Mobile Programming

Localization & Internationalization

Lecturer: Ali Najimi
Faculty of Computer Engineering
Sharif University of Technology
Fall 2027

l10n · i18n · Flutter intl · RTL · ARB · Persian



Table of Contents

1. Localization vs. Internationalization
2. Why l10n/i18n Matters
3. Locale & Language Basics
4. Flutter Internationalization Setup
5. ARB Files — Translation Resources
6. Flutter `intl` Package — Code Generation
7. Using Translations in Widgets
8. Locale Resolution & Fallback
9. Right-to-Left (RTL) Support
10. Persian (Farsi) Localization — Deep Dive

Table of Contents

11. Pluralization & Gender Rules
12. Date, Number & Currency Formatting
13. Dynamic Locale Switching
14. Android Native Localization
15. iOS Native Localization
16. Testing Localization
17. Localization Best Practices
18. Common Pitfalls
19. Summary & Key Takeaways

Localization vs. Internationalization

These two terms are often confused but have distinct meanings.

Term	Abbreviation	Definition
Internationalization	**i18n** (18 letters between i and n)	Designing the app so it *can* be adapted to any locale. No hardcoded strings, flexible layouts.
Localization	**l10n** (10 letters between l and n)	*Adapting* the app for a specific locale — translating text, formatting dates/numbers, adjusting images.

> **Rule of thumb:** i18n is the **architecture**, l10n is the **content**. You do i18n first, then l10n for each target language.

Why l10n/i18n Matters

Business Impact:

- **75%+** of users prefer apps in their native language (CSA Research).
- Localized apps see **128%+** increase in downloads (App Annie).
- Google Play & App Store feature apps with multi-language support.

Technical Impact:

- Hardcoded strings → technical debt and expensive refactoring.
- RTL languages (Arabic, Persian, Hebrew) require layout changes.
- Date/number formats vary: `01/02/2026` means Jan 2 (US) or Feb 1 (EU).

Legal Impact:

- Some regions **require** local language support (e.g., France, China).
- Privacy policies must be available in the user's language.

Locale & Language Basics

A **locale** identifies a specific language and region combination.

Locale Structure: `languageCode_CountryCode`

Locale	Language	Region	Example
`en_US`	English	United States	Color, 12/31/2026
`en_GB`	English	United Kingdom	Colour, 31/12/2026
`fa_IR`	Persian	Iran	رنگ، ۱۴۰۵/۰۳/۰۴
`ar_SA`	Arabic	Saudi Arabia	لون، ٢٠٢٦/١٢/٣١
`ja_JP`	Japanese	Japan	色、2026/12/31
`de_DE`	German	Germany	Farbe, 31.12.2026

> **Key:** Language ≠ Locale. `en_US` and `en_GB` share a language but differ in date/number formatting and spelling.

Flutter Internationalization Setup

Flutter provides built-in i18n support. Here's how to set it up.

Step 1 — Add dependencies:

```
# pubspec.yaml
dependencies:
  flutter_localizations:
    sdk: flutter
  intl: ^0.19.0

dev_dependencies:
  intl_utils: ^2.8.0
```

Flutter Internationalization Setup

Step 2 — Enable code generation:

```
# pubspec.yaml
flutter:
  generate: true
```

Flutter Internationalization Setup

Step 3 — Create `l10n.yaml` config:

```
# l10n.yaml
arb-dir: lib/l10n
template-arb-file: app_en.arb
output-localization-file: app_localizations.dart
```

ARB Files — Translation Resources

ARB (Application Resource Bundle) is the standard format for Flutter translations.

`lib/l10n/app_en.arb`

```
{
  "@@locale": "en",
  "appTitle": "My App",
  "@appTitle": {
    "description": "The application title"
  },
  "welcomeMessage": "Hello, {name}!",
  "@welcomeMessage": {
    "description": "Welcome greeting",
    "placeholders": {
      "name": {
        "type": "String"
      }
    }
  }
}
```

`lib/l10n/app_fa.arb`

```
{
  "@@locale": "fa",
  "appTitle": "اپلیکیشن من",
  "welcomeMessage": "سلام، {name}!"
}
```

Flutter `intl` Package — Code Generation

After creating ARB files, run code generation:

```
# Generate localization classes
flutter gen-l10n

# Or use intl_utils for more control
dart run intl_utils:generate
```

Generated files:

- `app_localizations.dart` — Main class with all strings.
- `app_localizations_en.dart` — English translations.
- `app_localizations_fa.dart` — Persian translations.

Flutter `intl` Package — Code Generation

Configure in `MaterialApp`:

```
MaterialApp(
  localizationsDelegates: AppLocalizations.localizationsDelegates,
  supportedLocales: AppLocalizations.supportedLocales,
  locale: Locale('fa'), // Force a specific locale
  home: MyHomePage(),
);
```

Using Translations in Widgets

Once configured, access translations anywhere in your widget tree:

```dart
import 'package:flutter_gen/gen_l10n/app_localizations.dart';

class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final l10n = AppLocalizations.of(context)!;

    return Scaffold(
      appBar: AppBar(
        title: Text(l10n.appTitle),
      ),
      body: Center(
        child: Text(l10n.welcomeMessage('Ali')),
      ),
    );
  }
}
```

Using Translations in Widgets

Key points:

- `AppLocalizations.of(context)` returns `null` if locale is not loaded — use `!` with caution.
- All generated methods are **type-safe** — compile-time errors for missing translations.
- Placeholders become method parameters.

Locale Resolution & Fallback

Control how Flutter resolves the user's preferred locale:

```
MaterialApp(
  localizationsDelegates: AppLocalizations.localizationsDelegates,
  supportedLocales: AppLocalizations.supportedLocales,
  localeResolutionCallback: (locale, supportedLocales) {
    // Exact match
    for (var supportedLocale in supportedLocales) {
      if (supportedLocale.languageCode == locale?.languageCode &&
          supportedLocale.countryCode == locale?.countryCode) {
        return supportedLocale;
      }
    }
    // Language-only match (fallback)
    for (var supportedLocale in supportedLocales) {
      if (supportedLocale.languageCode == locale?.languageCode) {
        return supportedLocale;
      }
    }
    // Default to English
    return Locale('en');
  },
);
```

Right-to-Left (RTL) Support

Flutter handles RTL automatically when you use `Directionality` or set the locale.

Automatic RTL:

- Locales like `fa`, `ar`, `he` automatically set `TextDirection.rtl`.
- `Directionality.of(context)` tells you the current direction.

Widget-level control:

```
Directionality(
  textDirection: TextDirection.rtl,
  child: Text('سلام دنیا'),
)

// Or use the locale-driven direction
Directionality(
  textDirection: Directionality.of(context),
  child: MyWidget(),
)
```

Right-to-Left (RTL) Support

Layout considerations:

- Use `EdgeInsetsDirectional` instead of `EdgeInsets` (for padding/margin).
- Use `AlignmentDirectional` instead of `Alignment`.
- Use `BorderRadiusDirectional` for border radius.

RTL — Common Widgets & Patterns

Use Directional Widgets:

```
// flips with RTL
Row(
  textDirection: TextDirection.rtl,
  children: [
    Icon(Icons.arrow_back),
    Text('بازگشت'),
  ],
)

// Use ListTile — auto RTL
ListTile(
  leading: Icon(Icons.person),
  title: Text('نام کاربری'),
  trailing: Icon(Icons.chevron_left),
)
```

Avoid Hardcoded Directions:

```
// Wrong — always LTR
Padding(
  padding: EdgeInsets.only(left: 16),
  child: Text('Hello'),
)

// Correct — flips with RTL
Padding(
  padding: EdgeInsetsDirectional.only(start: 16),
  child: Text('Hello'),
)
```

Rule: Never use `left` / `right` for layout. Use `start` / `end` via directional APIs.

Persian (Farsi) Localization — Deep Dive

Persian (`fa_IR`) is one of the most common RTL locales for Iranian apps.

Key considerations:

- **Text direction:** RTL — use `Directionality` and directional widgets.
- **Number system:** Persian numerals (۰۱۲۳۴۵۶۷۸۹) vs. Latin (0123456789).
- **Calendar:** Jalali (Solar Hijri) — not Gregorian.
- **Font:** Use a Persian-compatible font (Vazirmatn, IRANSans, Shabnam).

Jalali Calendar — `shamsi_date` package:

```dart
import 'package:shamsi_date/shamsi_date.dart';

final now = Jalali.now();
final formatted =
  '${now.year}/${now.month.toString().padLeft(2, '0')}'
  '/${now.day.toString().padLeft(2, '0')}';
// Output: 1405/03/04
```

Persian — Font & Number Formatting

Custom Persian font:

```yaml
# pubspec.yaml
flutter:
  fonts:
    - family: Vazirmatn
      fonts:
        - asset: assets/fonts/Vazirmatn-Regular.ttf
        - asset: assets/fonts/Vazirmatn-Bold.ttf
          weight: 700
```

```dart
// Apply globally
MaterialApp(
  theme: ThemeData(
    fontFamily: 'Vazirmatn',
    textTheme: TextTheme(
      bodyLarge: TextStyle(fontSize: 16),
      bodyMedium: TextStyle(fontSize: 14),
    ),
  ),
);
```

Pluralization & Gender Rules

Different languages have different plural rules. Flutter's `intl` handles this.

ARB with plurals:

```
{
  "messageCount": "{count, plural, =0{No messages} =1{1 message} other{{count} messages}}",
  "@messageCount": {
    "placeholders": {
      "count": { "type": "int" }
    }
  }
}
```

Pluralization & Gender Rules

Persian plurals:

```
{
  "messageCount": "{count, plural, =0{پیامی نیست} =1{۱ پیام} other{{count} پیام}}"
}
```

Pluralization & Gender Rules

Gender support:

```
{
  "greeting": "{gender, select, male{Mr. {name}} female{Ms. {name}} other{{name}}}",
  "@greeting": {
    "placeholders": {
      "gender": { "type": "String" },
      "name": { "type": "String" }
    }
  }
}
```

Date, Number & Currency Formatting

The `intl` package provides powerful formatting utilities.

```dart
import 'package:intl/intl.dart';
// Date formatting
final today = DateTime.now();
final usFormat = DateFormat.yMd('en_US').format(today); // 12/31/2026
final irFormat = DateFormat.yMd('fa_IR').format(today); // ۱۲/۳۱/۲۰۲۶

// Number formatting
final price = 1250000.50;
final usNumber = NumberFormat.currency(
  locale: 'en_US', symbol: '\$'
).format(price); // $1,250,000.50

final irNumber = NumberFormat.currency(
  locale: 'fa_IR', symbol: 'تومان'
).format(price); // ۱,۲۵۰,۰۰۰.۵۰ تومان
// Compact numbers
final compact = NumberFormat.compact(
  locale: 'en_US'
).format(1250000);// 1.3M
```

Dynamic Locale Switching

Allow users to change the app language at runtime.

```
class LocaleProvider extends ChangeNotifier {
  Locale _locale = Locale('en');
  Locale get locale => _locale;
  void setLocale(Locale newLocale) {
    if (!AppLocalizations.supportedLocales.contains(newLocale)) return;
    _locale = newLocale;
    notifyListeners();
  }
}
// In MaterialApp
MaterialApp(
  locale: provider.locale,
  localizationsDelegates: AppLocalizations.localizationsDelegates,
  supportedLocales: AppLocalizations.supportedLocales,
);
// Switch language from UI
ElevatedButton(
  onPressed: () => provider.setLocale(Locale('fa')),
  child: Text('فارسی'),
),
ElevatedButton(
  onPressed: () => provider.setLocale(Locale('en')),
  child: Text('English'),
```

Android Native Localization

Android uses the `res/values/` folder structure for translations.

Folder structure:

```
res/
  values/           ← Default (English)
    strings.xml
  values-fa/        ← Persian
    strings.xml
  values-ar/        ← Arabic
    strings.xml
  values-de/        ← German
    strings.xml
```

Android Native Localization

`res/values/strings.xml` **(English):**

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">My App</string>
    <string name="welcome">Hello, %s!</string>
    <string name="items_count">%d items</string>
</resources>
```

`res/values-fa/strings.xml` **(Persian):**

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">اپلیکیشن من</string>
    <string name="welcome">سلام، %s!</string>
    <string name="items_count">%d آیتم</string>
</resources>
```

Android Native — RTL & Layout

Android supports RTL natively since API 17 (Android 4.2).

Enable RTL in `AndroidManifest.xml`:

```
<application
    android:supportsRtl="true"
    ... >
```

Use start/end instead of left/right:

```
<!-- Wrong -->
<TextView
    android:paddingLeft="16dp"
    android:layout_alignParentLeft="true" />

<!-- Correct -->
<TextView
    android:paddingStart="16dp"
    android:layout_alignParentStart="true" />
```

Android Native — RTL & Layout

Access strings in Kotlin:

```
val appName = getString(R.string.app_name)
val welcome = getString(R.string.welcome, "Ali")
```

iOS Native Localization

iOS uses `.strings` files and Xcode's localization system.

Folder structure:

```
MyApp/
  en.lproj/
    Localizable.strings
  fa.lproj/
    Localizable.strings
  ar.lproj/
    Localizable.strings
```

iOS Native Localization

`en.lproj/Localizable.strings`:

```
"app.name" = "My App";
"welcome.message" = "Hello, %@!";
"items.count" = "%d items";
```

`fa.lproj/Localizable.strings`:

```
"app.name" = "اپلیکیشن من";
"welcome.message" = "سلام، %@!";
"items.count" = "%d آیتم";
```

Access in Swift:

```swift
let appName = NSLocalizedString("app.name", comment: "App name")
let welcome = String(format: NSLocalizedString("welcome.message", comment: ""), "Ali")
```

iOS Native — String Catalogs (Xcode 15+)

Modern iOS projects use **String Catalogs** (`.xcstrings`) instead of `.strings` files.

Benefits:

- Single file for all languages.
- Xcode automatically extracts strings from code.
- Built-in plural support.
- Visual editor in Xcode.

Setup:

1. File → New → String Catalog → `Localizable.xcstrings`.
2. Xcode auto-detects `NSLocalizedString` and `Text()` usage.
3. Add languages in the catalog editor.

iOS Native — String Catalogs (Xcode 15+)

SwiftUI:

```
// SwiftUI automatically localizes Text views
Text("Welcome") // Looks up "Welcome" key

// With parameters
Text("Hello \(name)") // Automatic string interpolation

// Plural
Text("^[\(count) item](inflect: true)")
```

Testing Localization

Manual Testing:

```
# Flutter — test with specific locale
flutter run --dart-define=LOCALE=fa
# Android — change device language
adb shell setprop persist.sys.locale fa-IR
adb shell setprop persist.sys.locale en-US
# iOS — Simulator language
xcrun simctl setlocale booted fa_IR
```

Testing Localization

Automated Testing:

```
testWidgets('Persian translation loads', (tester) async {
  await tester.pumpWidget(
    MaterialApp(
      locale: Locale('fa'),
      localizationsDelegates: AppLocalizations.localizationsDelegates,
      supportedLocales: AppLocalizations.supportedLocales,
      home: HomeScreen(),
    ),
  );
  await tester.pumpAndSettle();
  expect(find.text('اپلیکیشن من'), findsOneWidget);
});
```

Localization Best Practices

Architecture

- Start i18n from **day one** — even if you only support one language.
- Never hardcode strings in widgets.
- Use ARB files as the **single source of truth**.
- Keep translation keys **descriptive** (`login_button_text` not `text1`).
- Use `@` metadata for translator context.

Content

- Provide **context** in `@description` for translators.
- Use **placeholders** instead of string concatenation.

Layout

- Use `EdgeInsetsDirectional` and `AlignmentDirectional`.
- Test RTL layouts with **both** LTR and RTL locales.
- Avoid fixed widths — text lengths vary by language.
- Use `Flexible` and `Expanded` widgets.
- Test with **right-to-left** languages early.

Workflow

- Use a **translation management** platform (Crowdin, Lokalise, Phrase).
- Automate ARB file sync with CI/CD.
- Review translations with **native speakers**.

Common Pitfalls

Pitfall	Problem	Solution
Hardcoded strings	`"Hello"` in widget code	Always use `l10n.greeting`
String concatenation	`"Hello " + name`	Use placeholders: `l10n.welcome(name)`
Fixed layouts	`SizedBox(width: 200)`	Use `Flexible`, `Expanded`
LTR-only padding	`EdgeInsets.only(left: 16)`	Use `EdgeInsetsDirectional.only(start: 16)`
Ignoring plurals	`item + "s"`	Use ICU plural format in ARB
No fallback locale	Shows key if translation missing	Set `localeResolutionCallback`
Date/number formats	Always `MM/dd/yyyy`	Use `DateFormat` with locale
Missing font glyphs	Default font lacks Arabic/Persian	Bundle a compatible font
Not testing RTL	Layout breaks in Arabic/Persian	Test with both directions

Summary: Key Takeaways

Concepts

- i18n = architecture, l10n = content.
- Locale = language + region (`fa_IR`).
- ARB files are the standard for Flutter translations.
- `flutter gen-l10n` generates type-safe Dart code.
- Always provide a fallback locale.

RTL Support

- Use `EdgeInsetsDirectional`, `AlignmentDirectional`.
- Use `start`/`end` instead of `left`/`right`.
- Test with both LTR and RTL locales.
- Persian/Arabic/Hebrew are RTL.

Persian Specifics

- Jalali calendar via `shamsi_date`.
- Persian-compatible font (Vazirmatn).
- Persian numeral conversion.
- Full RTL layout support.

Native Platforms

- Android: `res/values-{locale}/strings.xml`.
- iOS: `.lproj/Localizable.strings` or `.xcstrings`.
- Both support RTL natively.
- Test on real devices with different locales.

References

- Flutter Internationalization: https://docs.flutter.dev/ui/accessibility-and-internationalization/internationalization
- Flutter `intl` package: https://pub.dev/packages/intl
- `shamsi_date` package: https://pub.dev/packages/shamsi_date
- `intl_utils` package: https://pub.dev/packages/intl_utils
- ARB Format Specification: https://github.com/google/app-resource-bundle
- Android Localization Guide: https://developer.android.com/guide/topics/resources/localization
- iOS Localization Guide: https://developer.apple.com/documentation/xcode/localization
- ICU Message Format: https://unicode-org.github.io/icu/userguide/format_parse/messages/
- Vazirmatn Font: https://github.com/rastikerdar/vazirmatn

Thank You! 🙌

Questions?

> *"I translated my app into 10 languages and still can't understand my own code."*
> *— Every developer after a l10n refactor*