# Estruturas de Dados Utilizadas no Projeto Uber SafeStart

## Introdução

Este documento apresenta uma análise detalhada das estruturas de dados empregadas no desenvolvimento do aplicativo Android Uber SafeStart, uma solução voltada para aumentar a segurança no transporte por aplicativo através de prevenção de riscos, alertas proativos e gamificação educativa.

## 1. Classes e Objetos (Modelos de Dados)

São a base da Programação Orientada a Objetos, encapsulando atributos e comportamentos relacionados em unidades coesas.

- **User.java:** Representa o usuário do sistema.
- **Achievement.java:** Representa conquistas do usuário.
- **SimulatedUser.java:** Simula motoristas e passageiros para testes.
- **AudioRecording.java:** Gerencia gravações de áudio com metadados.
- **SafetyTip.java:** Armazena dicas de segurança.
- **Modelos de Requisição/Resposta:** Implementam a comunicação com a API.

**Justificativa Geral:** Permitem organização, tipagem forte e reuso, alinhando-se aos princípios da POO e facilitando a manutenção do código.

## 2. Coleções Dinâmicas

### 2.1. List (ArrayList)

Representa sequências ordenadas de elementos com acesso por índice. É utilizada para armazenar usuários simulados, conquistas, gravações de áudio, dicas de segurança e para mapear respostas da API.

**Justificativa:** Ideal para ordenação, iteração sequencial e acesso indexado; compatível com componentes de UI como `RecyclerView`; suporta operações como filtragem e transformação via streams (Java 8+).

### 2.2. Map (HashMap)

Armazena pares chave-valor com busca rápida em tempo constante (O(1)). É empregada para manter metadados de conquistas, mapear itens de navegação, gerenciar elementos visuais e criar corpos de requisição dinâmicos.

**Justificativa:** Excelente para buscas eficientes por chave, mapeamentos de configuração e estruturação flexível de dados; simplifica a atualização da interface do

usuário através de referências diretas.

## 3. Primitivos e Arrays

### 3.1. Strings

Usadas extensivamente em todo o aplicativo para representar dados textuais como nomes, mensagens, identificadores, chaves para SharedPreferences, URLs de API e conteúdo criptografado.

### 3.2. Arrays Primitivos

Utilizados para dados de tamanho fixo e acesso rápido, como `byte[]` para criptografia e `int[]` para armazenar IDs de interface gráfica.

**Justificativa (para Strings e Arrays Primitivos):** Baixo custo de memória e processamento, ideal para dados binários, criptografia ou listas de tamanho fixo como elementos de UI.

### 3.3. Tipos Primitivos / Wrappers

`int`, `boolean`, `float` e seus equivalentes `Integer`, `Boolean`, `Float` são usados para IDs, flags, pontuações, avaliações e controle de estado em activities.

**Justificativa:** Eficientes em termos de memória e desempenho para valores simples, com wrappers fornecendo funcionalidades adicionais quando necessário.

## 4. Estruturas Específicas do Android e Bibliotecas

### 4.1. SharedPreferences

Armazenamento persistente de pares chave-valor para dados pequenos de configuração, como pontuação do usuário, dados de autenticação, progresso de conquistas e preferências de pareamento.

**Justificativa:** Solução nativa para persistência de dados simples sem necessidade de banco de dados completo; rápido acesso a configurações de usuário e estado do aplicativo entre sessões.

### 4.2. Intent & Bundle

Utilizados para navegação entre telas e transporte de dados entre atividades do Android.

**Justificativa:** Mecanismo padrão do Android para comunicação entre componentes;

permite navegação estruturada e transmissão de dados entre telas mantendo o desacoplamento.

### 4.3. RecyclerView.Adapter

Exibe listas otimizadas com `List<T>` e `ViewHolders`, reutilizando views para performance na exibição de conquistas e dicas de segurança.

**Justificativa:** Oferece desempenho otimizado para listas longas; reutiliza views para economia de memória; padrão recomendado para listagens no Android.

### 4.4. JSONObject

Permite a montagem e leitura manual de JSON para transmissão e processamento de dados estruturados, útil em cenários de criptografia e análise de erros da API.

**Justificativa:** Oferece controle granular sobre a estrutura JSON; útil para casos onde a serialização automática via Gson não é adequada ou para processamento manual de respostas.

### 4.5. okio.Buffer

Lê e escreve dados binários em requisições HTTP com eficiência, especialmente ao acessar o corpo da requisição para criptografia.

**Justificativa:** Oferece manipulação eficiente de dados binários com menos overhead que `InputStream`/`OutputStream` padrão; otimizado para operações de I/O em rede.

### 4.6. AtomicBoolean

Controla estado mutável em ambientes assíncronos garantindo consistência, por exemplo, ao coordenar o estado entre callbacks de API ou o estado de gravação entre threads.

**Justificativa:** Garante operações thread-safe sem necessidade de sincronização explícita; ideal para flags de controle em processamento assíncrono.

### 4.7. MediaRecorder & MediaPlayer

Utilizados para captura e reprodução de áudio no módulo de gravações de segurança.

**Justificativa:** APIs nativas do Android para manipulação de mídia; fornecem controle completo sobre captura e reprodução de áudio com configurações avançadas.

### 4.8. Handler & Looper

Gerenciam tarefas assíncronas e programam ações futuras com precisão, como controle de timers, atualizações de UI e simulação de busca de passageiros.

**Justificativa:** Mecanismo essencial para operações em segundo plano, com retorno garantido à thread principal para atualizações de UI; oferece controle preciso sobre timing de operações.

## Conclusão

As estruturas de dados escolhidas no Uber SafeStart refletem boas práticas de engenharia de software, balanceando eficiência, legibilidade e manutenibilidade. O projeto emprega uma combinação estratégica de:

- Modelos orientados a objetos para representação de domínio
- Coleções dinâmicas para manipulação flexível de dados
- Estruturas específicas do Android para integração com o ecossistema da plataforma
- Primitivos e wrappers para operações fundamentais

Esta arquitetura de dados suporta os principais requisitos funcionais do aplicativo: segurança proativa, monitoramento em tempo real, gamificação e feedback contínuo entre motoristas e passageiros.
O uso eficiente destas estruturas, aliado a padrões de design como adapters, builders e singletons, contribui para um código coeso e extensível, preparando o aplicativo para futuras integrações com sistemas de backend e expansão de funcionalidades.