `Save` (capitolo 17)
una domanda, una conferma, un errore	`await RDialog.Confirm(this, "Eliminare?", "Elimina")`, `Ask`, `Choose`, `Error(this, "...", e)` (capitolo 18)
scegliere un file, salvarlo	`string? f = await RFiles.OpenAsync(this, "Apri", "CSV
copiare una tabella in un foglio di calcolo	`await RClipboard.SetTable(this, righe)`; `GetTable`, `SetText`, `SetImage` (capitolo 19)

Voglio…	Scrivo
accettare i file trascinati sulla finestra	`RDrop.Files(controllo, Metodo, "*.csv")` (capitolo 19)
una tabella da centomila righe	`var t = new RVirtualTable(); t.LoadCsv(testo); t.Filter = "C` (capitolo 20)
le proprietà di un oggetto, modificabili	`new RPropertyList().Show(oggetto)`; `Changed`, `Rejected`, `Reloa` (capitolo 20)
un registratore dal vivo	`var s = new RLiveSeries(new RChart { TimeAxis = true }, "T") Window = 600 }; s.Add(DateTime.Now, v);` (capitolo 21)
una nuvola di punti XY	`new RSeries("nome", x, y) { Style = RSeriesStyle.Points }` (capitolo 21)
il tema chiaro/scuro/sistema, ricordato	`RThemes.Standard().Apply(app); RTheme.Remember(impostazioni) RTheme.Fill(menu);` (capitolo 16)
il programma in più lingue	`RLang.Open(cartella, "it"); new MenuItem().Tr("menu.file", "_File"); RLang.T("chiave", "testo")` (capitolo 16)
una carta di Smith	`new RSmithChart()` + `Series.Add(new RComplexSeries("S11", ga f))`; `RSmithMath` per i cerchi (capitolo 21)
un diagramma d'antenna	`new RPolarChart { Decibels = true, ZeroAngle = 90, Clockwise true }` + `RComplexSeries.Pattern` (capitolo 21)
barre, mappa di colori	`RBarChart` con `Categories` e `RBarSeries`; `RHeatMap.SetValues(matrice, x0, x1, y0, y1)` (capitolo 21)
le cartelle di un disco	`albero.Add(RTreeNode.FromFolder(percorso))`: si leggono quando aprono (capitolo 17)
un'impostazione	`RSettings.Open("app.json").GetInt("server.port", 5080)`; scriv `Set(chiave, valore)`
la finestra che riapre dov'era	`impostazioni.Remember(this, "window.main")` prima di `Show`

Voglio…	Scrivo
i file recenti	`new RRecentFiles(impostazioni, "files.recent")` + `Fill(menu)` `Add(percorso)`
qualcosa ogni secondo	`RAva.Every(TimeSpan.FromSeconds(1), Metodo)`, oppure `DispatcherTimer` con `Tick += Metodo`
un grafico a curve con i marcatori	`new RChart { UnitX = "Hz" }` + `Series.Add(new RSeries("nome" y))` (capitolo 15)
una tabella che si ordina	`new RTable().Columns("Nome", "Valore:R").AddRow("C1", "1.5p'`
valori da cambiare	`new RPropertyList().Add("R", "1k")` + `Changed += Metodo`
un numero con il prefisso SI	`RFormat.Si(1.296e9, "Hz")` → «1.296 GHz»; al contrario `RFormat.Parse("4k7")`
delle curve in CSV	`RCsv.Write("f (Hz)", grafico.Series)`; leggere: `RCsv.Read(test`
una figura per la stampa	`RImage.SavePng(controllo, "f.png", RImage.Dip(16), RImage.Dip(10), 300)`
l'aspetto di uno strumento	`RStyles.Clean().Apply(this)` + `RAva.Card`, `Caption`, `Bar`, `ToolbarButton`
disegnare	`class X : Control` con `override void Render(DrawingContext d`
ridisegnare	`InvalidateVisual()`

B. Glossario

- **Albero visuale**: i controlli di una finestra, ciascuno dentro il suo genitore.
- **API JSON**: un programma che risponde a richieste HTTP con dati in formato JSON (capitolo 14).
- **async / await**: `async` segna un metodo che può aspettare; `await` aspetta un Task senza fermare il thread, e il resto del metodo riprende dopo (§2.7).
- **Blocco mortale** (*deadlock*): due thread che si aspettano a vicenda e non vanno più avanti (§2.9).
- **CancellationToken**: la richiesta di fermarsi, passata a chi lavora (§2.11).

- **Contesto di sincronizzazione**: dice a `await` dove riprendere; quello del thread dell'interfaccia rimette il lavoro nella coda del Dispatcher (§2.8).
- **Corsa** (*race condition*): due thread che scrivono lo stesso dato e si perdono a vicenda delle modifiche (§2.4).
- **Caricatore**: il metodo che legge i figli di un nodo la prima volta che servono (capitolo 17).
- **Binding**: un legame fra una proprietà e una sorgente che la tiene aggiornata; `Resource` di RAva ne crea uno verso una risorsa del tema.
- **Delegato**: un metodo trattato come un valore (`Action`, `Func`, i gestori d'evento).
- **Dip**: l'unità delle misure di Avalonia, 1/96 di pollice; sullo schermo normale è un pixel.
- **Dispatcher**: la coda dei lavori del thread dell'interfaccia.
- **dpi**: punti per pollice; una figura stampata ne vuole almeno 300 alla sua misura sulla pagina (§15.6).
- **Evento instradato**: un evento che viaggia lungo l'albero (Tunnel verso l'interno, Bubble verso l'esterno).
- **Gruppo di metodi**: il nome di un metodo usato come delegato (`Click += Salva`).
- **Lambda**: una funzione senza nome (`x => x * 2`); in questo libro non si usa.
- **Log**: il diario del programma, una riga per ogni cosa che fa, con l'ora e il livello (capitolo 16).
- **Layout**: le due passate Measure e Arrange che decidono misura e posizione dei controlli.
- **lock**: un blocco di codice in cui entra un thread per volta (§2.4).
- **Metodo d'estensione**: un metodo statico con `this` sul primo parametro, che si chiama come se fosse del tipo esteso.
- **Modale**: una finestra che blocca la proprietaria finché non si chiude.
- **Pennello** (`IBrush`): come si riempie un'area. **Penna** (`Pen`): come si traccia una linea.
- **Proprietà di Avalonia** (`StyledProperty`): una proprietà che supporta stili, binding e risorse; il suo descrittore è il campo statico `NomeProperty`.
- **Risorsa**: un valore con un nome (colore, pennello) cercato risalendo l'albero fino all'applicazione.
- **Prefisso SI**: la lettera che moltiplica l'unità per una potenza di mille: p, n, µ, m, k, M, G (§15.1).
- **Pool di thread**: thread già pronti a cui .NET consegna i lavori brevi (`Task.Run`) (§2.6).
- **Task**: un lavoro in corso, o lo «scontrino» del suo risultato (`Task<T>`) (§2.6).
- **Thread**: una sequenza di istruzioni eseguita una dopo l'altra; più thread vanno avanti insieme (§2.3).
- **Thread dell'interfaccia**: l'unico thread da cui si possono toccare i controlli.
- **Virtualizzazione**: costruire solo le righe visibili di un elenco.

C. Come sono fatte le figure

Le figure si generano con il progetto `RAVAL\ESEMPI_RAVAL`:

```
dotnet run --project ESEMPI_RAVAL                      # tutte le figure in
RAVAL\DOC\immagini
dotnet run --project ESEMPI_RAVAL -- <cartella> 10_torta   # una sola
dotnet run --project ESEMPI_RAVAL -- --finestra        # il primo programma,
con la finestra vera
dotnet run --project ESEMPI_RAVAL -- --csharp          # gli esempi del
capitolo 1
dotnet run --project ESEMPI_RAVAL -- --thread          # gli esempi del
capitolo 2, con i tempi veri
```

Il programma usa `Avalonia.Headless` con il disegno vero di Skia: le finestre non compaiono sullo schermo ma si impaginano come se ci fossero. Poi `RImage.Render` ridisegna la finestra intera, menu aperti compresi, alla risoluzione dei libri: almeno 300 dpi e almeno 2400 pixel di larghezza. La scala è la più grande delle due, quindi una finestra piccola esce a più di 300 dpi. I dpi veri vengono scritti nel PNG. `CaptureRenderedFrame`, la fotografia della finestra, senza schermo dà solo i pixel della finestra, cioè 96 dpi: sulla carta le figure erano sgranate.

```
AppBuilder.Configure<AppEsempi>()
  .UseSkia()
  .UseHeadless(new AvaloniaHeadlessPlatformOptions { UseHeadlessDrawing =
false })
  .With(new FontManagerOptions { DefaultFamilyName = "Segoe UI" })
  .SetupWithoutStarting();
// menu e tendine disegnati dentro la finestra: senza schermo una finestra a
comparsa a parte non finirebbe nella foto
var sovrapposte = new Style(Finestrelle);
sovrapposte.Setters.Add(new Setter(Popup.ShouldUseOverlayLayerProperty,
true));
Application.Current!.Styles.Add(sovrapposte);
```

Ogni figura è una voce di `Catalogo.Tutti()`: il nome del PNG, la misura, il tema, il titolo della finestra finta disegnata attorno e il metodo che crea il contenuto. Per il menu aperto c'è anche un'azione da eseguire dopo che la finestra è apparsa (`IsSubMenuOpen = true`).