CS 193A

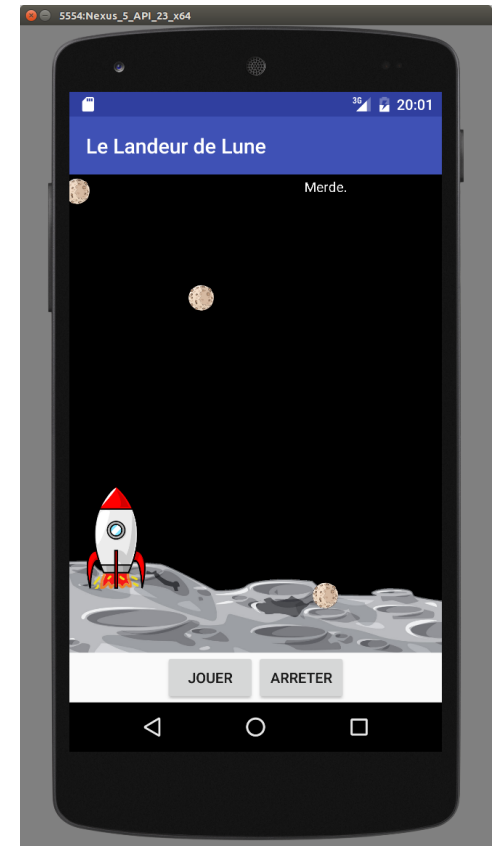## Internationalization (i18n) and Localization (L10n)

An app for the world

- If you want your app to reach the widest possible audience, you may want to add support for other languages and cultures.

- Many international users speak English, but many do not.

- Users will of course prefer an app that reflects their own language and culture over one that doesn't.

Goal of today's lecture

- Let's create a Lunar Lander that also supports **French** language.

  - Text strings in the app will appear in French.

  - Any numbers shown will use the type of number formatting expected by French users.

  - Some images in the app will change to new French-specific versions.

Changing device's locale

- testing locales (on a real Android device, or emulator AVD)
  - Settings → Language & Input → Select Language
    - then launch the app of your choice
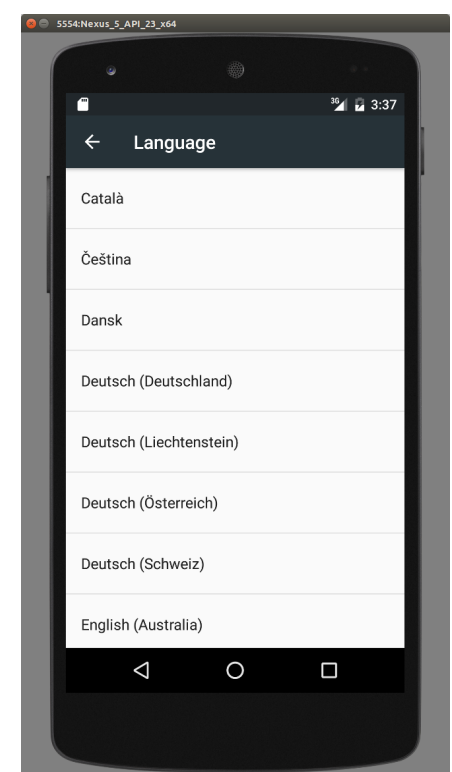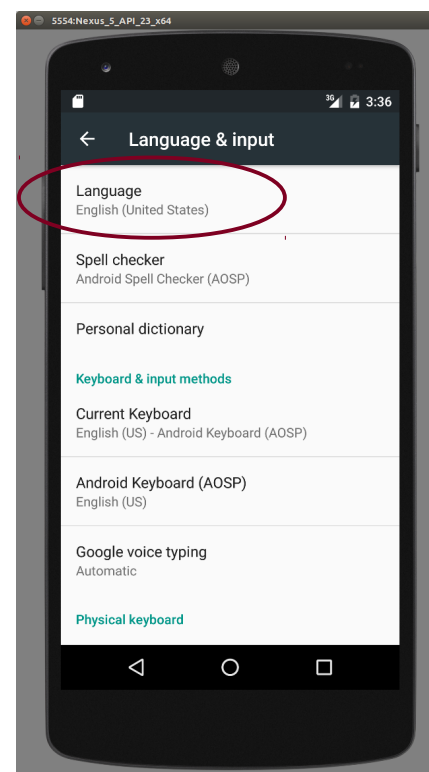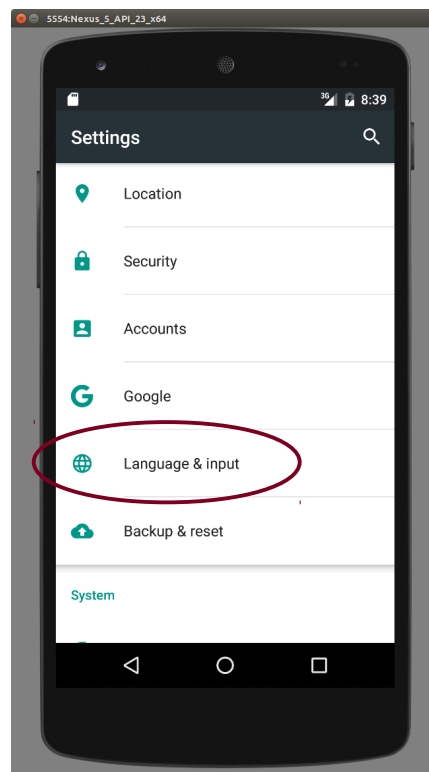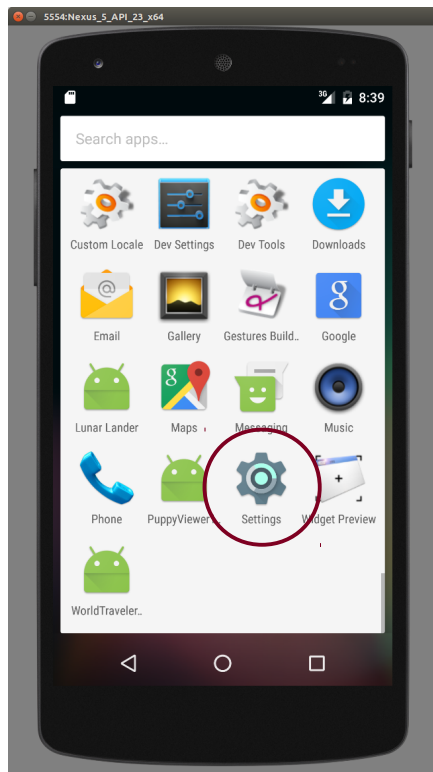
Localization

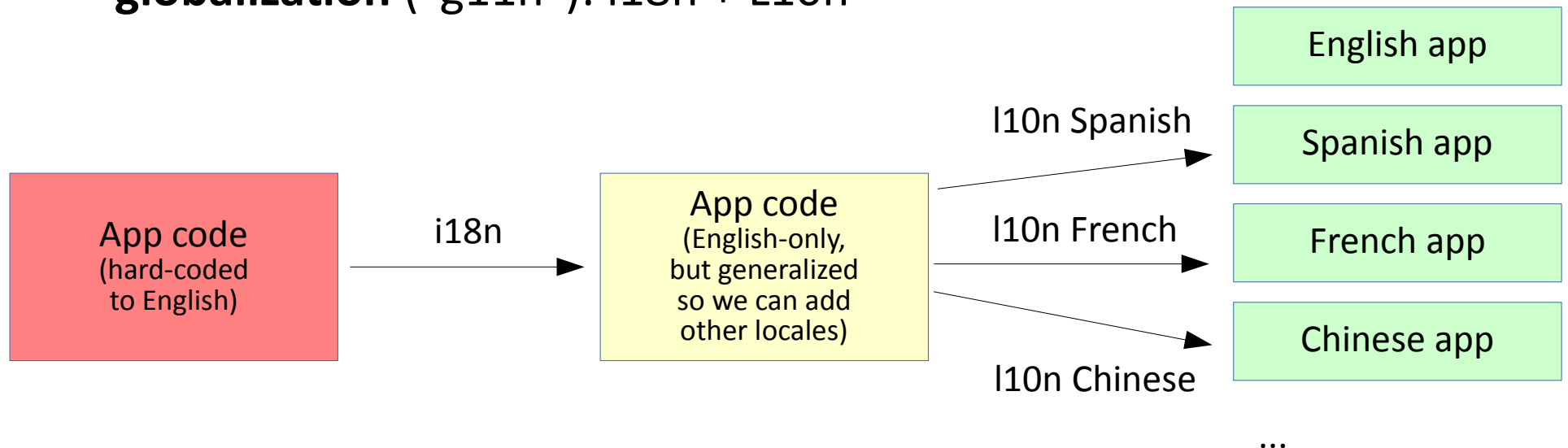- **internationalization** ("i18n"): Generalizing app's code so that it is not hard-coded to one language / region.

    - done once per product (ideally); updated as code is added

- **localization** ("L10n"): Translating internationalized software for a specific region or language.

    - done once per locale; each locale is updated as text is added

    - **globalization** ("g11n"): i18n + L10n

App code (hard-coded to English)	→ i18n →	App code (English-only, but generalized so we can add other locales)	

l10n Spanish → Spanish app

l10n French → French app

l10n Chinese → Chinese app

English app

...

Locales

- **locale**: A geographic/cultural location targeted for localization.

- A locale is often written e.g. **"en-US"** which consists of:
    - a <u>language</u> (e.g. English -> "en")
        - often expressed as an ISO-639-1 code: de, en, fr, ja
    - a <u>location</u> or variant (e.g. United States -> "US")
        - often expressed as an ISO-3166-1 code: CA, US, GB, DE, ES, JP

- Why isn't it enough to specify just the language?
    - different locations may use different conventions, spelling, etc.
        - "color" (US) vs. "colour" (UK)
        - "localize" (US) vs. "localise" (UK)
    - some locations use dialects of a given language
    - other differences (dates, currency, numbers, time zone, etc.)

Differences between locales

- language   English vs. German

- spelling   color vs. colour

- slang   line vs. queue

- numbers formatting   1,234.56 vs. 1234,56

- telephone numbers   (650)123-4567 vs. +1.650.123.4567

- currency units/format $123.45 vs. 123,45€

- date formatting   3/14/16 vs. 2016/Mar/14

- text direction   hello vs. שָׁלוֹם

- keyboard shortcuts

- spoken audio

- video subtitles

How are i18n / l10n done?

- developers **internationalize** the app's code
  - **pull all strings out of code** and into separate resource files
  - call methods that localize/format strings, numbers before printing
  - use libraries to help localize messages

- localizers (maybe not programmers) **localize** the app's text
  - often hired to localize an app for a particular locale at a time
  - desktop apps: possibly compile a different binary for each locale
  - web app: look up localized strings when generating each page
  - mobile app: different resource files depending on locale

Hard-coded Android text strings

- **Goal:** Remove <u>ALL</u> text strings from your XML and Java code!
  - That is, all strings that appear on the screen as part of the UI.

- You may have hard-coded English strings in the XML:

```
<Button ...
      android:text="Click me!" />
```

- You may also have hard-coded strings in the Java code:

```
TextView tv = (TextView) findViewById(R.id.foo);
tv.setText("Welcome");
```

Internationalized strings.xml file

- Declare constant string values in strings.xml file:
  - res/values/strings.xml
  - Each string is given a name which becomes its resource ID.

  ```xml
  <!-- res/values/strings.xml -->
  <resources>
      <string name="app_name">Shopping Cart</string>
      <string name="clickme">Click me!</string>
      <string name="welcome">Welcome</string>
      <string name="buy">Buy</string>
      <string name="sell">Sell</string>
  </resources>
  ```

  - Must escape " and ' in strings as \" and \'

Internationalized XML/Java

- Then refer to those constants in your XML and Java code.
  - in XML: @string/*name*
  - in Java: R.string.*name*

```xml
<!-- activity_main.xml -->
<Button    ... android:text="Click me!" />
<Button    ... android:text="@string/clickme" />
<TextView ... android:text="@string/app_name" />
```

```java
// MainActivity.java
TextView tv = (TextView) findViewById(R.id.foo);
tv.setText("Welcome");
tv.setText(R.string.welcome);
```

Localizing strings

- Now, to localize for a specific language:
  - make a folder values-***Language*** for each supported language
    - res/values-**es**/strings.xml (Spanish)
    - res/values-**fr**/strings.xml (French)

  - add a translated copy of strings.xml to each folder

 - no changes should need to be made to the app XML/Java code!

French strings.xml file

- Example French strings file:

```xml
<!-- res/values-fr/strings.xml -->
<resources>
    <string name="app_name">Panier</string>
    <string name="clickme">Cliquez moi!</string>
    <string name="welcome">Bienvenue</string>
    <string name="buy">Acheter</string>
    <string name="sell">Vendre</string>
</resources>
```

Android Studio string extraction

- In Android Studio, press Alt-Enter on a hard-coded string to get a helpful popup for extracting that string's text as a resource

    - can be done from XML or Java
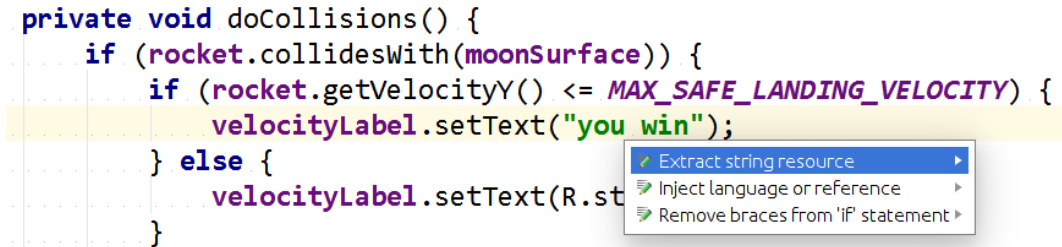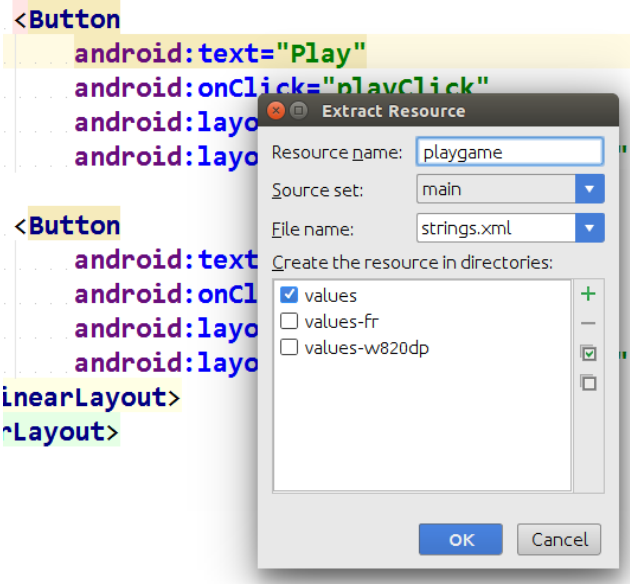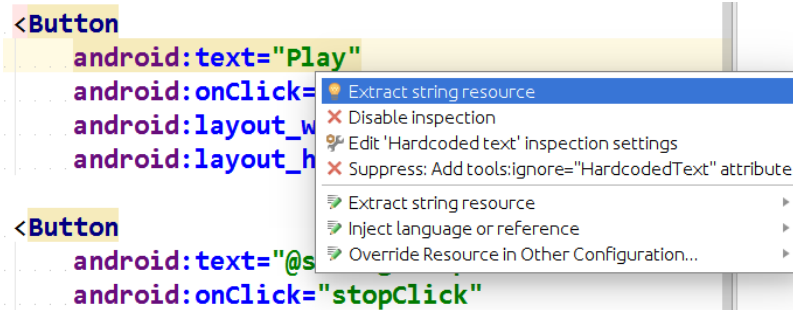
Android Studio locale preview

- In Android Studio's XML Layout "Preview" pane, click the small globe icon to select a preview in a different locale

  - any images, text, etc. that have been localized for that locale will show up without needing to run the app

  - changes made in the Java code won't be seen here (need to run it)
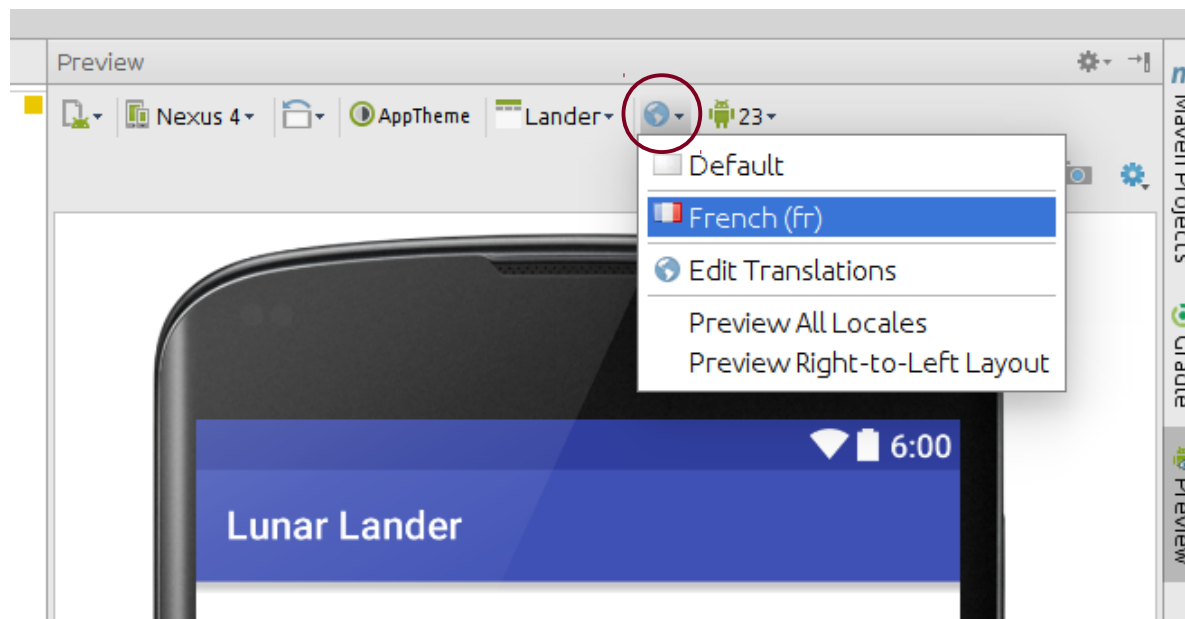
Localized images

- Images are in res/drawable
- Create a folder res/drawable-*Language*
  - res/drawable/companylogo.png (English)
  - res/drawable-**fr**/companylogo.png   (French)

```
<!-- res/layout/activity_main.xml -->
<ImageView
   android:layout_height="wrap_content"
   android:layout_width="wrap_content"
   android:src="@drawable/companylogo" />
```

  - Automatically uses the right one for your locale if available
    - falls back to default one in res/drawable if none available for your locale

Format strings

- Sometimes your app wants to build a UI string based on a value that is not known until runtime.

  - e.g. "You have 32 of 100 credits remaining."

  - In these cases, use a *format string* to insert the value.

- Example:

```
<!-- res/values/strings.xml -->
<resources> ...
  <string name="creditsleft">
    You have %1$d of %2$d credits remaining.</string>

<!-- res/values-fr/strings.xml -->
<resources> ...
  <string name="creditsleft">
    Il y a %1$d de %2$d crédits restant.</string>
```

Format string example

- To fill in a format string, call `getResources` and `getString`
  - Fill in template by passing values for each placeholder

```
int credits = 32;              // MainActivity.java
int total = 100;

// "You have %1$d of %2$d credits remaining."
// "You have 32 of 100 credits remaining."
String credStr = getResources().getString(
                        R.string.creditsleft,
                        credits, total);
```

Format string details (link)

- Placeholder formatting: % *NUMBER* $ *FORMAT* *TYPE*

$d	an integer
$0*N*d	an integer, 0-filled to *N* digits total
$x	an integer, in hexadecimal
$s	string
$f	float or double
$.*N*f	float/double, with *N* digits after decimal point
$b	boolean
$c	character (char)

```
<string name="example">
Student name %1$s, ID #%2$04d has GPA %3$.2f!!
</string>

String out = getResources().getString(R.string.example,
                                "Marty", 35, 3.21786);
// "Student name Marty, ID #0035 has GPA 3.22!!"
```

Pluralization

- Sometimes a string has a singular and plural form.
  - e.g. "You are taking <u>1</u> class."
  - e.g. "You are taking <u>2</u> classes."
  - In these cases, use a *quantity string* to represent the message.
    - Accepted quantities: `zero, one, two, few, many, other`

```xml
<!-- res/values/strings.xml -->
<resources> ...
  <plurals name="classestaken">
    <item quantity="one">You are taking %d class.</item>
    <item quantity="other">You are taking %d classes.</item>
  </plurals>

<!-- res/values-fr/strings.xml -->
<resources> ...
  <plurals name="classestaken">
    <item quantity="one">Vous avez pris une classe.</item>
    <item quantity="other">Vous avez pris %d classes.</item>
  </plurals>
```

Pluralization in Java code

- To fill in a quantity string, use getQuantityString :
  - getQuantityString(*id, quantity, placeholders*)

```
int classes = 4;           // MainActivity.java

String classStr = getResources()
        .getQuantityString(R.string.classestaken,
                        classes, classes);

// "You have taken 4 classes."
```

Localizing numbers (link)

- `java.text.NumberFormat` formats numbers for a locale

```
Locale loc = Locale.getDefault();    // assume France
NumberFormat fmt =
        NumberFormat.getInstance(loc);
double gpa = 3.145;
String result = fmt.format(gpa);     // "3,145"
```

- *Related issue:* If user types numbers into text fields in their locale's format, your app shouldn't crash!

```
// don't use Double.parseDouble, etc.!
String line = myEditText.getText().toString();  // "3,145"
double gpa = fmt.parse(line);                    // 3.145
```

  - throws `ParseException` if text is in invalid format

Localizing currencies $$$

- currencies are represented by ISO-4217 currency identifiers
  - examples: USD, GBP, EUR, JPY, CNY, INR, RUB
  - programming languages don't know exchange rates between currencies (can't tell you how many Euros equals $100.00)
  - but facilities exist for displaying a variable as a currency amount

- `java.text.NumberFormat` and currency objects

```
NumberFormat dollar =
  NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euro =
  NumberFormat.getCurrencyInstance(Locale.GERMANY);
euro.setCurrency(Currency.getInstance("EUR"));
String s = euro.format(123456.78);
```

Localizing dates (link)

- Differences in how to display dates across locales:
    - names of the months/days   (Monday vs. Lundi)
    - ordering of days   (what day does a week start/end)
    - relative order of y/m/d   (3/14/2016 vs. 2016/Mar/14)
    - time zone   (usually offset from UTC/GMT)
    - 12 vs. 24 hour time   (5:00 PM vs. 17:00)

- `java.text.DateFormat` formats dates
    - styles: `DateFormat.DEFAULT, FULL, LONG, MEDIUM, SHORT`

```java
DateFormat fmt = DateFormat.getDateTimeInstance(
        DateFormat.LONG, DateFormat.SHORT, locale);
String dateText = fmt.format(new Date());
Date d = fmt.parse(dateText.trim());  // parse a date
```

SimpleLocalization

| Method	Description
with(*context*)	get a SimpleLocalization instance
format(*id, args*)	format a resource string
get(*id*), get(*id, args*)	look up a resource string
isLTR(), isLTR(*locale*), isRTL(), isRTL(*locale*)	return whether locale is right-to-left
date(*date*), date(*locale*)	format a Date for this locale
currency(*amount*), currency(*amount, locale*)	format an amount of money for this locale
number(*n*), number(*n, locale*)	format a number for this locale
parseLocalizedInt/Long/ Double/Float(*numStr*)	parse string into a number
pluralize(*id, n, args*)	look up a quantity string

Localization gotchas

- Some languages (e.g. German) use long words
  - buttons/labels get too wide for space provided

- some fonts don't have all characters
  - but a smart OS can use font substitution

- regular expressions / text searches may not match i18n input
  - ex. \w "word boundary" doesn't match Unicode word delimiters

- databases / web servers / backends might return text that has not been localized yet
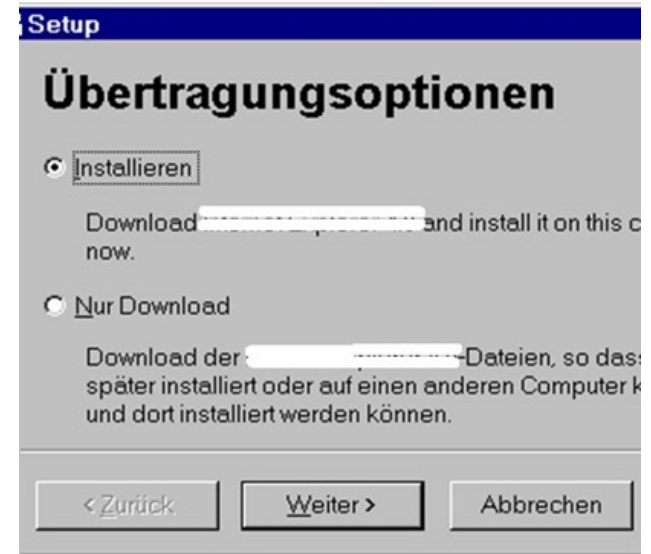
Right-to-left (RTL)

- some languages begin lines on the right side and go left
  - Arabic, Farsi/Persian, Hebrew, Kurdish, Punjabi, Somali, ...
  - hello vs. שָׁלוֹם

- often handled by separate style/layout files for RTL locales

- RTL can lead to lots of subtle UI bugs
  based on coders' LTR assumptions
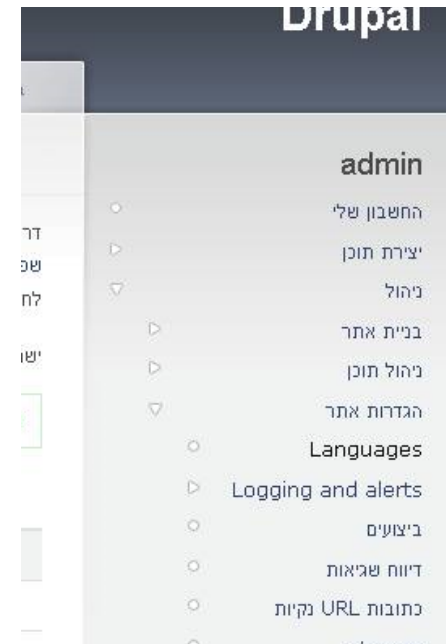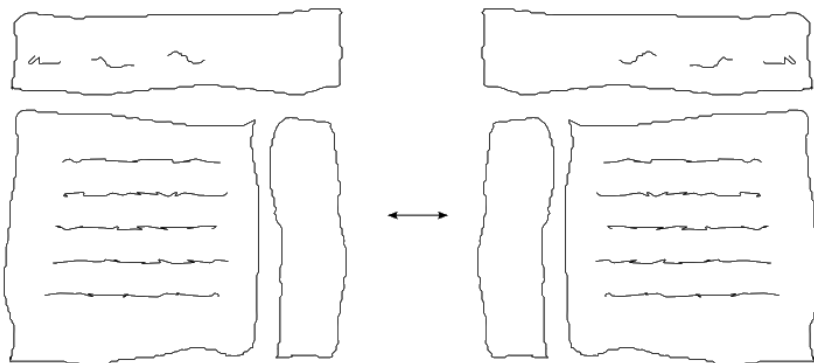
Checking for RTL locales

- In Android API 17 (4.2 Jelly Bean) or later:

```
Configuration config =
        getResources().getConfiguration();
if (config.getLayoutDirection() ==
        View.LAYOUT_DIRECTION_RTL) {
    // in Right To Left layout
}
```

- In older versions of Android:

```
if (ViewCompat.getLayoutDirection(view) ==
        ViewCompat.LAYOUT_DIRECTION_RTL) {
    // the view has RTL layout
} else {
    // the view has LTR layout
}
```

Things to avoid

- Don't hard-code widths/heights in layout or GUI code
  - e.g. `myButton.setWidth(200);`

- Try to avoid images that look like text.

- Avoid using symbols that have no meaning in other locales.
  - USA "STOP sign"
  - Hand up for "Wait"

- (if localizing to RTL locales) Avoid hard-coding the notion that "left" means "start" and "right" means "end".
  - example: Left for "Back", right for "Forward"
  - example: Left for "less", right for "more"

Perils of poor localization

- English words may have different connotation in another language
  - e.g. "Okay" could be translated as "so-so" or "mediocre"

- << and >> , when used as "arrows", can confuse some users whose languages use << and >> as quotation marks

- product's name or ad could translate poorly
  - Microsoft's "Bing" can translate to "disease" in Chinese
  - McDonald's pictures-only billboard in Saudi Arabia
  - ...

- product could offend users from other countries
  - an online dating site that allows users under 16 to register?
  - an online auction site that has bidding end on a holy day?

Poorly localized messages

- "Drop your pants here for best results." - dry cleaning, Tokyo
- "We take your bags and send them in all directions." - Scandinavian airport
- "Ladies may have a fit upstairs." - dry cleaning, Bangkok
- "Teeth extracted by latest methodists." - dentist, Hong Kong
- "Please leave your values at the front desk." - hotel, Paris
- "No smoothen the lion." - zoo, Czech
- "If you consider our help impolite, you should see the manager." - hotel, Athens
- "Our wines leave you nothing to hope for." - Swiss restaurant
- "It is forbidden to enter a woman, even if dressed as a man." - Bangkok temple
- "Fur coats made for ladies from their own skin." - Swedish furrier
- "Specialist in women and other diseases." - doctor, Rome
- "Leave clothes here and spend afternoon having good time." - laundry, Rome
- "We regret that you will be unbearable." - hotel, Bucharest
- "When passenger of foot heave in sight, tootle the horn. Trumpet him melodiously at first, but if he still obstacles you then tootle him with vigor." - car rental, Tokyo

Unicode

Unicode: Standard for storing, encoding, numbering over 107,000 chars from > 90 languages.

- created in 1991 by non-profit Unicode Consortium

- standard character -> integer mappings

- Translation Formats (UTF-*) to store chars as bytes

- supported by languages (Java, .NET, Python), browsers

- important for localization because it defines int'l chars and encodings we will use to present localized text

U+2620	U+2603	U+0E5B	U+22D9	U+2764	U+203D

upside-down
uʍop-ǝpısdn

THAI CHARACTER KHOMUT

SKULL AND CROSSBONES   SNOWMAN

VERY MUCH GREATER-THAN   HEAVY BLACK HEART   INTERROBANG

Character encodings

- **ISO-8859-1**: ANSI, 8-bit (extended ASCII)
  - backward-compatible; simple; mostly English-only

- **UTF-8**: 1 byte for all ANSI chars, which have the same code values as in standard ASCII; up to 4 bytes for other chars

- **UTF-16**: uses 2 bytes for almost all characters, and 4 bytes to encode certain special characters

- Code files, web pages may specify or be saved with an encoding:

```
<html>
    <head>
        <title>CS 142, Winter 2048</title>
        <meta charset="utf-8" />
```