# Appunti Computazione Pervasiva

Corso di Computazione Pervasiva

**Sommario**

Questo documento presenta una panoramica completa sui principali argomenti del corso: Unified Modeling Language (UML) per la modellazione software, sviluppo di applicazioni Android, e i concetti fondamentali della computazione pervasiva. Include inoltre approfondimenti su design pattern, testing, threading, interfacce grafiche, linguaggi di programmazione, e tecnologie moderne come Firebase, MQTT e architetture a microservizi.

# Indice

# 1 Unified Modeling Language (UML)

## 1.1 Introduzione alla modellazione

La modellazione è un'attività fondamentale dell'ingegneria del software. Un **modello** è un'astrazione che cattura le proprietà salienti della realtà, idealizzando una realtà complessa e separando i tratti importanti dai dettagli, facilitando così la comprensione del sistema.

### 1.1.1 Perché modellare?

- Per comprendere e strutturare le entità da analizzare
- Per comunicare la conoscenza acquisita tra soggetti diversi
- Per gestire la complessità nei progetti software, che raramente coinvolgono un solo sviluppatore
- Per facilitare il passaggio di conoscenza nei team con ricambio di personale
- Per adattarsi ai cambiamenti delle caratteristiche del progetto nel tempo

## 1.2 UML: Cos'è e a cosa serve

UML (Unified Modeling Language) è un linguaggio semiformale e grafico, basato su diagrammi, per specificare, visualizzare, realizzare, modificare e documentare gli artefatti di un sistema software. Questi artefatti includono codice sorgente, eseguibili, documentazione, file di configurazione, tabelle di database e molto altro.

### 1.2.1 Caratteristiche di UML

- Indipendente dall'ambito del progetto
- Indipendente dal processo di sviluppo
- Indipendente dal linguaggio di programmazione
- È un vero linguaggio, non una semplice notazione grafica
- È semiformale, descritto in linguaggio naturale e con l'uso di diagrammi, eliminando le ambiguità

## 1.3 Paradigma Object-Oriented

UML è un linguaggio orientato alla modellazione object-oriented e supporta i concetti fondamentali del paradigma OO:

**Astrazione** Uso di classi per astrarre la natura e le caratteristiche di un oggetto

**Incapsulamento** Nascondere i dettagli implementativi di un oggetto, esponendo solo le interfacce necessarie

**Ereditarietà** Specializzazione di classi attraverso l'ereditarietà, implementando solo le differenze comportamentali

**Polimorfismo** Comportamenti diversi in risposta allo stesso messaggio, a seconda dell'oggetto che lo riceve

## 1.4 Diagrammi e viste in UML

UML definisce 13 tipi di diagrammi, suddivisi in due categorie principali:

### 1.4.1 Structure Diagrams (come è fatto il sistema)

- Class Diagram
- Object Diagram
- Package Diagram
- Composite Structure Diagram
- Component Diagram
- Deployment Diagram

### 1.4.2 Behavior Diagrams (come funziona il sistema)

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

Questi diagrammi supportano la struttura a viste **"4+1"**:

- **Logical View**: Decomposizione logica in classi, oggetti e relazioni
- **Development View**: Organizzazione in blocchi strutturali
- **Process View**: Processi o thread in esecuzione e loro interazioni
- **Physical View**: Installazione ed esecuzione fisica
- **Use Case View**: Vista che collega le altre, spiegando il funzionamento esterno

## 1.5 Class Diagram

Il Class Diagram è uno dei diagrammi più utilizzati in UML. Viene tipicamente impiegato per modellare:

- Il dominio del sistema
- Il glossario di un sistema
- Lo schema concettuale di un database
- L'architettura/struttura di un sistema software

### 1.5.1 Elementi del Class Diagram

**Classe** Descritta da un nome, attributi e operazioni:

```
Libro
+cod_libro: String
+titolo: String
#data_edizione: Date
-ISDN: String
+create()
#richiesta()
+restituzione()
```

Listing 1: Esempio di classe

**Visibilità**

- `+` public: qualsiasi altro classifier può usare l'elemento
- `#` protected: solo i discendenti possono usare l'elemento
- `-` private: solo il classifier stesso ha visibilità dell'elemento

**Molteplicità** Indica il numero di istanze di una classe o il numero di valori di un attributo:

```
NetworkController
consolePort [2..]: Port
```

**Generalizzazione/Specializzazione** Relazione tra classi dove gli oggetti della classe specializzata sono sostituibili a quelli della classe generalizzata.

**Relazioni**

- **Dipendenza**: Cambiamenti in una classe influenzano l'altra
- **Associazione**: Relazione strutturale tra classi
- **Aggregazione**: Relazione "tutto-parti" dove le parti possono esistere indipendentemente dal tutto
- **Composizione**: Relazione "tutto-parti" stretta, dove le parti hanno la stessa durata di vita del tutto

## 1.6 Sequence Diagram

Il Sequence Diagram evidenzia l'ordinamento temporale dello scambio di messaggi tra oggetti in uno scenario specifico. Include:

- **Oggetti partecipanti**: Rappresentati in cima al diagramma in colonne
- **Lifeline**: Linee tratteggiate che rappresentano l'esistenza di un oggetto nel tempo
- **Execution specification**: Rettangoli che indicano il periodo in cui un oggetto esegue un'azione
- **Messaggi**: Rappresentano la comunicazione tra oggetti, possono essere di vari tipi:
  - Call: invoca un'operazione
  - Return: restituisce un valore
  - Send: invia un segnale
  - Create: crea un oggetto
  - Destroy: distrugge un oggetto

## 1.7 Il Meta-modello UML

UML fa parte di un'architettura standardizzata chiamata MOF (Meta-Object Facility), che ha 4 livelli:

- **M0**: La realtà da modellare
- **M1**: I modelli che descrivono la realtà
- **M2**: I meta-modelli che descrivono i modelli (UML è qui)
- **M3**: I meta-meta-modelli che descrivono i meta-modelli (MOF è qui)

# 2 Design Pattern GoF

## 2.1 Introduzione ai Design Pattern

### 2.1.1 Origini e Storia

I design pattern traggono ispirazione dal lavoro dell'architetto Christopher Alexander che, alla fine degli anni '70, introdusse il concetto di "pattern" in architettura. Alexander identificava strutture progettuali ricorrenti di alta qualità e studiava come soluzioni simili venissero applicate a problemi analoghi.

Questo approccio fu portato nell'ingegneria del software nel 1987 da Kent Beck e Ward Cunningham, ma divenne popolare solo nel 1994 con la pubblicazione del libro "Design Patterns: Elements of Reusable Object-Oriented Software" scritto da quattro autori (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides), comunemente noti come la "Gang of Four" o GoF.

### 2.1.2 Definizione di Design Pattern

Un design pattern può essere definito come una **soluzione per un problema ricorrente in un determinato contesto e sistema di vincoli esterni**. Analizziamo questa definizione:

- **Soluzione**: indica lo scopo dei pattern, risolvere problemi tipici
- **Problema ricorrente**: si riferisce a situazioni che si presentano frequentemente
- **Contesto**: rappresenta le condizioni in cui un pattern è applicabile
- **Sistema di vincoli esterni**: l'insieme di limitazioni in uno specifico contesto

Un progettista esperto utilizza queste soluzioni consolidate senza dover "reinventare la ruota" ogni volta. Come dice M. Johnson: "I Design Patterns consentono di imparare dai successi altrui invece che dai propri fallimenti".

### 2.1.3 Elementi dei Design Pattern

Ogni design pattern ha quattro elementi fondamentali:

1. **Nome del Pattern**: identifica in una o due parole il problema, la sua soluzione e le conseguenze
2. **Problema**: descrive quando applicare il pattern
3. **Soluzione**: illustra gli elementi del progetto, relazioni e responsabilità
4. **Conseguenze**: i risultati e i compromessi nell'applicare il pattern

### 2.1.4 Documentazione dei Pattern

Il formato GoF prevede per ogni pattern:

- Nome e classificazione
- Intento
- Alias
- Motivazione
- Applicabilità
- Struttura
- Partecipanti
- Collaborazioni
- Conseguenze
- Implementazione/Esempi
- Usi conosciuti
- Pattern correlati

### 2.1.5 Classificazione dei Design Pattern

I 23 pattern nel catalogo GoF sono classificati secondo due criteri:

| **Purpose/Scope** | **Classe** | **Oggetto** |
|---|---|---|
| **Creazionali** | Factory Method | Abstract Factory, Builder, Prototype, Singleton |
| **Strutturali** | Adapter | Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy |
| **Comportamentali** | Interpreter, Template Method | Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor |

Tabella 1: Classificazione dei Design Pattern GoF

- **Purpose (Obiettivo)**:
  - Creazionali: per la creazione di oggetti
  - Strutturali: per organizzare classi e oggetti
  - Comportamentali: per gestire algoritmi e responsabilità
- **Scope (Contesto)**:
  - Classe: operano a livello di classe usando l'ereditarietà
  - Oggetto: operano a livello di oggetto usando la composizione

## 2.2 Pattern Creazionali: Singleton

### 2.2.1 Intento

Assicurarsi che una classe abbia un'unica istanza e fornire un punto di accesso globale.

### 2.2.2 Motivazione

In alcuni casi, dobbiamo garantire che esista una sola istanza di una classe (es. uno spooler di stampa) accessibile globalmente.

### 2.2.3 Applicabilità

Il Singleton va utilizzato quando:

- Deve esistere esattamente una singola istanza di una classe
- L'istanza deve essere accessibile da un punto ben definito
- L'istanza deve essere estensibile tramite sottoclassi

### 2.2.4 Struttura e Implementazione

Il pattern prevede:

1. Un costruttore privato o protetto
2. Un metodo statico che restituisce l'istanza unica
3. Un attributo statico che mantiene l'istanza

```
1  public class Singleton {
2      private static Singleton instance;
3
4      private Singleton() { }
5
6      public static Singleton getInstance() {
7          if (instance == null) {
8              instance = new Singleton();
9          }
10         return instance;
11     }
12 }
```

Listing 2: Implementazione Singleton

### 2.2.5 Conseguenze

- Accesso controllato all'istanza
- Riduzione del namespace
- Maggiore flessibilità rispetto agli attributi statici
- Possibilità di variare il numero di istanze

## 2.3 Pattern Strutturali: Adapter

### 2.3.1 Intento

Convertire l'interfaccia di una classe in un'altra interfaccia attesa dal client, permettendo a classi con interfacce incompatibili di collaborare.

### 2.3.2 Struttura

L'Adapter può essere implementato in due modi:

1. **Class Adapter**: usa l'ereditarietà multipla (dove supportata)
2. **Object Adapter**: usa la composizione di oggetti

### 2.3.3 Applicabilità

L'Adapter è utile quando:

1. Si vuole usare una classe esistente con interfaccia incompatibile
2. Si vuole creare una classe riusabile che collabora con classi impreviste
3. Si vuole riusare sottoclassi esistenti senza dover estendere ogni sottoclasse

### 2.3.4 Esempio

Nel caso di un editor grafico che usa una classe Shape, ma dispone di una classe TextView incompatibile, si può creare una classe TextShape che adatta l'interfaccia di TextView a quella di Shape.

```
class TextShape: public Shape, private TextView {
public:
    TextShape();
    virtual void BoundingBox(Point& bottomLeft, Point& topRight)
       const;
    virtual bool IsEmpty() const;
    virtual Manipulator CreateManipulator() const;
};

void TextShape::BoundingBox(Point& bottomLeft, Point& topRight)
   const {
    Coord bottom, left, width, height;
    GetOrigin(bottom, left);
    GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom+height, left+width);
}
```

Listing 3: Esempio di Adapter

### 2.3.5 Partecipanti

- **Target**: interfaccia usata dal client (es. Shape)
- **Client**: collabora con oggetti conformi all'interfaccia target
- **Adaptee**: interfaccia che deve essere adattata (es. TextView)
- **Adapter**: adatta l'interfaccia di Adaptee a Target (es. TextShape)

## 2.4 Pattern Comportamentali: Observer

### 2.4.1 Intento

Definire una relazione uno-a-molti tra oggetti, in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengano notificati e aggiornati automaticamente.

### 2.4.2 Alias

Dependant, Publish-Subscribe

### 2.4.3 Motivazione

Ad esempio, quando si modificano i dati in un foglio di calcolo, i grafici basati su quei dati si aggiornano automaticamente. L'interazione avviene tra un "subject" e un numero di "observer" dipendenti.

### 2.4.4 Applicabilità

L'Observer è utile quando:

- Un'astrazione ha due aspetti, uno dipendente dall'altro
- Un cambiamento a un oggetto richiede cambiamenti ad altri oggetti non noti a priori
- Un oggetto deve notificare altri oggetti senza fare assunzioni su di essi

### 2.4.5 Struttura

Il pattern prevede:

- **Subject**: mantiene l'elenco degli observer e li notifica dei cambiamenti
- **Observer**: definisce l'interfaccia per ricevere notifiche
- **ConcreteSubject**: implementa l'oggetto osservato
- **ConcreteObserver**: implementa l'aggiornamento in risposta alle notifiche

### 2.4.6 Conseguenze

- Accoppiamento astratto tra Subject e Observer
- Supporto alla comunicazione multicast
- Possibili aggiornamenti a cascata indesiderati

## 2.5 Pattern Strutturali: Composite

### 2.5.1 Intento

Comporre oggetti in strutture ad albero per rappresentare gerarchie tutto-parti, permettendo di trattare oggetti singoli e composizioni in maniera uniforme.

### 2.5.2 Motivazione

In un programma di disegno, dobbiamo gestire sia oggetti elementari (linee, testo) sia oggetti composti da elementi più semplici (figure, paragrafi).

### 2.5.3 Applicabilità

Il Composite è utile quando:

- Si rappresentano gerarchie tutto-parti
- Si vuole che i client possano ignorare le differenze tra insiemi di oggetti e oggetti individuali

### 2.5.4 Struttura

- **Component**: interfaccia comune a tutti gli oggetti
- **Leaf**: rappresenta oggetti primitivi che non hanno figli
- **Composite**: rappresenta oggetti composti che possono avere figli
- **Client**: manipola gli oggetti tramite l'interfaccia Component

### 2.5.5 Implementazione

```
Currency CompositeEquipment::NetPrice() {
    Iterator<Equipment> i = getIterator();
    Currency total = 0;
    for (i->First(); !i->IsDone(); i->Next())
        total += i->CurrentItem()->NetPrice();
    delete i;
    return total;
}
```

Listing 4: Implementazione Composite

### 2.5.6 Conseguenze

- Definisce gerarchie di oggetti primitivi e composti
- Semplifica il client
- Facilita l'aggiunta di nuovi tipi di componenti
- Può essere difficile imporre vincoli sui componenti

## 2.6 Conclusione sui Design Pattern

I design pattern rappresentano un catalogo di soluzioni consolidate a problemi ricorrenti di progettazione. La loro conoscenza permette ai progettisti di:

1. Risparmiare tempo evitando di reinventare soluzioni già note
2. Adottare un vocabolario comune che facilita la comunicazione
3. Aumentare la qualità del software attraverso architetture collaudate
4. Favorire il riuso e la manutenibilità del codice

# 3 Android e le Applicazioni

## 3.1 La piattaforma Android

Android è un complesso insieme di strati software per gestire tutti gli aspetti di un sistema mobile, includendo:

- Sistema Operativo
- Librerie
- Servizi intermedi (middleware)
- Applicazioni

Per sviluppare applicazioni Android si usa un Software Development Kit (SDK) che contiene gli strumenti necessari.

### 3.1.1 Processo di aggiornamento

Google gestisce il rilascio di Android, con un processo ben definito:

1. Il codice sorgente è rilasciato per i dispositivi prodotti da Google (Nexus e Pixel)
2. Per gli altri dispositivi, l'aggiornamento è gestito dai rispettivi produttori
3. Ogni produttore adatta Android ai suoi dispositivi seguendo un processo di qualificazione

### 3.1.2 Architettura di Android

L'architettura di Android è suddivisa in diversi strati:

1. **Kernel Linux**: Fornisce servizi di base come sicurezza, gestione della memoria, dei processi e dell'alimentazione
2. **Hardware Abstraction Layer (HAL)**: Fornisce interfacce standard che espongono le funzionalità hardware del dispositivo al framework API Java di livello superiore
3. **Android Runtime (ART)**: Include librerie di runtime che forniscono funzionalità del linguaggio di programmazione Java
4. **Librerie C/C++ native**: Utilizzate da vari componenti del sistema
5. **Framework API Java**: L'intero set di funzionalità del sistema operativo Android disponibili tramite API scritte in Java o Kotlin, che includono:
   - Sistema di visualizzazione
   - Resource Manager
   - Gestore delle notifiche
   - Activity Manager
   - Componenti per la generazione di contenuti
6. **Applicazioni di sistema**: Le app preinstallate che fungono sia da app per gli utenti sia da API per altre applicazioni

### 3.1.3 L'ambiente di sviluppo: Android Studio

Android Studio è l'IDE ufficiale per lo sviluppo di applicazioni Android, con componenti quali:

- Barra dei menu e degli strumenti
- Barra di navigazione
- Finestra dell'editor
- Barra di stato
- Finestra degli strumenti di progetto

## 3.2 Anatomia di un'Applicazione Android

Le applicazioni Android sono costituite da componenti e risorse:

- Android crea questi elementi dinamicamente in base alle necessità
- Ogni componente ha uno scopo specifico e una propria API (Application Programming Interface)

Le risorse predefinite includono:

- Layout (e le relative viste o views)
- Stringhe
- Colori
- Drawable (risorse grafiche)

I quattro componenti chiave di un'applicazione Android sono:

1. Activities
2. Services
3. Broadcast Receivers
4. Content Providers

### 3.2.1 Processo di Packaging di un'Applicazione

Per impostazione predefinita, ogni applicazione Android:

- Ha un ID utente Linux univoco
- Viene eseguita nel proprio processo Linux
- Può avere molteplici punti d'ingresso

Il sistema Android gestisce autonomamente:

- La creazione del processo quando è necessario eseguire codice dell'applicazione

- La terminazione del processo quando non è più necessario o quando altre applicazioni richiedono risorse

Un aspetto importante è il ruolo di **Zygote**, un processo speciale del sistema operativo che consente la condivisione del codice su Dalvik/ART VM. A differenza della Java VM tradizionale, dove ogni istanza ha una propria copia delle classi e librerie, Zygote ottimizza l'uso della memoria.

## 3.3 Componenti Fondamentali delle Applicazioni Android

### 3.3.1 Activity

L'Activity è il componente primario per l'interazione con l'utente:

- Rappresenta l'interfaccia utente
- Di solito corrisponde a una singola schermata
- Può contenere una o più viste (views)
- Estende la classe Java Activity

Il ciclo di vita di un'Activity è fondamentale per garantire un'esperienza utente fluida e una corretta gestione delle risorse. Il sistema Android non lascia alle applicazioni il controllo della durata dei propri processi; l'Android Runtime (ART) gestisce il processo di ciascuna applicazione e, di conseguenza, quello di ogni Activity al suo interno.

**Ciclo di vita delle Activity** Le principali callback del ciclo di vita sono:

- **onCreate()**: chiamato quando l'Activity viene creata
- **onStart()**: chiamato quando l'Activity diventa visibile all'utente
- **onResume()**: chiamato quando l'Activity inizia a interagire con l'utente
- **onPause()**: chiamato quando l'Activity non è più in primo piano
- **onStop()**: chiamato quando l'Activity non è più visibile
- **onRestart()**: chiamato quando l'Activity ritorna visibile dopo essere stata fermata
- **onDestroy()**: chiamato prima che l'Activity venga distrutta

**Gestione dello stack delle Activity** Le Activity sono organizzate in una pila (back stack) nell'ordine in cui vengono aperte. Nel comportamento standard:

- Quando l'Activity A avvia l'Activity B, A viene arrestata ma il sistema ne mantiene lo stato
- Se l'utente preme il tasto Indietro durante B, A riprende con il suo stato ripristinato
- Quando l'utente preme Home, l'Activity corrente passa in background mantenendo il suo stato

- Premendo il tasto Indietro, l'Activity corrente viene distrutta e quella precedente riprende

È importante notare che le Activity possono essere istanziate più volte, anche da altre Activity.

### 3.3.2 Service

Un Service è un componente che opera in background senza interfaccia utente:

- Esegue operazioni non interattive
- Estende la classe Java Service
- Continua a funzionare anche quando l'utente passa ad un'altra applicazione

### 3.3.3 Broadcast Receivers

I Broadcast Receivers sono componenti che ascoltano e reagiscono a eventi di sistema:

- Implementati mediante Intent specifici
- Non hanno interfaccia utente visiva
- Possono attivare una Activity in risposta a eventi

Esempi di eventi includono:

- Allarmi
- Arrivo di SMS
- Chiamate telefoniche
- Livello batteria basso

Dal punto di vista progettuale, questo sistema è basato sul pattern Observer con una tassonomia di eventi.

### 3.3.4 Content Providers

I Content Providers consentono lo scambio di dati tra applicazioni:

- Estendono la classe Java ContentProvider
- Forniscono un'interfaccia standardizzata per l'accesso ai dati

### 3.3.5 Intents

Gli Intents sono messaggi che connettono i componenti:

- Permettono la comunicazione tra componenti della stessa applicazione o di applicazioni diverse
- Possono essere espliciti (specificando il componente di destinazione) o impliciti (specificando un'azione)

## 3.4 Modelli di Navigazione nelle Applicazioni Android

La navigazione è un aspetto cruciale dell'esperienza utente (UX) nelle applicazioni Android. È importante ricordare che la navigazione rappresenta un "overhead" cognitivo per l'utente: muoversi all'interno di un'applicazione richiede concentrazione e tempo.

### 3.4.1 Principi di Navigazione Efficace

La navigazione deve aiutare l'utente a:

- Comprendere quali informazioni e strumenti sono disponibili
- Capire come sono strutturati contenuti e funzionalità
- Sapere dove si trova attualmente
- Capire dove può andare
- Ricordare da dove proviene e come tornare indietro

### 3.4.2 Pattern di Segnaletica

I pattern di segnaletica sono funzionalità che aiutano gli utenti a orientarsi, come:

- Titoli di pagine e finestre
- Indicatori di selezione
- Breadcrumb
- Barre di scorrimento annotate

### 3.4.3 Wayfinding

Il wayfinding consente agli utenti di trovare la strada verso il loro obiettivo all'interno dell'applicazione. Alcune regole che facilitano l'orientamento sono:

- **Buona segnaletica**: etichette chiare che indicano dove andare
- **Indizi ambientali**: convenzioni culturali (come la X in alto a destra per chiudere)
- **Mappe (anche visive)**: per fornire un'immagine mentale dell'intero spazio applicativo

### 3.4.4 Tipi di Navigazione

- **Navigazione globale**: accesso a tutte le sezioni principali dell'app
- **Navigazione di utilità**: accesso a funzioni trasversali (impostazioni, profilo)
- **Navigazione associativa e inline**: collegamenti contestuali
- **Contenuti correlati**: suggerimenti basati sul contesto attuale

### 3.4.5 Pattern di Navigazione

I pattern di navigazione affrontano diversi aspetti:

- **Clear Entry Points**: punti di ingresso chiari nell'applicazione
- **Menu Page**: pagina con opzioni di navigazione
- **Pyramid**: struttura gerarchica
- **Modal Panel**: pannelli modali per task specifici
- **Deep Links**: collegamenti diretti a contenuti profondi
- **Escape Hatch**: uscita di emergenza verso la home
- **Progress Indicator**: indicatore di avanzamento
- **Breadcrumbs**: percorso di navigazione

## 3.5 Layouts, Views e Widgets

### 3.5.1 Gerarchia delle Viste

Tutti i componenti visivi in Android discendono dalla classe View:

- Le viste sono spesso chiamate controlli o widget
- La classe ViewGroup è un'estensione di View che supporta viste innestate
- I ViewGroup che si concentrano sulla disposizione delle viste sono chiamati layout

### 3.5.2 Tipi di Layout

Android offre diversi tipi di layout per organizzare le viste:

- **LinearLayout**: organizza elementi in una singola riga o colonna
- **RelativeLayout**: posiziona elementi in relazione ad altri elementi
- **FrameLayout**: progettato per contenere un singolo elemento
- **ConstraintLayout**: posiziona elementi tramite vincoli relativi ad altri elementi

Il ConstraintLayout è particolarmente potente, offrendo massima flessibilità e una gerarchia di visualizzazione semplice. È disponibile come parte della libreria di supporto Android e deve essere incluso come dipendenza.

## 3.6 Processo di Sviluppo di un'Applicazione Android

Il processo di sviluppo di un'applicazione Android comprende diverse fasi:

### 3.6.1 Analisi e Progettazione

- Incontri con i committenti
- Comprensione dei bisogni degli utenti
- Identificazione delle funzionalità richieste
- Identificazione di vincoli e limiti

### 3.6.2 Design dell'Applicazione

- Definizione della struttura di navigazione
- Definizione della struttura dei contenuti di ogni vista

### 3.6.3 Implementazione

- Definizione delle risorse
- Progettazione e codifica dell'applicazione
- Generazione del package
- Installazione e lancio dell'applicazione

### 3.6.4 Testing

È essenziale testare l'applicazione a diversi livelli:

- **Test di unità**: convalidano il comportamento dell'app una classe alla volta
- **Test di integrazione**: convalidano le interazioni tra i livelli dello stack
- **Test di interfaccia**: test end-to-end che convalidano i percorsi utente

### 3.6.5 Validazione e Correzioni

- Verifica del comportamento dell'applicazione
- Identificazione dei problemi
- Ristrutturazione del codice
- Rigenerazione del package
- Generazione di report di accettazione

# 4 Interfacce Utente e Componenti in Android

## 4.1 Fondamenti dell'Interfaccia Utente Android

### 4.1.1 Gerarchia dei Componenti Visivi

La struttura dell'interfaccia utente in Android segue un modello gerarchico ben definito:

- **View**: È la classe base di tutti i componenti visivi in Android. Ogni elemento visibile nell'interfaccia utente discende da questa classe.
- **ViewGroup**: È una sottoclasse di View che può contenere altre View (chiamate "figlie"). Funge da contenitore e ha la responsabilità di:
  - Determinare le dimensioni di ogni view figlia
  - Controllare il posizionamento di ciascun elemento
  - Gestire la disposizione complessiva dell'interfaccia

I ViewGroup che si concentrano principalmente sulla disposizione degli elementi vengono chiamati "layout". È importante notare che, essendo sottoclassi di View, i ViewGroup possono anche disegnare la propria interfaccia personalizzata e gestire le interazioni utente.

### 4.1.2 Approcci per la Creazione dell'Interfaccia

In Android, esistono due approcci principali per costruire interfacce:

1. **Dichiarativo (XML)**: Consente di definire lo scheletro statico dell'interfaccia utente attraverso file XML. Questo metodo permette di:
   - Separare la presentazione dalla logica
   - Creare layout ottimizzati per diverse configurazioni hardware
   - Adattare l'interfaccia a diverse dimensioni dello schermo
2. **Programmatico (Java/Kotlin)**: Consente di modificare dinamicamente l'interfaccia durante l'esecuzione dell'applicazione.

La combinazione di questi due approcci rappresenta la soluzione ottimale per interfacce complete e adattabili.

## 4.2 Propagazione degli Eventi nell'Interfaccia Utente

Gli eventi di interazione utente, come i tocchi sullo schermo, seguono un percorso ben definito all'interno della gerarchia delle view:

### 4.2.1 Ciclo di Vita di un Evento Touch

1. L'evento inizia dall'Activity, che riceve la notifica tramite `dispatchTouchEvent()`
2. L'evento viene passato alla finestra (Window) attraverso `superDispatchTouchEvent()`
3. Da qui passa a DecorView (sottoclasse di FrameLayout)
4. Quindi raggiunge il layout principale dell'activity
5. Il ViewGroup notifica tutti i suoi figli dell'evento
6. L'evento arriva infine alla View specifica su cui è avvenuto il tocco

### 4.2.2 Gestione dell'Evento

La logica di processamento dell'evento segue questo ordine:

1. Se una View ha un OnTouchListener registrato, l'evento viene gestito dal metodo `onTouch()`
2. Altrimenti, viene gestito dal metodo `onTouchEvent()` della View
3. Se nessun componente gestisce l'evento, questo risale la gerarchia fino all'Activity
4. L'Activity ha l'ultima possibilità di gestire l'evento con il suo `onTouchEvent()`

Un ViewGroup può interrompere questo flusso restituendo true dal metodo `onInterceptTouchEvent` impedendo così la propagazione dell'evento alle view figlie.

## 4.3 ConstraintLayout: Il Layout Flessibile

### 4.3.1 Introduzione al ConstraintLayout

Il ConstraintLayout rappresenta uno dei layout più potenti e flessibili disponibili in Android. Offre:

- Massima flessibilità nella disposizione degli elementi
- Una gerarchia di view semplice senza eccessivo annidamento
- Un editor di layout visuale intuitivo

Per utilizzarlo è necessario includerlo come dipendenza nel file build.gradle:

```
dependencies {
    implementation "com.android.support.constraint:constraint-
       layout:1.1.2"
}
```

Listing 5: Dipendenza ConstraintLayout

### 4.3.2 Principi di Funzionamento

Il ConstraintLayout posiziona le view attraverso vincoli (constraints) che definiscono relazioni tra:

- Una view e i bordi del layout
- Una view e altre view
- Una view e linee guida personalizzate

Caratteristiche essenziali:

- Ogni view deve avere almeno un vincolo orizzontale e uno verticale
- L'editor avvisa in caso di vincoli mancanti
- È fondamentale evitare il posizionamento assoluto per garantire la compatibilità con diversi dispositivi

### 4.3.3 Tipi di Vincoli

- **Vincoli ai bordi del parent**
- **Vincoli di allineamento orizzontale e verticale**
- **Vincoli con offset**
- **Allineamento alla linea di base (baseline)**
- **Vincoli a linee guida**
- **Vincoli a barriere**

### 4.3.4 Modalità di Dimensionamento

- **Wrap Content**: La view si adatta alle dimensioni del suo contenuto
- **Fixed**: Dimensioni specificate in un valore fisso (da evitare quando possibile)
- **Match Constraint**: La view si espande per soddisfare i vincoli
- **Aspect Ratio**: Dimensioni specificate come rapporto (es. 16:9)

### 4.3.5 Catene di Vincoli (Chains)

Le catene permettono di gestire la distribuzione di più elementi collegati:

- **Spread**: Distribuzione equa dello spazio
- **Spread Inside**: Spazio distribuito solo tra gli elementi
- **Weighted**: Distribuzione proporzionale al peso assegnato
- **Packed**: Elementi raggruppati

## 4.4 Altri Componenti Fondamentali di Android

### 4.4.1 Service

- Componente eseguito in background
- Utilizzato per operazioni non interattive
- Non ha un'interfaccia utente visibile
- Ideale per operazioni di lunga durata che non richiedono interazione con l'utente

### 4.4.2 Broadcast Receivers

- Componenti che "ascoltano" annunci di sistema o custom
- Gli eventi sono implementati mediante Intent specifici
- Non hanno un'interfaccia utente visibile
- Esempi di eventi ascoltati:
  - Allarmi
  - Arrivo di SMS
  - Chiamate in entrata
  - Livello batteria basso

### 4.4.3 Content Providers

- Consentono di memorizzare e recuperare dati tra applicazioni diverse
- Utilizzano un'interfaccia simile a quella dei database
- Superano il modello di sicurezza di Android che normalmente isola i dati di ciascuna applicazione
- Esempio tipico: l'accesso ai contatti del dispositivo

## 4.5 Ciclo di Sviluppo delle Applicazioni Android

### 4.5.1 Progettazione

1. Comprendere i bisogni degli utenti
2. Identificare le funzionalità richieste
3. Definire vincoli e limiti
4. Progettare la struttura di navigazione
5. Definire il contenuto di ogni vista

### 4.5.2 Implementazione

1. Definire le risorse necessarie
2. Progettare e implementare l'applicazione
3. Generare il package dell'applicazione
4. Installare e lanciare l'applicazione

### 4.5.3 Testing

Il testing è fondamentale per garantire la qualità dell'applicazione. La "piramide dei test" illustra tre categorie principali:

1. **Test di Unità**: Validano il comportamento dell'app una classe alla volta
2. **Test di Integrazione**: Verificano le interazioni tra i vari livelli dell'applicazione
3. **Test di Interfaccia**: Test end-to-end che validano i percorsi utente attraverso più moduli

Android fornisce framework specifici per ciascun livello di testing, inclusi strumenti dedicati al testing dell'interfaccia utente.

# 5 Threading e Messaging in Android

## 5.1 Concetti di Threading

Un thread dal punto di vista concettuale è un'unità di computazione parallela all'interno di un processo, mentre dal punto di vista implementativo è rappresentato da un program counter e uno stack. Le aree heap e static sono condivise con altri thread.

In Java, un thread è rappresentato da un oggetto di tipo `java.lang.Thread` che implementa l'interfaccia Runnable con il metodo `run()`. I metodi principali includono:

- `void start()`: Avvia il thread
- `void sleep(long time)`: Sospende il thread per un periodo specificato
- `void wait()`: Il thread corrente attende fino a quando un altro thread invoca notify() sull'oggetto
- `void notify()`: Risveglia un singolo thread in attesa sull'oggetto

## 5.2 Il Thread UI in Android

Android ha una particolare architettura di threading:

- Ogni applicazione ha un thread principale (UI Thread)
- I componenti nella stessa applicazione condividono lo stesso UI Thread
- L'interazione dell'utente, le callback di sistema e i metodi del ciclo di vita vengono gestiti sul UI Thread
- Il toolkit UI non è thread-safe

Per evitare di bloccare l'UI Thread (che causerebbe un'applicazione non reattiva), le operazioni lunghe devono essere eseguite su thread in background. Tuttavia, è importante ricordare che non si deve accedere al toolkit UI da un thread non-UI.

## 5.3 Soluzioni per Background Thread

Android offre diverse soluzioni per eseguire lavoro in background e aggiornare l'UI:

1. **View.post e Activity.runOnUiThread**: Metodi garantiti per eseguire codice sul UI Thread
2. **AsyncTask**: Una classe che fornisce un modo strutturato per gestire lavoro su thread di background e UI
3. **Handler**: Permette di accodare ed elaborare messaggi e Runnable sulla coda di messaggi di un Thread

### 5.3.1 AsyncTask

AsyncTask è una classe generica con tre parametri di tipo:

- **Params**: Tipo usato nel lavoro in background
- **Progress**: Tipo usato per indicare l'avanzamento
- **Result**: Tipo del risultato

Metodi principali:

- `onPreExecute()`: Eseguito sul UI Thread prima dell'inizio del task
- `doInBackground(Params... params)`: Eseguito sul thread di background
- `publishProgress(Progress... values)`: Può essere chiamato da doInBackground
- `onProgressUpdate(Progress... values)`: Invocato in risposta a publishProgress() sul UI Thread
- `onPostExecute(Result result)`: Eseguito dopo doInBackground() sul UI Thread

Attenzioni da prestare con AsyncTask:

- Deve essere caricato sul UI Thread
- L'istanza deve essere creata sul UI Thread
- execute(Params...) deve essere invocato sul UI Thread
- Un task può essere eseguito una sola volta

### 5.3.2 Handler

Il Handler permette di accodare e processare Message e Runnable sulla coda di messaggi di un Thread. Ogni Handler è associato al Thread in cui è stato creato.

Ciascun thread Android è associato a una MessageQueue e un Looper. Il Looper dispatcha i messaggi chiamando il metodo `handleMessage()` dell'Handler sul thread dell'Handler stesso, mentre dispatcha i Runnable chiamando il loro metodo `run()`.

Metodi principali per accodare Runnable:

- `boolean post(Runnable r)`: Aggiunge un Runnable alla MessageQueue
- `boolean postAtTime(Runnable r, long uptimeMillis)`: Aggiunge un Runnable da eseguire a un tempo specifico
- `boolean postDelayed(Runnable r, long delayMillis)`: Aggiunge un Runnable da eseguire dopo un ritardo

Per i messaggi:

- `sendMessage()`: Accoda il messaggio immediatamente
- `sendMessageAtFrontOfQueue()`: Inserisce il messaggio all'inizio della coda
- `sendMessageAtTime()`: Accoda il messaggio a un tempo specifico
- `sendMessageDelayed()`: Accoda il messaggio dopo un ritardo

# 6 Lezione sullo Sviluppo Android: Intent, Room, LiveData e Architettura MVVM

Esploreremo alcuni concetti fondamentali per lo sviluppo di applicazioni Android moderne ed efficienti: Intent, Room, LiveData e l'architettura MVVM. Vedremo anche un esempio pratico di un'applicazione Note Taking.

## 6.1 Parte 1: Intent - Comunicazione tra Componenti

Gli Intent sono un meccanismo fondamentale di comunicazione tra i componenti di un'applicazione Android e tra diverse applicazioni. Possiamo considerarli come "messaggi" che permettono l'interazione tra:

- Activity
- Service
- Broadcast Receiver

### 6.1.1 Intent Espliciti

Gli Intent espliciti vengono utilizzati quando si conosce esattamente il componente da avviare. Ad esempio, per avviare una specifica Activity:

```
1 Intent intent = new Intent(MyActivity.this, MyOtherActivity.class
    );
2 startActivity(intent);
```

In questo caso, il sistema sa esattamente quale Activity avviare (MyOtherActivity) e non ha bisogno di fare alcuna risoluzione.

### 6.1.2 Intent Impliciti

Gli Intent impliciti vengono utilizzati quando non sappiamo esattamente quale componente dovrà gestire la nostra richiesta. Specifichiamo un'azione da eseguire e i dati su cui eseguirla:

```
1 Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel
    :555-2368"));
2 startActivity(intent);
```

Il sistema Android risolverà questo Intent trovando e avviando un'applicazione in grado di eseguire l'azione di chiamata telefonica. Questo è un esempio di "late binding a runtime".

### 6.1.3 Trasmissione di Dati tramite Intent

Gli Intent permettono di trasmettere dati tra componenti utilizzando il metodo `putExtra()`:

```
1 intent.putExtra("STRING_EXTRA", "Benevento");
2 intent.putExtra("INT_EXTRA", 82100);
```

### 6.1.4 Verifica della Risolvibilità di un Intent

È buona pratica verificare se un Intent implicito può essere gestito prima di utilizzarlo:

```
Intent intent = new Intent(Intent.ACTION_DIAL, Uri.parse("tel
    :555-2368"));
PackageManager pm = getPackageManager();
ComponentName cn = intent.resolveActivity(pm);
if (cn == null) {
     Log.e(TAG, "Intent could not resolve to an Activity.");
} else {
     startActivity(intent);
}
```

### 6.1.5 Sub-Activity e Risultati

Quando abbiamo bisogno di ricevere un risultato da un'Activity avviata, utilizziamo `startActivityForResult()`:

```
// Avvio di una sub-activity per ottenere un risultato
private static final int SHOW_SUBACTIVITY = 1;
private void startSubActivity() {
     Intent intent = new Intent(this, MyOtherActivity.class);
     startActivityForResult(intent, SHOW_SUBACTIVITY);
}
```

La sub-activity può restituire un risultato utilizzando `setResult()`:

```
// Restituzione di un risultato
setResult(RESULT_OK, resultIntent);
finish();
```

## 6.2 Parte 2: Architettura di Riferimento per Applicazioni Android

Le architetture moderne di applicazioni seguono principi di separazione delle preoccupazioni (separation of concerns) e indipendenza dai framework esterni. Nell'ambito Android, utilizziamo spesso l'architettura MVVM (Model-View-ViewModel).

### 6.2.1 Livelli dell'Architettura Clean

L'architettura clean è organizzata in cerchi concentrici, dove le dipendenze puntano sempre verso l'interno:

1. **Entities (Entità):** Regole di business di alto livello
2. **Use Cases (Casi d'Uso):** Regole di business specifiche dell'applicazione
3. **Interface Adapters:** Adattatori per convertire i dati tra formati interni ed esterni
4. **Frameworks e Drivers:** Framework esterni come database, UI frameworks

### 6.2.2 Principio di Inversione delle Dipendenze

Per mantenere la regola che le dipendenze puntino sempre verso l'interno, utilizziamo il principio di inversione delle dipendenze: i cerchi interni definiscono interfacce che i cerchi esterni implementano.

## 6.3 Parte 3: Room, LiveData e MVVM in Android

### 6.3.1 Room

Room è un ORM (Object-Relational Mapping) che facilita la mappatura tra classi Java e database SQLite. Con Room:

- Non è più necessario utilizzare Cursors e Loaders
- Le query al database sono più semplici e intuitive
- Room non permette di eseguire query sul thread principale per evitare ANR (Application Not Responding)

### 6.3.2 LiveData

LiveData è una classe osservabile che rispetta il ciclo di vita dei componenti Android:

- Permette di osservare cambiamenti nei dati attraverso più componenti dell'app
- Rispetta il ciclo di vita di Activities e Fragments
- Combinato con Room, permette di ricevere aggiornamenti automatici dal database

### 6.3.3 MVVM (Model-View-ViewModel)

MVVM è un pattern architetturale che separa la logica di presentazione dalla logica di business:

- **Model:** Rappresenta i dati e la logica di business
- **View:** Mostra i dati e invia le azioni dell'utente al ViewModel
- **ViewModel:** Media tra Model e View, trasforma i dati dal Model in un formato adatto alla View

## 6.4 Parte 4: Esempio Pratico - Note Taking App

Vediamo ora un esempio pratico di un'applicazione per prendere note che implementa Room, LiveData e MVVM.

### 6.4.1 1. Definizione dell'Entità Note

```
@Entity(tableName = "note_table")
public class Note {
    @PrimaryKey(autoGenerate = true)
    private int id;
    private String title;
    private String description;
    private int priority;

    // Constructor, getters and setters
}
```

### 6.4.2 2. Creazione del DAO (Data Access Object)

```
@Dao
public interface NoteDao {
    @Insert void insert(Note note);
    @Update void update(Note note);
    @Delete void delete(Note note);
    @Query("DELETE␣FROM␣note_table") void deleteAllNotes();
    @Query("SELECT␣␣FROM␣note_table␣ORDER␣BY␣priority␣DESC")
    LiveData<List<Note>> getAllNotes();
}
```

### 6.4.3 3. Configurazione del Database

```
@Database(entities = {Note.class}, version = 1)
public final abstract class NoteDatabase extends RoomDatabase {
    private static NoteDatabase instance;
    public abstract NoteDao noteDao();

    public static synchronized NoteDatabase getInstance(Context
       context){
        if (instance == null){
            instance = Room.databaseBuilder(context.
               getApplicationContext(),
                NoteDatabase.class, "sannio.notetaking")
                .fallbackToDestructiveMigration()
                .build();
        }
        return instance;
    }
}
```

### 6.4.4 4. Creazione del Repository

Il Repository fa da mediatore tra le fonti di dati (come il database) e il resto dell'applicazione:

```
public class NoteRepository {
    private NoteDao noteDao;
    private LiveData<List<Note>> allNotes;

    public NoteRepository(Application ctx){
        NoteDatabase db = NoteDatabase.getInstance(ctx);
        noteDao = db.noteDao();
        allNotes = noteDao.getAllNotes();
    }

    // Methods for data operations
    public void insert(Note note) {
        new InsertNoteAsyncTask(noteDao).execute(note);
    }

    public LiveData<List<Note>> getAllNotes() {
        return allNotes;
    }

    // Other CRUD operations
}
```

### 6.4.5 5. Implementazione del ViewModel

```
public class NoteViewModel extends AndroidViewModel {
    private NoteRepository repository;
    private LiveData<List<Note>> allNotes;

    public NoteViewModel(@NonNull Application application) {
        super(application);
        repository = new NoteRepository(application);
        allNotes = repository.getAllNotes();
    }

    // Methods delegated to repository
    public LiveData<List<Note>> getAllNotes() {
        return allNotes;
    }

    public void insert(Note note) {
        repository.insert(note);
    }
}
```

### 6.4.6 6. Configurazione della UI con RecyclerView

```
public class MainActivity extends AppCompatActivity {
    private NoteViewModel noteViewModel;

```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        RecyclerView recyclerView = findViewById(R.id.
            recycler_view);
        recyclerView.setLayoutManager(new LinearLayoutManager(
            this));
        final NoteAdapter adapter = new NoteAdapter();
        recyclerView.setAdapter(adapter);

        noteViewModel = new ViewModelProvider(this)
            .get(NoteViewModel.class);
        noteViewModel.getAllNotes().observe(this,
            new Observer<List<Note>>() {
                @Override
                public void onChanged(@Nullable List<Note> notes)
                    {
                    adapter.setNotes(notes);
                }
            });
    }
}
```

### 6.4.7 7. Implementazione dell'Adapter per RecyclerView

```
public class NoteAdapter extends RecyclerView.Adapter<NoteAdapter
    .NoteHolder> {
    private List<Note> notes = new ArrayList<>();

    @NonNull
    @Override
    public NoteHolder onCreateViewHolder(@NonNull ViewGroup
        parent, int viewType) {
        View itemView = LayoutInflater
            .from(parent.getContext())
            .inflate(R.layout.note_item, parent, false);
        return new NoteHolder(itemView);
    }

    @Override
    public void onBindViewHolder(@NonNull NoteHolder holder, int
        position) {
        Note currentNote = notes.get(position);
        holder.textViewTitle.setText(currentNote.getTitle());
        holder.textViewDescription.setText(currentNote.
            getDescription());
        holder.textViewPriority.setText(String.valueOf(
            currentNote.getPriority()));
    }
```

```
20
21      // Other adapter methods
22  }
```

## 6.5 Conclusioni

In questa lezione abbiamo esaminato:

1. Intent per la comunicazione tra componenti
2. Architettura Clean e MVVM per organizzare il codice in modo robusto e manutenibile
3. Room per la persistenza dei dati
4. LiveData per l'osservazione reattiva dei cambiamenti nei dati
5. Esempio pratico di un'applicazione Note Taking

Questi concetti sono fondamentali per lo sviluppo di applicazioni Android moderne, robuste e facilmente manutenibili. L'utilizzo di questi pattern architetturali permette di creare codice più testabile, meno accoppiato e più facile da evolvere nel tempo.

# 7 Mobile Development - Lezione: Testing, Threading e Interfacce Grafiche in Android

## 7.1 Parte 1: Testing Framework e Testing Mobile

Il testing è una fase essenziale nello sviluppo software, particolarmente critica in ambiente mobile dove le applicazioni vengono eseguite su una grande varietà di dispositivi. Analizziamo innanzitutto le tipologie di testing e i framework disponibili. I framework di unit test hanno attraversato diverse generazioni di evoluzione:

- **Prima generazione:** Utilizzavano semplici istruzioni assert, che però non fornivano messaggi di errore sufficientemente dettagliati.
- **Seconda generazione:** Hanno introdotto una famiglia di asserzioni specializzate come `assert_equal`, `assert_not_equal`, migliorando la chiarezza dei messaggi di errore.
- **Terza generazione:** Basati sull'approccio Hamcrest, utilizzano un operatore generico `assert_that` combinabile con "matcher" per una sintassi più espressiva e flessibile:

```
1 assert_that(x, equal_to(y))
2 assert_that(x, is_not(equal_ to(y)))
```

Nel testing Android, è importante distinguere tra test eseguibili sull'host di sviluppo e test che richiedono l'esecuzione sul dispositivo, quest'ultimi più complessi da implementare.

### 7.1.1 Mocking Framework

I framework di mocking come Mockito sono particolarmente utili per il testing in isolamento. Permettono di sostituire le dipendenze di un'unità sotto test (SUT) con "double" che simulano comportamenti specifici, rendendo i test più semplici e veloci. Mockito offre funzionalità come:

- Argument matchers
- Mocking di classi
- Verifica delle interazioni
- Spying su classi reali

Framework di riferimento includono JUnit 5, Mockito e Hamcrest.

## 7.2 Parte 2: Threading e Messaging in Android

Le applicazioni Android moderne richiedono operazioni in background per mantenere l'interfaccia utente reattiva. Analizziamo ora i concetti fondamentali del threading in Android.

### 7.2.1 Concetti di Threading

Un thread dal punto di vista concettuale è un'unità di computazione parallela all'interno di un processo, mentre dal punto di vista implementativo è rappresentato da un program counter e uno stack. Le aree heap e static sono condivise con altri thread.

In Java, un thread è rappresentato da un oggetto di tipo `java.lang.Thread` che implementa l'interfaccia `Runnable` con il metodo `run()`. I metodi principali includono:

- `void start()`: Avvia il thread
- `void sleep(long time)`: Sospende il thread per un periodo specificato
- `void wait()`: Il thread corrente attende fino a quando un altro thread invoca `notify()` sull'oggetto
- `void notify()`: Risveglia un singolo thread in attesa sull'oggetto

### 7.2.2 Il Thread UI in Android

Android ha una particolare architettura di threading:

- Ogni applicazione ha un thread principale (UI Thread)
- I componenti nella stessa applicazione condividono lo stesso UI Thread
- L'interazione dell'utente, le callback di sistema e i metodi del ciclo di vita vengono gestiti sul UI Thread
- Il toolkit UI non è thread-safe

Per evitare di bloccare l'UI Thread (che causerebbe un'applicazione non reattiva), le operazioni lunghe devono essere eseguite su thread in background. Tuttavia, è importante ricordare che non si deve accedere al toolkit UI da un thread non-UI.

### 7.2.3 Soluzioni per Background Thread

Android offre diverse soluzioni per eseguire lavoro in background e aggiornare l'UI:

1. `View.post` e `Activity.runOnUiThread`: Metodi garantiti per eseguire codice sul UI Thread
2. `AsyncTask`: Una classe che fornisce un modo strutturato per gestire lavoro su thread di background e UI
3. `Handler`: Permette di accodare ed elaborare messaggi e Runnable sulla coda di messaggi di un Thread

### 7.2.4 AsyncTask

`AsyncTask` è una classe generica con tre parametri di tipo:

- Params: Tipo usato nel lavoro in background
- Progress: Tipo usato per indicare l'avanzamento
- Result: Tipo del risultato

Metodi principali:

- `onPreExecute()`: Eseguito sul UI Thread prima dell'inizio del task
- `doInBackground(Params... params)`: Eseguito sul thread di background
- `publishProgress(Progress... values)`: Può essere chiamato da `doInBackground`
- `onProgressUpdate(Progress... values)`: Invocato in risposta a `publishProgress()` sul UI Thread
- `onPostExecute(Result result)`: Eseguito dopo `doInBackground()` sul UI Thread

Attenzioni da prestare con `AsyncTask`:

- Deve essere caricato sul UI Thread
- L'istanza deve essere creata sul UI Thread
- `execute(Params...)` deve essere invocato sul UI Thread
- Un task può essere eseguito una sola volta

‘‘‘

### 7.2.5 Handler

Il `Handler` permette di accodare e processare `Message` e `Runnable` sulla coda di messaggi di un Thread. Ogni `Handler` è associato al Thread in cui è stato creato.

Ciascun thread Android è associato a una `MessageQueue` e un `Looper`. Il `Looper` dispatcha i messaggi chiamando il metodo `handleMessage()` dell'`Handler` sul thread dell'`Handler` stesso, mentre dispatcha i `Runnable` chiamando il loro metodo `run()`.

Metodi principali per accodare `Runnable`:

- `boolean post(Runnable r)`: Aggiunge un `Runnable` alla `MessageQueue`
- `boolean postAtTime(Runnable r, long uptimeMillis)`: Aggiunge un `Runnable` da eseguire a un tempo specifico
- `boolean postDelayed(Runnable r, long delayMillis)`: Aggiunge un `Runnable` da eseguire dopo un ritardo

Per i messaggi:

- `sendMessage()`: Accoda il messaggio immediatamente
- `sendMessageAtFrontOfQueue()`: Inserisce il messaggio all'inizio della coda
- `sendMessageAtTime()`: Accoda il messaggio a un tempo specifico
- `sendMessageDelayed()`: Accoda il messaggio dopo un ritardo

## 7.3 Parte 3: Interfacce Grafiche Android e Custom Views

Le interfacce utente in Android sono composte da gerarchie di `View` e `ViewGroup`. Approfondiamo ora come creare view personalizzate e comprendere il pipeline di rendering.

### 7.3.1 Custom Views e Canvas

La creazione di view personalizzate permette di modellare radicalmente l'aspetto e il funzionamento delle applicazioni. Si può partire dalla classe base `View` o `SurfaceView`:

- `View`: Fornisce un oggetto `Canvas` con metodi di disegno e classi `Paint` per creare interfacce con bitmap e grafica raster
- `SurfaceView`: Fornisce un oggetto `Surface` che supporta il disegno da un thread in background e, opzionalmente, l'uso di OpenGL per grafica complessa

Per implementare una view personalizzata, è necessario sovrascrivere metodi come:

- `onMeasure(int widthMeasureSpec, int heightMeasureSpec)`: Chiamato quando il genitore sta disponendo i controlli figlio
- `onDraw(Canvas canvas)`: Per disegnare la view
- Handler di eventi come `onKeyDown`, `onKeyUp`, `onTouchEvent`: Per gestire l'interazione dell'utente

### 7.3.2 Pipeline di Rendering

Il processo di rendering in Android segue diverse fasi:

1. **Invalidation:** Il processo per segnalare che occorre ridisegnare determinati elementi
2. **Layout Traversal:** Effettua tutte le fasi del rendering per un singolo frame per tutte le view
    - Measure: Misura le dimensioni delle view
    - Layout: Posiziona le view
    - Draw: Disegna le view

Il componente `Choreographer` coordina queste operazioni per sincronizzarle con il refresh del display.

### 7.3.3 Fragments

I `Fragment` rappresentano un comportamento o una porzione di UI all'interno di un'`Activity`. Sono particolarmente utili per tablet e dispositivi con display più grandi, in quanto permettono di creare interfacce multi-pane.

Caratteristiche principali:

- Possono essere riutilizzati in diverse `Activity`
- Hanno un proprio ciclo di vita coordinato con quello dell'`Activity` contenitore
- Possono essere aggiunti, rimossi e sostituiti dinamicamente a runtime

### 7.3.4 Ciclo di vita dei Fragment

Il ciclo di vita di un `Fragment` è coordinato con quello della sua `Activity` contenitore, ma ha metodi specifici:

- `onAttach()`: Il `Fragment` viene collegato all'`Activity`
- `onCreate()`: Inizializzazione del `Fragment`
- `onCreateView()`: Configurazione dell'interfaccia utente
- `onActivityCreated()`: L'`Activity` contenitore ha completato `onCreate()`
- `onStart()`, `onResume()`: `Fragment` diventa visibile e attivo
- `onPause()`, `onStop()`: `Fragment` non è più visibile
- `onDestroyView()`: Distruzione dell'UI
- `onDestroy()`: Distruzione del `Fragment`
- `onDetach()`: `Fragment` è distaccato dall'`Activity`

Una caratteristica importante è che l'interfaccia utente di un `Fragment` viene inizializzata non in `onCreate()` ma in `onCreateView()`:

```
1 @Override
2 public View onCreateView(LayoutInflater inflater, ViewGroup
     container, Bundle savedInstanceState) {
3     return inflater.inflate(R.layout.my_fragment_layout,
         container, false);
4 }
```

### 7.3.5 Gestione dei Fragments

Le `FragmentTransaction` permettono di aggiungere, rimuovere o sostituire `Fragments` dinamicamente:

```
FragmentTransaction fragmentTransaction = fragmentManager.
    beginTransaction();
fragmentTransaction.add(R.id.container, new DetailFragment(), "
    detail_fragment");
fragmentTransaction.commitNow();
```

È possibile anche gestire uno stack di operazioni per consentire la navigazione all'indietro:

```
fragmentTransaction.addToBackStack(BACKSTACK_TAG);
fragmentTransaction.commit();
```

### 7.3.6 Comunicazione tra Fragment e Activity

Per la comunicazione tra `Fragment` e `Activity`, è buona pratica definire un'interfaccia di callback:

```
public interface OnItemSelectedListener {
    void onItemSelected(int position);
}
```

### 7.3.7 Frammenti senza UI

È possibile creare `Fragments` senza interfaccia utente per fornire comportamenti in background che persistono durante i riavvii dell'`Activity`:

```
public class WorkerFragment extends Fragment {
    public static final String MY_FRAGMENT_TAG = "worker_fragment
        ";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true); // Mantiene l'istanza durante i
            riavvii
    }
}
```

## 7.4 Conclusioni

In questa lezione abbiamo esplorato due temi fondamentali:

1. I concetti chiave dei linguaggi di programmazione moderni, con particolare attenzione a Java, Kotlin, JavaScript e Dart, analizzando variabili, tipi, classi e generics.

2. I `Fragments` in Android, componenti essenziali per creare interfacce utente modulari e flessibili, con attenzione al loro ciclo di vita e alla loro gestione.

# 8 Linguaggi di Programmazione e Frammenti in Android

## 8.1 Parte 1: Linguaggi di Programmazione - Concetti Fondamentali

### 8.1.1 Nomi, Variabili, Tipi e Binding

Un concetto cruciale in ogni linguaggio di programmazione è il **binding**, cioè l'associazione tra i nomi simbolici (o identificatori) e gli elementi referenziabili come:

- Variabili
- Procedure/Funzioni
- Tipi di dati

### 8.1.2 Type Binding

Esistono due approcci principali:

1. **Static Type Binding:** il tipo è specificato alla dichiarazione

```
int score;
score = 0;
```

2. **Dynamic Type Binding:** il tipo è associato al valore assegnato in fase di esecuzione

```
var x = 3;
x = 4.0; // cambia tipo dinamicamente
```

### 8.1.3 Storage Binding e Lifetime

La durata (**lifetime**) di una variabile è il tempo durante il quale essa è associata a un'area di memoria specifica. Possiamo distinguere:

1. **Variabili statiche con nome (named static):** allocate prima dell'esecuzione e presenti fino alla fine
2. **Dinamiche di stack (stack-dynamic):** create quando le loro dichiarazioni vengono elaborate
3. **Heap dinamiche esplicite (explicit heap-dynamic):** allocate/deallocate con istruzioni esplicite
4. **Heap dinamiche implicite (implicit heap-dynamic):** create automaticamente quando necessario

### 8.1.4 Scope

L'ambito (**scope**) di una variabile è l'intervallo di istruzioni in cui la variabile è visibile. L'ambito può essere:

- **Statico (lexical):** determinato prima dell'esecuzione in base alla struttura del codice
- **Dinamico:** determinato durante l'esecuzione

```
fun main() {
    val x = 1
    val y = 2
    if (x < y) {
        val z = x + y
        println(z) // stampa "3"
    } else {
        val z = x - y
        println(z) // stampa "-1"
    }
    // z non    accessibile qui
    // println(z) // Errore di compilazione
}
```

### 8.1.5 Classi, Oggetti e Programmazione a Oggetti

I linguaggi moderni condividono concetti fondamentali dell'OOP, ma con differenze importanti:

**Identity ed Equality** Un concetto fondamentale è la distinzione tra **identità** (stesso oggetto in memoria) ed **uguaglianza** (stesso valore):

- In JavaScript, i tipi primitivi sono confrontati per valore, mentre gli oggetti per riferimento
- Ogni linguaggio implementa in modo diverso i metodi per confrontare oggetti (`equals`, `===`, etc.)

**Costruttori e Constructor Chaining** I costruttori permettono di inizializzare gli oggetti. Il **constructor chaining** è il meccanismo per cui un costruttore può chiamare un altro costruttore della stessa classe.

**Genericità** La programmazione generica permette di parametrizzare i tipi di dati:

```
class Stack<T> {
    List<T> _stack = [];
    void push(T item) => _stack.add(item);
    T pop() => _stack.removeLast();
}
```

Un concetto avanzato legato alla genericità è la **covarianza** e **controvarianza**:

- **Covarianza:** se A è sottotipo di B, allora `List<A>` è sottotipo di `List<B>`
- **Controvarianza:** se A è sottotipo di B, allora `List<B>` è sottotipo di `List<A>`

Dart, come vediamo nel materiale, considera i tipi generici covarianti per default, ma permette di specificare esplicitamente la varianza con i modificatori `out`, `in` e `inout`.

**Riflessività** La **reflection** è la capacità di un programma di esaminare e modificare la propria struttura e comportamento a runtime:

- JavaScript ha forte capacità riflessiva con `eval()`
- Java è staticamente tipizzato ma la JVM permette modifiche dinamiche
- Dart offre vari metodi per la riflessione (`reflect()`, `ReflectClass()`, `ReflectType()`)

**Lambda Expressions** Le lambda expressions (o funzioni anonime) sono un modo conciso di rappresentare funzioni:

```
// JavaScript
const sum = (a, b) => a + b;
```

```
// Kotlin
val sum = { a: Int, b: Int -> a + b }
```

## 8.2 Parte 2: Fragments in Android

La seconda parte del materiale si concentra sui `Fragments` in Android, un concetto fondamentale per lo sviluppo di interfacce utente flessibili.

### 8.2.1 Cos'è un Fragment?

Un `Fragment` rappresenta una porzione di comportamento o interfaccia utente all'interno di un'`Activity`. I `Fragments` permettono di:

- Creare interfacce multi-pannello, particolarmente utili per tablet
- Riutilizzare componenti UI attraverso diverse `Activities`
- Gestire in modo modulare parti dell'interfaccia

### 8.2.2 Ciclo di Vita dei Fragments

Il ciclo di vita di un `Fragment` è coordinato con quello della sua `Activity` contenitore, ma ha metodi specifici:

1. `onAttach()`: `Fragment` è attaccato alla sua `Activity`
2. `onCreate()`: Inizializzazione del `Fragment`
3. `onCreateView()`: Creazione dell'interfaccia utente

4. `onActivityCreated()`: L'`Activity` contenitore ha completato `onCreate()`
5. `onStart()`, `onResume()`: `Fragment` diventa visibile e attivo
6. `onPause()`, `onStop()`: `Fragment` non è più visibile
7. `onDestroyView()`: Distruzione dell'UI
8. `onDestroy()`: Distruzione del `Fragment`
9. `onDetach()`: `Fragment` è distaccato dall'`Activity`

Una caratteristica importante è che l'interfaccia utente di un `Fragment` viene inizializzata non in `onCreate()` ma in `onCreateView()`:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup
    container,
                         Bundle savedInstanceState) {
    return inflater.inflate(R.layout.my_fragment_layout,
        container, false);
}
```

### 8.2.3 Gestione dei Fragments

Le `FragmentTransaction` permettono di aggiungere, rimuovere o sostituire `Fragments` dinamicamente:

```
FragmentTransaction fragmentTransaction = fragmentManager.
    beginTransaction();
fragmentTransaction.add(R.id.container, new DetailFragment(), "
    detail_fragment");
fragmentTransaction.commitNow();
```

È possibile anche gestire uno stack di operazioni per consentire la navigazione all'indietro:

```
fragmentTransaction.addToBackStack(BACKSTACK_TAG);
fragmentTransaction.commit();
```

### 8.2.4 Comunicazione tra Fragment e Activity

Per la comunicazione tra `Fragment` e `Activity`, è buona pratica definire un'interfaccia di callback:

```
public interface OnItemSelectedListener {
    void onItemSelected(int position);
}
```

Il `Fragment` definisce l'interfaccia e l'`Activity` la implementa:

```java
public class MyFragment extends Fragment {
    private OnItemSelectedListener listener;

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);
        if (context instanceof OnItemSelectedListener) {
            listener = (OnItemSelectedListener) context;
        } else {
            throw new RuntimeException(context.toString()
                    + " must implement OnItemSelectedListener");
        }
    }

    // Chiamata del callback quando necessario
    private void notifyItemSelected(int position) {
        if (listener != null) {
            listener.onItemSelected(position);
        }
    }
}
```

### 8.2.5 Frammenti senza UI

È possibile creare `Fragments` senza interfaccia utente per fornire comportamenti in background che persistono durante i riavvii dell'`Activity`:

```java
public class WorkerFragment extends Fragment {
    public static final String MY_FRAGMENT_TAG = "worker_fragment
        ";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true); // Mantiene l'istanza durante i
            riavvii
    }

    // Metodi per operazioni in background
    public void performBackgroundWork() {
        // Lavoro che deve persistere durante i cambi di
            configurazione
    }
}
```

Questo tipo di `Fragment` è utile per:

- Mantenere operazioni asincrone durante i cambi di configurazione
- Conservare dati temporanei senza doverli serializzare
- Gestire connessioni di rete o operazioni di lunga durata

### 8.2.6 Best Practices per i Fragments

**Comunicazione**

- Non comunicare direttamente tra `Fragments`
- Utilizzare sempre l'`Activity` come mediatore
- Definire interfacce chiare per la comunicazione

**Gestione del Ciclo di Vita**

- Prestare attenzione alle differenze tra il ciclo di vita del `Fragment` e dell'`Activity`
- Utilizzare `onViewCreated()` per operazioni che richiedono la view
- Pulire le risorse in `onDestroyView()`

**Transazioni**

- Utilizzare `commitNow()` quando è necessaria l'esecuzione immediata
- Gestire correttamente il back stack per una navigazione intuitiva
- Evitare transazioni annidate complesse

## 8.3 Conclusioni

In questa lezione abbiamo esplorato due temi fondamentali:

1. **I concetti chiave dei linguaggi di programmazione moderni,** con particolare attenzione a Java, Kotlin, JavaScript e Dart, analizzando:
   - Binding di variabili e tipi
   - Gestione della memoria e lifetime delle variabili
   - Scope e visibilità
   - Programmazione orientata agli oggetti
   - Genericità e varianza
   - Riflessività e lambda expressions
2. **I `Fragments` in Android,** componenti essenziali per creare interfacce utente modulari e flessibili, con attenzione a:
   - Ciclo di vita e coordinamento con le `Activities`
   - Gestione dinamica attraverso `FragmentTransaction`
   - Comunicazione tra componenti
   - `Fragments` senza UI per operazioni in background
   - Best practices per lo sviluppo

# 9 Content Providers

## 9.1 Introduzione ai Content Providers

I Content Providers sono componenti fondamentali di un'app Android. Forniscono un'interfaccia standardizzata per l'accesso ai dati e permettono la condivisione di informazioni tra diverse applicazioni in modo sicuro.

## 9.2 Quando utilizzare un Content Provider

Un Content Provider dovrebbe essere implementato nei seguenti casi:

- Quando si desidera offrire dati complessi o file ad altre applicazioni.
- Quando si vuole consentire ad altre app di fornire dati alla propria applicazione in modo uniforme e trasparente.
- Quando si desidera fornire suggerimenti di ricerca personalizzati utilizzando il Search Framework.
- Quando si vuole definire nella piattaforma una modalità di accesso standardizzata a determinati contenuti tramite URI.

## 9.3 Anatomia del Content Provider

L'architettura di un Content Provider prevede che le applicazioni client interagiscano con esso attraverso un Content Resolver. Questo schema consente un'astrazione efficace: l'applicazione client non necessita di conoscere i dettagli implementativi del provider, ma solo l'URI per accedere ai dati.

## 9.4 Creazione di un Content Provider

Per implementare un Content Provider è necessario seguire alcuni passaggi fondamentali:

1. Implementare un sistema di archiviazione per i dati: Questo può essere un database SQLite, file, o altre strutture di dati.
2. Determinare il formato del Content URI: L'URI è l'identificatore che altre applicazioni utilizzeranno per accedere ai contenuti.
3. Implementare il provider: Ciò comporta la creazione di una o più classi e la dichiarazione dell'elemento `<provider>` nel manifest.
4. Ereditare dalla classe ContentProvider: Questo richiede l'implementazione di metodi specifici che costituiscono l'interfaccia tra il provider e le altre app.

## 9.5 Content URI

Un elemento cruciale nella progettazione di un Content Provider è il Content URI, che identifica i dati nel provider. Un URI del contenuto è strutturato in diverse parti:

- **Authority**: Il nome simbolico qualificato del provider.
- **Path**: Un nome che punta a una tabella o un file.
- **ID (opzionale)**: Una parte che identifica un record specifico (es. una singola riga in una tabella).

Un esempio di URI potrebbe essere:

```
content://com.example.app.provider/table/42
```

## 9.6 Implementazione dei metodi del ContentProvider

La classe ContentProvider richiede l'implementazione di diversi metodi fondamentali:

- `query()`: Per recuperare dati dal provider.
- `insert()`: Per inserire nuovi dati.
- `update()`: Per aggiornare dati esistenti.
- `delete()`: Per eliminare dati.
- `getType()`: Per restituire il tipo MIME corrispondente a un Content URI.

## 9.7 Gestione dei permessi

La sicurezza è un aspetto cruciale dei Content Provider, dato che forniscono accesso ai dati dell'applicazione. I permessi devono essere dichiarati nel manifest, utilizzando gli attributi:

- `android:readPermission`: Per definire i permessi di lettura.
- `android:writePermission`: Per definire i permessi di scrittura.

Le applicazioni client dovranno richiedere questi permessi nel loro manifest per poter accedere ai dati del provider.

# 10 Broadcast Receivers

## 10.1 Introduzione ai Broadcast

I Broadcast sono messaggi inviati dal sistema Android o da altre applicazioni quando si verifica un evento di interesse. Sono incapsulati in oggetti Intent che contengono i dettagli dell'evento.

Possiamo distinguere due tipi principali di broadcast:

- **System Broadcast**: Inviati dal sistema Android.
- **Custom Broadcast**: Inviati dalle applicazioni.

## 10.2 System Broadcast

I System Broadcast sono messaggi generati dal sistema operativo Android quando si verificano eventi di sistema che potrebbero influenzare le applicazioni. Alcuni esempi includono:

- `ACTION_BOOT_COMPLETED`: Inviato quando il dispositivo completa l'avvio.
- `ACTION_POWER_CONNECTED`: Inviato quando il dispositivo viene collegato all'alimentazione esterna.

## 10.3 Custom Broadcast

I Custom Broadcast sono messaggi che la vostra applicazione può inviare, simili a quelli del sistema Android. Possono essere utilizzati, ad esempio, per informare altre applicazioni che alcuni dati sono stati scaricati e sono disponibili per l'uso.

## 10.4 Modalità di invio dei Broadcast

Android fornisce tre modalità principali per l'invio di broadcast:

1. **Ordered Broadcast**:
   - Viene consegnato a un ricevitore alla volta.
   - Si utilizza il metodo `sendOrderedBroadcast()`.
   - I ricevitori possono propagare il risultato al ricevitore successivo o interrompere la trasmissione.
   - L'ordine è controllato dall'attributo `android:priority` nel manifest.
   - Ricevitori con la stessa priorità vengono eseguiti in ordine arbitrario.
2. **Normal Broadcast**:
   - Viene consegnato a tutti i ricevitori registrati contemporaneamente, in un ordine non definito.
   - È il modo più efficiente per inviare un broadcast.
   - I ricevitori non possono propagare i risultati tra loro né interrompere la trasmissione.

- Si utilizza il metodo `sendBroadcast()`.

3. **Local Broadcast**:
   - Invia broadcast ai ricevitori all'interno della propria applicazione.
   - Non presenta problemi di sicurezza poiché non coinvolge comunicazione tra processi.
   - Si ottiene un'istanza di `LocalBroadcastManager` e si chiama il metodo `sendBroadcast()` su di essa.

## 10.5 Broadcast Receivers

I Broadcast Receivers sono componenti delle applicazioni che si registrano per vari broadcast di sistema o personalizzati. Vengono notificati (tramite un Intent) quando si verifica un evento per cui sono registrati.

## 10.6 Registrazione dei Broadcast Receivers

I Broadcast Receivers possono essere registrati in due modi:

1. **Static Receivers (Ricevitori statici)**:
   - Registrati nel file `AndroidManifest.xml`.
   - Chiamati anche Manifest-declared receivers.
   - A partire da Android 8.0 (API level 26), i ricevitori statici non possono ricevere la maggior parte dei broadcast di sistema.
2. **Dynamic Receivers (Ricevitori dinamici)**:
   - Registrati utilizzando il contesto dell'app o delle attività nei file Java.
   - Chiamati anche Context-registered receivers.
   - Necessari per ricevere la maggior parte dei broadcast di sistema a partire da Android 8.0.

## 10.7 Implementazione di un Broadcast Receiver

Per creare un Broadcast Receiver, è necessario:

1. Estendere la classe `BroadcastReceiver` e sovrascrivere il metodo `onReceive()`.
2. Registrare il ricevitore e specificare gli intent-filter:
   - Staticamente, nel Manifest.
   - Dinamicamente, con `registerReceiver()`.

## 10.8 Registrazione statica nel Manifest

```
<receiver
    android:name=".CustomReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

## 10.9 Registrazione dinamica nel codice

```
// Registrazione del ricevitore usando il contesto dell'attività
this.registerReceiver(mReceiver, filter);

// Deregistrazione del ricevitore
this.unregisterReceiver(mReceiver);
```

È importante ricordare di registrare i ricevitori dinamici in `onCreate()` o `onResume()` e di deregistrarli in `onDestroy()` o `onPause()` per evitare memory leak.

## 10.10 Registrazione di ricevitori locali

Per i broadcast locali, è possibile solo la registrazione dinamica:

```
// Registrazione di un ricevitore locale
LocalBroadcastManager.getInstance(this).registerReceiver(mReceiver,
    new IntentFilter(CustomReceiver.ACTION_CUSTOM_BROADCAST));

// Deregistrazione di un ricevitore locale
LocalBroadcastManager.getInstance(this).unregisterReceiver(mReceiver);
```

## 10.11 Restrizione dei Broadcast

È fortemente consigliato limitare i broadcast per motivi di sicurezza. Un broadcast non ristretto può rappresentare una minaccia alla sicurezza, poiché qualsiasi applicazione potrebbe registrarsi per riceverlo. Esistono diversi modi per limitare un broadcast:

- Utilizzo di `LocalBroadcastManager` per mantenere i dati all'interno dell'app.
- Utilizzo del metodo `setPackage()` per specificare il nome del pacchetto destinatario.
- Applicazione di permessi che possono essere imposti dal mittente o dal ricevitore.

## 10.12 Best Practices

1. Assicurarsi che il namespace per l'intent sia unico e di vostra proprietà.
2. Limitare i Broadcast Receivers.

3. Altre app possono rispondere ai broadcast inviati dalla vostra app — utilizzate i permessi per controllare questo comportamento.

4. Preferire i ricevitori dinamici rispetto a quelli statici.

5. Non eseguire mai operazioni a lunga durata all'interno di un Broadcast Receiver.

## 10.13 Conclusione

Content Providers e Broadcast Receivers sono componenti essenziali nell'ecosistema Android, che facilitano la comunicazione tra applicazioni e la condivisione dei dati. I Content Providers forniscono un'interfaccia strutturata per la condivisione dei dati, mentre i Broadcast Receivers permettono alle applicazioni di rispondere agli eventi di sistema o personalizzati. Comprendere il loro funzionamento e le best practices associate è fondamentale per sviluppare applicazioni Android robuste e sicure.

# 11 Notifiche in Android

## 11.1 Introduzione alle Notifiche

Le notifiche rappresentano uno dei principali canali di comunicazione tra un'app e l'utente quando l'applicazione non è attivamente in uso. Ogni notifica è tipicamente composta da tre elementi essenziali:

- Una piccola icona che identifica l'app.
- Un titolo che cattura l'attenzione dell'utente.
- Un testo dettagliato che fornisce informazioni aggiuntive.

Quando Android emette una notifica, questa appare inizialmente come un'icona nella barra di stato. L'utente può visualizzare i dettagli della notifica aprendo il cassetto delle notifiche (notification drawer), un'area dedicata che raccoglie tutte le notifiche attive e che può essere consultata in qualsiasi momento.

A partire da Android 8.0 (API level 26), le nuove notifiche vengono inoltre visualizzate come un "badge" colorato (noto anche come "notification dot") sull'icona dell'applicazione. Gli utenti possono tenere premuta l'icona dell'app per visualizzare rapidamente le notifiche relative a quell'applicazione, in modo simile al cassetto delle notifiche.

## 11.2 Canali di Notifica (Notification Channels)

Con Android 8.0 è stato introdotto il concetto di "canali di notifica", una caratteristica fondamentale che ha cambiato profondamente il modo in cui le notifiche vengono gestite. I canali di notifica permettono di creare categorie personalizzabili dall'utente per ogni tipo di notifica che la vostra applicazione deve visualizzare. Più notifiche possono essere raggruppate in un unico canale, e il comportamento della notifica (come suono, vibrazione, luci) viene applicato a tutte le notifiche all'interno di quel canale.

## 11.3 Importanza dei canali di notifica

È cruciale comprendere che a partire da Android 8.0, i canali di notifica sono obbligatori. Tutte le notifiche devono essere assegnate a un canale, altrimenti non verranno visualizzate. Per le applicazioni che hanno come target versioni precedenti ad Android 8.0, non è necessario implementare i canali di notifica. Nelle impostazioni del dispositivo, i canali di notifica appaiono come "Categorie" nelle impostazioni delle notifiche dell'app.

## 11.4 Creazione di un Canale di Notifica

Per creare un canale di notifica, è necessario istanziare un oggetto `NotificationChannel` utilizzando il relativo costruttore. Bisogna specificare:

- Un ID univoco all'interno del package dell'applicazione.
- Un nome del canale visibile all'utente.
- Il livello di importanza del canale.

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    NotificationChannel notificationChannel =
        new NotificationChannel(CHANNEL_ID, "Mascot Notification",
                                NotificationManager.IMPORTANCE_DEFAULT);
}
```

## 11.5 Livello di importanza (Importance Level)

Il livello di importanza, disponibile in Android 8.0 e versioni successive, determina il livello di intrusività della notifica, influenzando caratteristiche come suono e visibilità per tutte le notifiche pubblicate nel canale. I livelli di importanza variano da `IMPORTANCE_NONE` (0) a `IMPORTANCE_HIGH` (4). Per supportare versioni precedenti di Android (precedenti all'API level 26), è necessario impostare anche la priorità attraverso il metodo `setPriority()` per ogni notifica.

## 11.6 Creazione delle Notifiche

Una volta configurato il canale, possiamo procedere alla creazione della notifica vera e propria. Le notifiche vengono create utilizzando la classe `NotificationCompat.Builder`. Al costruttore vengono passati il contesto dell'applicazione e l'ID del canale di notifica:

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this, CHANNEL_ID)
```

## 11.7 Configurazione del contenuto della notifica

Per configurare il contenuto della notifica, utilizziamo diversi metodi della classe `NotificationCompat.B`

1. `setSmallIcon()`: imposta una piccola icona (unico contenuto obbligatorio).
2. `setContentTitle()`: imposta il titolo della notifica.
3. `setContentText()`: imposta il testo del corpo della notifica.

```
NotificationCompat.Builder mBuilder =
    new NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(R.drawable.android_icon)
    .setContentTitle("You've been notified!")
    .setContentText("This is your notification text.");
```

## 11.8 Azioni al Tocco e Pulsanti d'Azione

È fondamentale che ogni notifica risponda quando viene toccata, solitamente avviando un'Activity nell'app. Questo comportamento viene impostato utilizzando il metodo `setContentIntent()`, a cui viene passato un oggetto `PendingIntent` che incapsula l'Intent desiderato.

## 11.9 Notifiche con Vista Espansa

Le notifiche nel cassetto delle notifiche possono apparire in due layout principali: la vista normale (predefinita) e la vista espansa. Le notifiche con vista espansa sono state introdotte in Android 4.1 e dovrebbero essere utilizzate con parsimonia, poiché occupano più spazio e attirano maggiormente l'attenzione.

## 11.10 Conclusione

In questa lezione abbiamo esplorato come le notifiche in Android rappresentino un canale di comunicazione cruciale tra le applicazioni e gli utenti, e come i canali di notifica e le azioni al tocco possano migliorare l'esperienza utente.

# 12 Internet of Things: MQTT e Settings in Android

## 12.1 Parte 1: Protocolli dell'Internet of Things e MQTT

### 12.1.1 Introduzione all'Internet of Things

L'Internet of Things, o IoT, rappresenta un nuovo paradigma di connettività che permette a oggetti fisici di diventare "smart", capaci di raccogliere e scambiare dati attraverso reti, comunicando sia tra di loro che con server centrali.

In uno scenario IoT, le comunicazioni possono avvenire in diverse modalità:

- **Device to device**: gli oggetti smart comunicano direttamente tra loro.
- **Device to server**: i dispositivi raccolgono dati per inviarli a un server centrale.
- **Server to server**: i server si scambiano dati tra loro per distribuire le informazioni.

Per rendere possibili queste comunicazioni, sono stati sviluppati diversi protocolli specializzati:

- UPnP: fornisce supporto ai dispositivi in reti IP ad hoc.
- MQTT: specializzato nella raccolta di dati dai dispositivi per trasmetterli ai server.
- CoAP: ottimizzato per il trasferimento via Web di dati in reti con capacità limitate.
- XMPP: ideale per connettere dispositivi a persone tramite server.
- DDS: un bus veloce per integrare oggetti smart tra loro.
- AMQP: sistema per connettere server tra loro.

È importante comprendere che l'Internet of Things non è semplicemente un'estensione del Web. L'IoT rappresenta una nuova sfida con requisiti e problematiche specifiche, tra cui la necessità di protocolli efficienti per dispositivi con risorse limitate.

### 12.1.2 MQTT: Message Queuing Telemetry Transport

Tra questi protocolli, MQTT merita un'attenzione particolare per la sua rilevanza e diffusione crescente nell'ambito IoT.

**Storia e caratteristiche fondamentali** MQTT (Message Queuing Telemetry Transport) è stato sviluppato originariamente da IBM come protocollo leggero di messaggistica. Le sue caratteristiche principali lo rendono particolarmente adatto per ambienti con risorse limitate:

- È estremamente leggero e minimale.
- Richiede una banda minima per funzionare.
- Ottimizzato per reti con latenza elevata o instabili.
- Progettato per dispositivi con risorse computazionali limitate.
- Ideale per connessioni machine-to-machine (M2M).

**Il modello Publish/Subscribe** MQTT utilizza un modello di comunicazione chiamato "publish/subscribe" che disaccoppia il mittente dal destinatario:

- Un client può agire sia come publisher (pubblica messaggi) sia come subscriber (si iscrive per ricevere messaggi).
- Un broker funge da intermediario centrale, ricevendo tutti i messaggi e inoltrando quelli pertinenti agli subscribers.
- I messaggi vengono organizzati in topic, che funzionano come canali tematici.

Questo approccio offre numerosi vantaggi:

- Separazione tra produttori e consumatori di dati.
- Comunicazione molti-a-molti efficiente.
- I client non hanno bisogno di conoscere l'esistenza l'uno dell'altro.

**Comportamento dinamico del broker** Il broker MQTT gestisce dinamicamente le connessioni e le iscrizioni ai topic:

- Mantiene un registro di tutti i client connessi.
- Traccia le sottoscrizioni attive per ogni client.
- Gestisce l'inoltro dei messaggi in base alle sottoscrizioni.
- Si occupa della qualità del servizio richiesta.

**Comunicazione asincrona e bidirezionale** Un aspetto fondamentale di MQTT è la sua natura asincrona:

- I publisher possono inviare messaggi in qualsiasi momento.
- I subscriber ricevono messaggi quando questi vengono pubblicati.
- Non è necessario che tutti i dispositivi siano attivi contemporaneamente.
- La comunicazione può avvenire in entrambe le direzioni.

**Garanzia nella comunicazione: Quality of Service (QoS)** MQTT offre tre livelli di Quality of Service (QoS) per adattarsi a diverse esigenze di affidabilità:

- **QoS 0 - At most once**: il messaggio viene inviato una volta sola, senza conferma ("fire and forget").
- **QoS 1 - At least once**: il messaggio viene consegnato almeno una volta, con possibili duplicati.
- **QoS 2 - Exactly once**: il messaggio viene consegnato esattamente una volta, garantito.

**Sicurezza in MQTT** MQTT implementa la sicurezza su tre livelli:

- Autenticazione: tramite username/password o certificati SSL/TLS.
- Autorizzazione: controllo degli accessi ai topic per client specifici.
- Crittografia: supporto per TLS/SSL per proteggere i dati in transito.

**Prestazioni: MQTT vs HTTPs** Rispetto a HTTP, MQTT offre vantaggi significativi in contesti IoT:

- Header molto più compatti (2 byte vs 200 byte di HTTP).
- Minor consumo di banda.
- Minore latenza.
- Supporto nativo per comunicazioni push.
- Overhead ridotto per connessioni persistenti.

**MQTT-SN: evoluzione per sistemi embedded** Per sistemi ancora più vincolati in termini di risorse, esiste MQTT-SN (MQTT for Sensor Networks), una variante ottimizzata per reti wireless di sensori e dispositivi ultra-leggeri.

**Principali implementazioni** Esistono diverse implementazioni di broker MQTT disponibili:

- Mosquitto: broker open source leggero e ampiamente utilizzato.
- HiveMQ: soluzione enterprise scalabile.
- AWS IoT Core: servizio MQTT cloud di Amazon.
- Azure IoT Hub: piattaforma Microsoft con supporto MQTT.
- Eclipse Paho: un client MQTT multi-piattaforma disponibile per diverse lingue di programmazione.

**Casi d'uso di MQTT** MQTT viene utilizzato in numerosi ambiti:

- Monitoraggio remoto di sensori.
- Domotica e smart home.
- Telemetria industriale.
- Sistemi di notifica push.
- Applicazioni sanitarie e wearable.
- Smart cities.
- Automotive.

## 12.2 Parte 2: Settings in Android

### 12.2.1 Cosa sono i Settings?

I settings sono le impostazioni dell'applicazione che permettono agli utenti di personalizzare funzionalità e comportamenti dell'app. Alcuni esempi includono:

- La posizione di casa o le unità di misura predefinite.
- Il comportamento delle notifiche per l'app.
- Preferenze di visualizzazione o linguaggio.

I settings sono appropriati per valori che cambiano raramente e sono rilevanti per la maggior parte degli utenti. Se i valori cambiano frequentemente, è meglio utilizzare un menu opzioni o un navigation drawer.

### 12.2.2 Accesso alle impostazioni

Gli utenti accedono tipicamente alle impostazioni attraverso:

- Il navigation drawer.
- Il menu opzioni (in genere rappresentato dall'icona dei tre puntini).

### 12.2.3 Organizzazione delle schermate di impostazioni

Una buona organizzazione delle impostazioni è fondamentale per l'usabilità dell'app:

- Per 7 o meno impostazioni: organizzarle per priorità con le più importanti in cima.
- Per 7-15 impostazioni: raggruppare le impostazioni correlate sotto divisori di sezione.
- Per 16+ impostazioni: raggruppare in schermate separate accessibili dalla schermata principale delle impostazioni.

### 12.2.4 View vs Preference

Nel contesto delle impostazioni, Android utilizza oggetti Preference invece di oggetti View:

- Gli oggetti Preference sono specificamente progettati per le schermate di impostazioni.
- Possono essere progettati e modificati nell'editor di layout come gli oggetti View normali.
- Offrono funzionalità specifiche per la gestione delle impostazioni.

### 12.2.5 Definizione delle impostazioni in una Preference Screen

Le impostazioni vengono definite in una "preference screen", simile a un layout normale:

- Si definiscono in un file XML in `res > xml > preferences.xml`.
- Ogni impostazione deve avere una chiave (key) univoca.
- Android utilizza questa chiave per salvare il valore dell'impostazione.

### 12.2.6 Tipi di Preference

Android offre diversi tipi di Preference per vari tipi di impostazioni:

- `SwitchPreference`: Per opzioni attivabili/disattivabili.
- `EditTextPreference`: Per inserire testo.
- `ListPreference`: Per selezionare da un elenco di opzioni.

### 12.2.7 Implementazione dell'interfaccia utente dei Settings

L'interfaccia utente dei settings utilizza i Fragment:

- Si usa un'Activity con un Fragment per visualizzare la schermata delle impostazioni.
- Si utilizzano sottoclassi specializzate di Activity e Fragment che gestiscono il salvataggio delle impostazioni.

### 12.2.8 Implementazione dei Settings

Per implementare le impostazioni con `AppCompatActivity` e `PreferenceFragmentCompat`, seguire questi passaggi:

1. Creare la schermata delle preferenze (file XML).
2. Creare un'Activity per le impostazioni.
3. Creare un Fragment per le impostazioni.
4. Aggiungere il `preferenceTheme` all'AppTheme.
5. Aggiungere il codice per invocare l'interfaccia utente delle impostazioni.

### 12.2.9 Valori predefiniti delle impostazioni

È importante impostare valori predefiniti appropriati per le impostazioni:

- Scegliere valori che la maggior parte degli utenti selezionerebbe.
- Preferire opzioni che consumano meno batteria.
- Privilegiare la sicurezza e prevenire la perdita di dati.
- Interrompere l'utente solo quando è importante.

### 12.2.10 Gestione dei cambiamenti nelle impostazioni

Spesso è necessario reagire quando un'impostazione viene modificata, per:

- Mostrare impostazioni correlate di follow-up.
- Disabilitare o abilitare impostazioni correlate.
- Modificare il sommario per riflettere la scelta corrente.
- Agire sull'impostazione (ad esempio, cambiare lo sfondo dello schermo).

### 12.2.11 Conclusione

In questa lezione abbiamo esplorato due temi fondamentali per lo sviluppo di applicazioni moderne:

1. MQTT e l'Internet of Things: abbiamo visto come MQTT rappresenti uno dei protocolli più adatti per la comunicazione nell'IoT grazie alla sua leggerezza, efficienza e al modello publish/subscribe che ben si adatta ai requisiti dei dispositivi con risorse limitate.

2. Impostazioni in Android: abbiamo esaminato come implementare correttamente le impostazioni nelle applicazioni Android, utilizzando le classi Preference appropriate e gestendo valori predefiniti, salvataggio e reazione ai cambiamenti.

Entrambi questi argomenti sono essenziali per creare applicazioni moderne che interagiscano con l'ecosistema IoT e offrano un'esperienza utente personalizzabile e intuitiva.

# 13 Sviluppo di Applicazioni Mobile Multi-piattaforma con Flutter

## 13.1 Introduzione e Limitazioni di Flutter

Flutter è un framework open source sviluppato da Google che permette di creare applicazioni native cross-platform con un unico codebase. Tuttavia, come ogni tecnologia, presenta alcune limitazioni che è importante conoscere:

- **Gestione dei pacchetti**: Flutter introduce un proprio package manager, il che può rappresentare un ulteriore strumento da imparare e gestire rispetto agli ecosistemi nativi.
- **Supporto JSON**: Il supporto per JSON in Flutter manca di alcune flessibilità presenti in altri framework, il che può complicare la gestione di strutture dati complesse.

## 13.2 Sistema di Navigazione in Flutter

La navigazione è un componente fondamentale di qualsiasi applicazione mobile. Flutter gestisce la navigazione tramite la classe `Navigator`, che implementa un pattern di navigazione a stack:

- La classe `Navigator` (versione 1.0) permette di gestire le schermate come uno stack, dove è possibile aggiungere (push) o rimuovere (pop) pagine.
- Questo sistema è concettualmente equivalente al metodo `startActivityForResult` di Android, permettendo non solo di navigare tra schermate ma anche di restituire risultati.

### 13.2.1 Esempio di navigazione tra schermate

- Da una schermata principale `TodosScreen` si può navigare a una schermata di dettaglio `DetailScreen`.
- Al ritorno dalla schermata di dettaglio, è possibile passare dati alla schermata principale.

La navigazione si implementa con comandi come:

```
// Navigare verso una nuova schermata
Navigator.push(context, MaterialPageRoute(builder: (context) => DetailScreen()));

// Tornare indietro
Navigator.pop(context, [risultato opzionale]);
```

## 13.3 Gestione delle Risorse e Immagini

Flutter offre un sistema robusto per la gestione delle risorse come immagini, font e altri asset:

- La documentazione ufficiale fornisce linee guida dettagliate su come importare e utilizzare le risorse nell'applicazione.
- È importante definire correttamente le risorse nel file `pubspec.yaml` per permettere a Flutter di gestirle efficacemente.

## 13.4 Utilizzo della Fotocamera

Flutter permette di accedere alle funzionalità hardware dei dispositivi, come la fotocamera:

- Si utilizza `CameraPreview` per visualizzare il feed della fotocamera nell'interfaccia utente.
- `CameraController` consente di controllare la fotocamera e scattare foto.
- La documentazione ufficiale offre esempi completi su come implementare queste funzionalità.

## 13.5 Platform Channels: Comunicazione con Codice Nativo

Una delle caratteristiche più potenti di Flutter è la capacità di comunicare con il codice nativo della piattaforma attraverso i "Platform Channels":

```
// Codice Flutter
static const platform = MethodChannel('samples.flutter.io/battery');

Future<Null> _getBatteryLevel() async {
  String batteryLevel;
  try {
    final int result = await platform.invokeMethod('getBatteryLevel');
    batteryLevel = 'Battery level at $result % .';
  } on PlatformException catch (e) {
    batteryLevel = "Failed to get battery level: '${e.message}'.";
  }
  setState(() {
    _batteryLevel = batteryLevel;
  });
}
```

Questo codice Flutter comunica con l'implementazione specifica della piattaforma.

### 13.5.1 Implementazione Android (Kotlin)

```
private fun getBatteryLevel(): Int {
  val batteryLevel: Int
  if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) {
    val batteryManager = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
    batteryLevel = batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPAC
  } else {
    val intent = ContextWrapper(applicationContext).registerReceiver(null, IntentFilte
```

```
    batteryLevel = intent!!.getIntExtra(BatteryManager.EXTRA_LEVEL, -1)  100 / intent
  }
  return batteryLevel
}
```

Nell'esempio mostrato, vediamo come Flutter permetta di accedere a una funzionalità nativa (la lettura del livello della batteria) utilizzando un canale di comunicazione tra Flutter e il codice nativo. Questo approccio è estremamente potente perché consente di utilizzare qualsiasi API nativa della piattaforma mentre si mantiene un'unica base di codice per la logica dell'applicazione.

## 13.6 Supporto per Sviluppatori Android

Flutter offre risorse specifiche per facilitare la transizione degli sviluppatori Android, rendendo più semplice applicare concetti già familiari al nuovo framework.

## 13.7 Conclusione

Flutter rappresenta una soluzione efficace per lo sviluppo di applicazioni cross-platform, combinando i vantaggi di un'unica base di codice con la possibilità di accedere alle funzionalità native delle diverse piattaforme. Nonostante alcune limitazioni, offre un ecosistema ricco e ben documentato che permette di creare applicazioni performanti e dall'aspetto nativo.

# 14 Autenticazione e Autorizzazione nei Sistemi Informatici

## 14.1 Definizioni Fondamentali

### 14.1.1 Autenticazione vs Autorizzazione

- **Autenticazione**: è il processo di verifica dell'identità di un utente - confermare che qualcuno è effettivamente chi dichiara di essere.
  - In pratica: login + password (chi sei).
- **Autorizzazione**: riguarda le regole che determinano chi può fare cosa - quali azioni un utente autenticato può eseguire.
  - In pratica: permessi (cosa puoi fare).

## 14.2 Autenticazione: Restrizione degli Accessi

Spesso è necessario limitare l'accesso a determinate risorse web solo a utenti specifici. Un utente viene identificato attraverso un nome utente e una password. Esistono vari meccanismi per controllare l'accesso alle pagine web.

### 14.2.1 Schema di Autenticazione Base (HTTP Basic Authentication)

Questo è un meccanismo basilare fornito dal protocollo HTTP:

- Per ogni URL che il server desidera proteggere, viene mantenuta una lista di utenti autorizzati.
- Utilizzando gli header HTTP, il server dichiara che una pagina richiesta è riservata (richiede autenticazione).
- Il client passa nome utente e password all'interno di un header HTTP.
- La decisione su quali pagine sono riservate e a quali utenti è implementata dal server (non fa parte del protocollo HTTP).

### 14.2.2 Funzionamento del Meccanismo HTTP Basic

1. Nome utente e password devono essere inviati con ogni richiesta di una risorsa protetta.
2. Quando il server riceve una richiesta per una risorsa protetta, verifica se contiene l'header HTTP:

   ```
   Authorization: Basic username:password
   ```

   (dove username:password viene codificato per permettere caratteri speciali).
3. Se le credenziali sono accettate, la risorsa richiesta viene restituita.

4. Se la richiesta non contiene l'header di autorizzazione o le credenziali non sono accettate, il server risponde con codice 401 (Unauthorized).

5. Una risposta 401 può includere l'header:

   ```
   WWW-Authenticate: Basic realm="realm-name"
   ```

   che indica "per ottenere questa risorsa, devi autenticarti usando il metodo basic".

### 14.2.3 Realm di Autenticazione

- Un "realm" definisce uno spazio di autenticazione distinto.
- Diversi realm possono proteggere diverse parti di un sito web.
- Il browser memorizza le credenziali per un realm e le invia automaticamente quando:
  - La risorsa richiesta si trova nella stessa directory della risorsa originariamente autenticata.
  - Il browser riceve un 401 dal server web e l'header `WWW-Authenticate` ha lo stesso realm della precedente risorsa protetta.

### 14.2.4 Autenticazione Digest (Digest Authentication)

L'autenticazione digest è un'alternativa all'autenticazione basic che risolve alcune delle sue vulnerabilità:

- **Vantaggi dell'autenticazione digest**:
  - Non invia mai password in chiaro attraverso la rete.
  - Previene attacchi di tipo replay (riproduzione di autenticazioni intercettate).
  - Può opzionalmente proteggere dall'alterazione dei contenuti.
  - Protegge da molte altre forme comuni di attacco.
- **Limitazioni dell'autenticazione digest**:
  - Non fornisce un meccanismo di autenticazione forte come quelli basati su chiave pubblica.
  - Non offre protezione della riservatezza oltre alla protezione della password.
  - Il resto della richiesta e della risposta rimane disponibile per gli intercettatori.

### 14.2.5 Utilizzo di Nonce per Prevenire Attacchi Replay

Anche se i digest unidirezionali evitano di inviare password in chiaro, un attaccante potrebbe catturare il digest e usarlo ripetutamente. Per prevenire questo:

1. Il server passa al client un token speciale chiamato "nonce", che cambia frequentemente.

2. Il client aggiunge questo nonce alla password prima di calcolare il digest.

3. Mescolando il nonce con la password, il digest cambia ogni volta che cambia il nonce.
4. Ciò previene attacchi replay, poiché il digest è valido solo per un particolare valore nonce.

### 14.2.6 Esempio di Autenticazione Digest con JSON

```
secret = acquireSharedSecret(...)
JSONRequest request = createRequestWithDigest(secret)
{
  'attribute1': 'content',
  'attribute2': 'content'
}
```

Viene poi calcolato e iniettato un digest SHA-256:

```
{
  'attribute1': 'content',
  'attribute2': 'content',
  'digest': 'd2345d0a05c2f2801889543050826ab3a58fbfd...'
}
```

Il server a sua volta calcola il digest usando il contenuto e il segreto condiviso, poi confronta i valori per verificare l'autenticità.

## 14.3 Autorizzazione: Controllo degli Accessi

Una volta autenticato un utente, è necessario determinare a quali risorse può accedere e quali operazioni può eseguire. Esistono diversi modelli per gestire l'autorizzazione:

- **Matrice di Controllo degli Accessi (Access Control Matrix)**: Caratterizza i diritti di ogni soggetto rispetto a ogni oggetto nel sistema.
- **Lista di Controllo degli Accessi (Access Control List - ACL)**: Le liste di permessi sono collegate agli oggetti (risorse).
- **Lista di Capacità (Capability List)**: Le liste di permessi sono collegate ai soggetti (utenti).
- **Controllo degli Accessi Basato sui Ruoli (Role-Based Access Control - RBAC)**: Raggruppa i permessi in ruoli e assegna ruoli agli utenti.

### 14.3.1 Vantaggi di RBAC

- I ruoli modellano specifici lavori o compiti in un'organizzazione.
- Un singolo utente può ricoprire più ruoli contemporaneamente o in momenti diversi.
- Più utenti possono ricoprire lo stesso ruolo.
- L'assegnazione utente-ruolo può essere effettuata separatamente dall'assegnazione ruolo-permesso.

### 14.3.2 Limitazioni di RBAC

- Esplosione dei Ruoli: Il numero di ruoli può crescere eccessivamente.
- Tolleranza al Rischio di Sicurezza: Difficoltà nel gestire livelli differenziati di sicurezza.
- Scalabilità e Dinamicità: Problemi nell'adattarsi a contesti complessi e mutevoli.
- Implementazione Costosa e Difficile: Richiede significativi investimenti di tempo e risorse.

### 14.3.3 Controllo degli Accessi Basato sugli Attributi (Attribute-Based Access Control - ABAC)

ABAC è un modello più avanzato che valuta le autorizzazioni in base agli attributi di:

- Soggetto (utente).
- Risorsa.
- Azione.
- Contesto.

### 14.3.4 Framework per l'Implementazione di Sistemi di Autorizzazione

Esistono numerosi framework che facilitano l'implementazione di sistemi di autorizzazione:

- Keycloak (https://www.keycloak.org/).
- CASL (https://casl.js.org/v6/en/).
- Open Policy Agent (https://www.openpolicyagent.org/).
- Casbin (https://casbin.org/).

# 15 Architettura delle Applicazioni Web e Microservizi

## 15.1 Introduzione all'Architettura delle Applicazioni Web

L'architettura delle applicazioni web si riferisce alla struttura e all'organizzazione delle componenti software che costituiscono un'applicazione web. Essa comprende vari aspetti, tra cui:

- **Client**: l'interfaccia utente, che può essere un browser web o un'applicazione mobile.
- **Server**: il backend che gestisce la logica dell'applicazione e l'accesso ai dati.
- **Database**: il sistema di gestione dei dati, che può essere relazionale o non relazionale.

## 15.2 Modelli di Architettura

Esistono diversi modelli di architettura per le applicazioni web, tra cui:

- **Architettura Monolitica**: tutte le componenti dell'applicazione sono integrate in un'unica unità. Questo modello è semplice da sviluppare e distribuire, ma può diventare difficile da gestire e scalare.
- **Architettura a Microservizi**: l'applicazione è suddivisa in piccoli servizi indipendenti, ognuno dei quali gestisce una specifica funzionalità. Questo modello offre maggiore flessibilità e scalabilità, ma introduce complessità nella gestione delle comunicazioni tra i servizi.
- **Architettura Serverless**: l'applicazione è composta da funzioni che vengono eseguite in risposta a eventi. Non è necessario gestire l'infrastruttura, poiché il provider cloud si occupa della scalabilità e della disponibilità.

## 15.3 Architettura a Microservizi

L'architettura a microservizi è un approccio che consente di costruire applicazioni come una serie di servizi indipendenti. Ogni microservizio è responsabile di una specifica funzionalità e comunica con gli altri servizi tramite API.

### 15.3.1 Vantaggi dei Microservizi

- **Scalabilità**: i microservizi possono essere scalati in modo indipendente, consentendo di gestire carichi di lavoro variabili.
- **Flessibilità tecnologica**: ogni microservizio può essere sviluppato utilizzando tecnologie diverse, consentendo l'adozione delle migliori pratiche per ogni specifica funzionalità.
- **Resilienza**: se un microservizio fallisce, gli altri possono continuare a funzionare, migliorando la disponibilità complessiva dell'applicazione.
- **Sviluppo parallelo**: i team possono lavorare su microservizi diversi in parallelo, accelerando il processo di sviluppo.

### 15.3.2 Svantaggi dei Microservizi

- **Complessità**: la gestione delle comunicazioni tra microservizi può diventare complessa, richiedendo strumenti e pratiche adeguate.
- **Overhead di rete**: le comunicazioni tra microservizi possono introdurre latenza e aumentare il carico di rete.
- **Difficoltà nel testing**: testare un'applicazione a microservizi richiede strategie di testing più sofisticate.

## 15.4 Comunicazione tra Microservizi

I microservizi comunicano tra loro utilizzando vari protocolli e tecnologie, tra cui:

- **REST**: un'architettura basata su HTTP che utilizza metodi come GET, POST, PUT e DELETE per interagire con le risorse.
- **gRPC**: un framework di comunicazione ad alte prestazioni che utilizza HTTP/2 e Protocol Buffers per la serializzazione dei dati.
- **Message Brokers**: sistemi come RabbitMQ o Apache Kafka che gestiscono la comunicazione asincrona tra microservizi tramite messaggi.

## 15.5 Gestione dei Dati nei Microservizi

Ogni microservizio può avere il proprio database, consentendo una maggiore flessibilità nella gestione dei dati. Tuttavia, questo approccio introduce sfide nella coerenza dei dati e nella gestione delle transazioni.

### 15.5.1 Strategie di Gestione dei Dati

- **Database per Microservizio**: ogni microservizio gestisce il proprio database, riducendo le dipendenze tra i servizi.
- **Event Sourcing**: gli eventi che modificano lo stato dell'applicazione vengono memorizzati, consentendo di ricostruire lo stato attuale a partire dagli eventi.
- **CQRS (Command Query Responsibility Segregation)**: separa le operazioni di scrittura (command) da quelle di lettura (query), ottimizzando le prestazioni e la scalabilità.

## 15.6 Sicurezza nei Microservizi

La sicurezza è un aspetto cruciale nell'architettura a microservizi. È importante implementare misure di sicurezza a livello di rete, applicazione e dati.

### 15.6.1 Pratiche di Sicurezza

- **Autenticazione e Autorizzazione**: implementare meccanismi di autenticazione robusti (es. OAuth2) e autorizzazione per controllare l'accesso alle risorse.
- **Crittografia**: proteggere i dati in transito e a riposo utilizzando protocolli di crittografia come TLS.
- **Monitoraggio e Logging**: implementare sistemi di monitoraggio e logging per rilevare attività sospette e garantire la conformità alle normative.

## 15.7 Conclusione

L'architettura a microservizi offre un approccio flessibile e scalabile per lo sviluppo di applicazioni moderne. Tuttavia, richiede una pianificazione attenta e l'adozione di pratiche adeguate per gestire la complessità e garantire la sicurezza. Con la giusta strategia, le applicazioni a microservizi possono fornire un'esperienza utente migliore e una maggiore resilienza rispetto ai modelli monolitici tradizionali.

# 16 Firebase: Piattaforma di Sviluppo Mobile

## 16.1 Introduzione a Firebase

Firebase è una piattaforma di sviluppo mobile e web fornita da Google, che offre una serie di strumenti e servizi per facilitare lo sviluppo di applicazioni. Firebase è particolarmente popolare per le sue funzionalità in tempo reale e per la facilità d'uso, rendendolo una scelta comune per gli sviluppatori di applicazioni mobili.

## 16.2 Componenti Principali di Firebase

Firebase offre diversi servizi, tra cui:

- **Firebase Realtime Database**: un database NoSQL che consente la sincronizzazione dei dati in tempo reale tra client e server.
- **Cloud Firestore**: un database NoSQL scalabile e flessibile, progettato per la sincronizzazione dei dati e le query complesse.
- **Firebase Authentication**: un servizio che semplifica l'autenticazione degli utenti tramite email, password, social media e provider di identità.
- **Firebase Cloud Messaging (FCM)**: un servizio per inviare notifiche push agli utenti.
- **Firebase Hosting**: un servizio di hosting per applicazioni web statiche e dinamiche.
- **Firebase Analytics**: uno strumento per monitorare e analizzare il comportamento degli utenti all'interno dell'app.

## 16.3 Firebase Realtime Database

Il Firebase Realtime Database è un database NoSQL che consente di archiviare e sincronizzare i dati tra client e server in tempo reale. I dati sono memorizzati come JSON e possono essere letti e scritti in modo semplice.

### 16.3.1 Caratteristiche del Realtime Database

- **Sincronizzazione in tempo reale**: i dati vengono sincronizzati automaticamente tra client e server.
- **Offline support**: le applicazioni possono continuare a funzionare anche senza connessione a Internet, grazie alla memorizzazione locale dei dati.
- **Sicurezza**: le regole di sicurezza possono essere configurate per controllare l'accesso ai dati.

### 16.3.2 Esempio di utilizzo del Realtime Database

Per utilizzare il Realtime Database, è necessario configurare il progetto Firebase e aggiungere il pacchetto Firebase al proprio progetto. Ecco un esempio di scrittura e lettura dei dati:

```
DatabaseReference database = FirebaseDatabase.getInstance().getReference();
database.child("users").child(userId).setValue(user);
```

Per leggere i dati:

```
database.child("users").child(userId).addValueEventListener(new ValueEventListener() {
    @Override
    public void onDataChange(DataSnapshot dataSnapshot) {
        User user = dataSnapshot.getValue(User.class);
    }
    @Override
    public void onCancelled(DatabaseError databaseError) {
        // Gestire l'errore
    }
});
```

## 16.4 Cloud Firestore

Cloud Firestore è un database NoSQL scalabile e flessibile, progettato per la sincronizzazione dei dati e le query complesse. A differenza del Realtime Database, Firestore supporta una struttura di dati più complessa e query più potenti.

### 16.4.1 Caratteristiche di Cloud Firestore

- **Struttura a documenti**: i dati sono organizzati in documenti e collezioni.
- **Query avanzate**: supporta query complesse e indicizzazione automatica.
- **Sincronizzazione in tempo reale**: come il Realtime Database, Firestore supporta la sincronizzazione in tempo reale.

### 16.4.2 Esempio di utilizzo di Cloud Firestore

Per utilizzare Firestore, è necessario configurare il progetto Firebase e aggiungere il pacchetto Firestore al proprio progetto. Ecco un esempio di scrittura e lettura dei dati:

```
FirebaseFirestore db = FirebaseFirestore.getInstance();
Map<String, Object> user = new HashMap<>();
user.put("name", "John Doe");
user.put("age", 30);
db.collection("users").document(userId).set(user);
```

Per leggere i dati:

```
db.collection("users").document(userId).get().addOnCompleteListener(new OnCompleteList
    @Override
    public void onComplete(@NonNull Task<DocumentSnapshot> task) {
        if (task.isSuccessful()) {
            DocumentSnapshot document = task.getResult();
            if (document.exists()) {
                User user = document.toObject(User.class);
            }
        }
    }
});
```

## 16.5 Firebase Authentication

Firebase Authentication semplifica l'autenticazione degli utenti, supportando vari metodi di accesso, tra cui email e password, social media e provider di identità.

### 16.5.1 Caratteristiche di Firebase Authentication

- **Supporto per più provider**: consente l'autenticazione tramite Google, Facebook, Twitter, email e password, e altro.
- **Gestione degli utenti**: offre strumenti per gestire gli utenti, come la registrazione, il login e il recupero della password.
- **Sicurezza**: gestisce la sicurezza delle credenziali e delle sessioni utente.

### 16.5.2 Esempio di utilizzo di Firebase Authentication

Per utilizzare Firebase Authentication, è necessario configurare il progetto Firebase e aggiungere il pacchetto Authentication al proprio progetto. Ecco un esempio di registrazione e login:

```
FirebaseAuth mAuth = FirebaseAuth.getInstance();

// Registrazione
mAuth.createUser WithEmailAndPassword(email, password)
    .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                // Registrazione riuscita
            } else {
                // Registrazione fallita
            }
        }
    });

// Login
mAuth.signInWithEmailAndPassword(email, password)
```

```
    .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
        @Override
        public void onComplete(@NonNull Task<AuthResult> task) {
            if (task.isSuccessful()) {
                // Login riuscito
            } else {
                // Login fallito
            }
        }
    });
```

## 16.6 Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging è un servizio che consente di inviare notifiche push agli utenti. FCM è utile per inviare aggiornamenti, promozioni e messaggi personalizzati.

### 16.6.1 Caratteristiche di FCM

- **Invio di notifiche in tempo reale**: consente di inviare messaggi in tempo reale agli utenti.
- **Targeting degli utenti**: consente di inviare messaggi a gruppi specifici di utenti.
- **Analisi delle notifiche**: offre strumenti per monitorare l'efficacia delle notifiche inviate.

### 16.6.2 Esempio di utilizzo di FCM

Per utilizzare FCM, è necessario configurare il progetto Firebase e aggiungere il pacchetto FCM al proprio progetto. Ecco un esempio di invio di una notifica:

```
FirebaseMessaging.getInstance().subscribeToTopic("news")
    .addOnCompleteListener(new OnCompleteListener<Void>() {
        @Override
        public void onComplete(@NonNull Task<Void> task) {
            String msg = "Subscribed to news topic";
            if (!task.isSuccessful()) {
                msg = "Subscription failed";
            }
            Log.d(TAG, msg);
        }
    });
```

## 16.7 Firebase Hosting

Firebase Hosting è un servizio di hosting per applicazioni web statiche e dinamiche. È progettato per essere veloce, sicuro e facile da usare.

### 16.7.1 Caratteristiche di Firebase Hosting

- **Hosting veloce**: distribuzione rapida delle applicazioni web.
- **Sicurezza integrata**: supporta HTTPS per tutte le applicazioni.
- **Facilità di distribuzione**: consente di distribuire le applicazioni con un semplice comando.

### 16.7.2 Esempio di utilizzo di Firebase Hosting

Per utilizzare Firebase Hosting, è necessario installare Firebase CLI e configurare il progetto. Ecco un esempio di distribuzione:

```
firebase init
firebase deploy
```

## 16.8 Firebase Analytics

Firebase Analytics è uno strumento per monitorare e analizzare il comportamento degli utenti all'interno dell'app. Fornisce informazioni preziose per migliorare l'esperienza utente e ottimizzare le strategie di marketing.

### 16.8.1 Caratteristiche di Firebase Analytics

- **Monitoraggio degli eventi**: consente di monitorare eventi personalizzati e predefiniti.
- **Reportistica dettagliata**: offre report dettagliati sul comportamento degli utenti.
- **Integrazione con altri servizi Firebase**: si integra facilmente con altri servizi Firebase per una visione completa delle prestazioni dell'app.

### 16.8.2 Esempio di utilizzo di Firebase Analytics

Per utilizzare Firebase Analytics, è necessario configurare il progetto Firebase e aggiungere il pacchetto Analytics al proprio progetto. Ecco un esempio di monitoraggio di un evento:

```
Bundle bundle = new Bundle();
bundle.putString("item_name", "example_item");
mFirebaseAnalytics.logEvent("select_content", bundle);
```

## 16.9 Conclusione

Firebase è una piattaforma potente e versatile che offre una vasta gamma di strumenti e servizi per lo sviluppo di applicazioni mobili e web. Con funzionalità come il Realtime Database, Cloud Firestore, Authentication, FCM, Hosting e Analytics, Firebase semplifica notevolmente il processo di sviluppo e consente agli sviluppatori di concentrarsi sulla creazione di esperienze utente eccezionali.