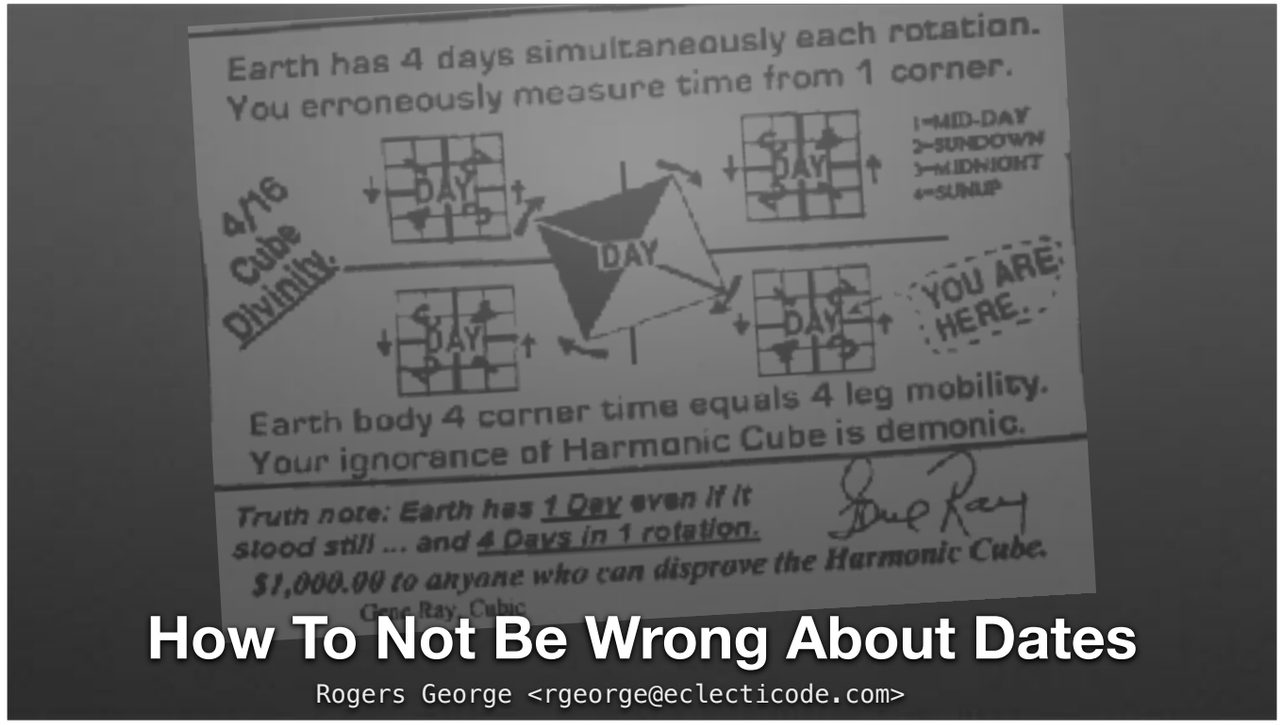


How To Not Be Wrong About Dates

Rogers George <rgeorge@eclecticode.com>

who remembers time cube? You are too educated stupid to comprehend nature's universal harmonic 4-day time cube!

How many "Today"s are there?

I want to talk about dates, whatever a date is
and Time

"Dates Are Hard"

use the platform tools for dates
yeah... but

Lists of Falsehoods are interesting.. but they omit the why

oops 1

```
struct User
{
  var birthday: Date
  // ...
}
```

start with easy mistake, I've made this one (imagine corresponding DB schema &c)

Store a user's birthday to compute their age.

`Date` is a point in time, not coordinates on a calendar. So do we actually know the user's birthday?

User picks their birthday with a calendar picker, but it's stored in db as a point in time (probably user's local midnight) with tz info stripped or converted to Z

since we don't know the user's zone, it may change, we don't really know their birthday

they know their local zone, so it looks OK to them, until...
User picks birthday, then: moves, or, is in different TZ from backend, or, DST changes... boom

oops! GDPR, happy birthday, age of majority mistakes

oops 2

```
let date = /* some local midnight */

for hour in 0...23
{
  let fromtime = date + (hour * 3600)
  let totime = fromtime + 3600
  /* output a table row for that hour */
}
```

Daily schedule. how many hours in the day?

oops, breaks 2x per year

bonus: how many seconds in an hour?

oops 3

```
let components = Calendar.current.dateComponents(
    [.year, .month, .day],
    from: /* a user-picked date; a birthday, say */
)
let json = try! JSONEncoder().encode(
  [ "birthday": [
     "year": components.year!,
     "month": components.month!,
     “day": components.day!
  ], /* other stuff */ ]
)
/* then post the json to a rest endpoint on your backend */
```

subtle, mostly correct

oops, user was using buddhist calendar: 2000 years off

buddhist would post {“userPickedDay”:{“year":2565,"month":5,"day":26}}. Hebrew gives {“userPickedDay":{"day":25,"month":9,"year":5782}} !

ISO formatter vulnerable to this! produces strings that look like ISO8601s... but YMD all wrong

fix: use Calendar.Identifier.iso8601, or Calendar.Identifier.gregorian, or Locale(identifier: “en_US_POSIX”).calendar, and a explicit TZ, all set on the picker too
(those calendars differ by first weekday)

oops n+1

- Does “today” ever go backwards?
- Is “today” my friends’ “today”?
- Is it ever the same “today” over the whole earth?
- Is a string like “Friday, May 27, 2022, 11:25 AM” really a timestamp?

1. users can move time zones, clocks can be reset

2. friends can be global, days will not line up

3. timezones cover 28 hour span (-12 to +14). no single point in time is an unambiguous surrogate for a “day"

4. yes this was real

common theme

You cannot freely convert between

Physical Time

and

Calendar Time

...even though we call them both “Dates”

Thesis:
two worlds, cannot freely wander between them
common modeling error, modeling two disparate things as one

converting between the two is an “egocentric” operation that relies on personal context... so your local tests may pass

It Works on My Machine is not good enough!
inclusivity, multiculturality, globalism

we pretend they’re the same below Seconds.. but even that’s bad news (leap vs solar)

What even is a “date” anyway?

- **Position in Time**
  - natural, physical
  - simple
  - can do math with it
  - ... but useless for humans
- **Symbols on some Clock and Calendar**
  - subjective, arbitrary
  - historic, variable
  - messy, multiple standards
  - ... but this is where humans live

we love living in the physical world
can compare, do easy math, unambiguous “now”
until it’s time for display. What time is unix 1653589547 ?
Does not “have”/exist in a timezone! (it’s always now everywhere)

clock/calendar:
all about counting culturally significant things
prayers, days, moons, seasons, kings (sec, min, hour, day, month, year, era)
different scales have different cultural purposes
subjective: different cultures and places have different rules
historic: Gregorian only became global in 2016 (Saudi; Turkey was 1926); DST and TZs fudged annually

The mapping is not 1 : 1 !

- An absolute point in time maps to many different calendar dates
  - May differ in any component
    - even Era!
- A calendar date only describes a point in time *in a context*
  - which includes: calendar system, era, *full* timezone, location, DST rules
  - assuming any part of the context is *egocentric*

even just using Gregorian, about 40 (1 per zone)

(japanese eras are frequent)

because of the 28-hour TZ span, two identical Y/M/D in different contexts may not even intersect

conversion context cannot be global unless doing POSIXy things.

“egocentric”: “This thing that is true for me is true for everybody”

Physical Time

count of units from an epoch (“a reference point from which time is measured”)

popular ones include:

- Unix: Solar Seconds since 00:00:00 UTC on 1 January 1970, Int
  - Java: same, but 64-bit milliseconds, Int64
- Apple: Solar Seconds since 00:00:00 UTC on 1 January 2001, Double
- Julian days: Solar Days since November 24, 4714 B.C., noon Z, using the proleptic Gregorian calendar

Couple ways of measuring this

“Solar Seconds” heh, little surprise there. put off a few slides

Apple 1st correct approach: Double, imprecise

not just computers. Astronomers use Julian. “proleptic” $2 word = “retroactively applied anachronistically”. Hints at one significance of “Calendar”s and zones: timespans they apply to

Physical Time

Easy!

- basic math works: comparison, addition, subtraction, subdivision
- Forms a one-dimensional space
- Is not “in” a time zone at all

absolute. It’s always “now” everywhere

no other calendrical weirdness

“3600” and “86400” common magic numbers here - danger sign. or “365” near a modulo-4 operator

Physical Time

Pitfalls!

- can overflow
- imprecise
  - clock skew
  - imprecise hardware
  - coarse or unknown resolution
  - (not 100% truly physical: solar vs. UTC seconds)
- ...and of course it's useless for humans

03:14:07 UTC on 19 January 2038 is coming, databases and embedded still speak 32-bit, don't get smug

Int-based types imply precision they don't offer. Apple gets this right (double)
SQLite gets this slightly right (fractional julian days)

Nasty Surprise: not quite physical; "UTC" is a lie: unix etc count solar seconds, not UTC seconds
digress into difference if interest, implies that this isn't really physical time (or is only if earth-referenced)
Leap Second handling important for process control, physical measurements. Find out what your platform does - your date fns may not be good enough
"leap smear" or repeating the leap second are common strategies
be ready for your precise-timing code and date code to suddenly disagree

Calendar (and Clock) Time

Combinations of counted things

- Days
- Seasons
- Moon phases
- Culturally Significant Events and patterns
- Sub-day divisions

almost none of these divide nicely into one another;

different scales have different cultural purposes

Culturally Significant Events: week cycle; reigns of kings
Rough Sub-day divisions: liturgical and work hours; forenoon/afternoon

seasons don't divide nicely into moons ("harvest moon" etc = lunisolar approximation, phase relative to equinox)

seasons/years don't divide nicely into days, so: Leap Year rules: none, Julian modulo 4, Gregorian modulo 4 minus modulo 100 plus modulo 400

moons don't divide nicely into days or years: let them rotate; intercalary months; insert extra non-month days; pick arbitrary varying lengths

Day length bounces around with the season: start day at dawn; or, fudge the clock an hour for that approximate effect

precession of the equinoxes: seasons don't quite match up with the sidereal year. only the astrologers care?

Calendar Time

Highly subjective!

- days drift out of phase as you travel east or west
- seasons invert north to south, days change length
- different cultural “week” conventions
- different kings, different eras

assuming your local context globally is **Egocentric**

Inuit don’t use solar at all
Japanese calendar still uses 1 era per emperor (showa through 1988; heisei through 2019, now Reiwa)

“AD” invented by Dionysius Exiguus in 525, starts with 532 a.d. = “Diocletian Anno Martyrium 247”, was 800 years old by wide adoption
puts Jesus birth 4-6 “BC” (Herod died in 4BC [Matthew 2:15])

Roman 8-day week
Chinese and French Republican 10-day week
Soviet 6 and 5 day weeks, others

Calendar Time

Dealing with all this has caused a “mess”

We deal with it by using explicit context:

- Calendar System
- Time Zone
  - offsets
  - DST rules

It’s easy to think of year-month-day-hour-minute-second being a nice big-endian list of counters.
Do Not Fall For This

The correct answer for any question of the form, “how many x in a y”, is, “I don’t know”.

55+ different timezones, considering offset and tz use (more given slight variations in TZ txn dates)

zone ≠ offset!

Calendar Systems

- Apple alone supports 16 of these!
- Yes, ones other than “Gregorian” get used
- Some are Gregorian + different year 1, some aren’t
- For old dates, “Gregorian” is ambiguous

there are more
Soviet
French Republican (decimal) (17 years)
Swatch Internet Time
Discordian

Gregorian last adopted in 2016 (by Saudi) (Turkey was in 1927)

Mostly, these days, are Gregorian plus different zero year. Dangerous! makes it harder to spot when calendar confusion happens

Proleptic gregorian vs. julian changeover. see “cal sep 1752”

fun with time zones, sniped from wikipedia

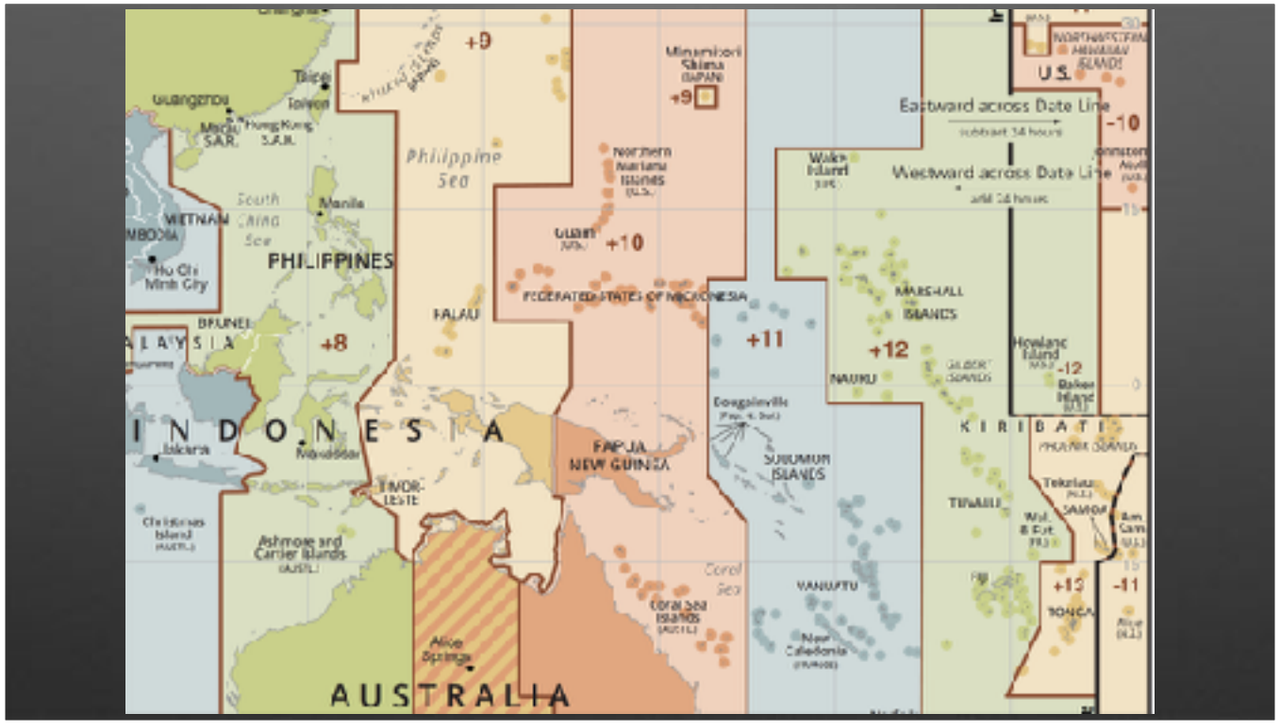
55+ different timezones, considering offset and tz use (more given slight variations in TZ txn dates)
highly malleable! annual-ish amendments (DST)

non-integral gmt offsets

No official names - IANA / Olson IDs come close (region + city, cities are more permanent than countries). (Europe/Kiev disputed)

see “Falsehoods programmers believe about time zones” web page

don’t query GPS to figure this out



nice: -12 (Baker and Howland) uninhabited. “anywhere on earth” zone

however: -11 (Niue, Am.Sam.) very inhabited, +13 (Tokelau, Samoa) also very inhabited

26-hour spread (25-hour inhabited)
+14 Kiritimati (Line Islands), -12 is Baker and Howland islands (uninhabited), -11 is American Samoa and Niue. (+13 is Tokelau and Samoa.)

Kiritimati begins a new calendar day an hour before American Samoa ends the previous one. (and 2 whole hours before legal “anywhere on earth”)

(+14 recent-ish change (1995); Kiribati does more business with Australia than with the US & wants to be on their side of the date line)

Calendar Time

But we need this!

- Birthdays
- Deadlines
- Business Transactions
- Laws
- Customer Expectations

We can’t live in the physical world! Calendar is where Humans live! Humans are all egocentric

This is part of our Human Interface, and the Humans win

we have to talk to people in their language & fulfill their expectations

birthday = legal entity, exact span of time doesn’t matter

deadlines: aviation = by end of X full months since event

Customers think in days and clock times, don’t want to be surprised by DST or Zone stuff. Start work at 1:30am - how does alarm clock handle it at fall DST?

“Date” could mean one of Two things

Physical Time ≠ Calendar Time

or

You rarely have both!

You may have a physical time (“timestamp”, “event”)
OR You may have a calendar date or time (days, seasons, kings)
They are NOT fungible
except with respect to some context: Calendar, Zone, cannot infer or assume

Calendars are ways of looking at or generating a physical time, from a certain point of view (zone)
Roundtrip with extreme care!

Datatypes

Don’t match up exactly with either kind of date

Must be tracked manually through your data flow

Must be aware of which one you “have” at all points in your data flow

your platform gives you some help. they vary on how much help

Datatypes

Physical Time, almost.

Platform fundamental “Date” types		
Apple	`Date, NSDate`	Double seconds
Unix	`time_t`	Int seconds
Java	`java.util.Date`	64-bit milliseconds
MySQL	`timestamp`	32-bit Unix time in disguise
JS		
Win		

(weakest part of the talk: no Microsoft or Javascript perspective)

These don’t “have” any component!
they definitely aren’t “in” a zone

Sometimes used for Calendar Dates when the db author didn’t know any better

btw these are Solar Seconds when dealing with absolute times! Relative times… who knows?

Datatypes

Calendar Time

Platform types for clock and calendar dates		
Apple	`DateComponents`, `NSDateComponents`	arbitrary mix of fields, all optional, incl. zone
Unix	`struct tm`	all possible fields (but not all are significant)
Java	`java.util.Calendar`	does lots more

MySQL	`DATE`
	`TIME`
	`DATETIME`
		year month day
		hour minutes seconds
		ymdhms
		(but not TZ!)

JS		
Win		

types
Apple: (NS)DateComponents, arbitrary mix of fields. but also represents a span of components (“3 days” = “the third day”)
and detours into (NS)Date for doing date math. No DST

Unix: `struct tm`, all fields (but not all are significant in both directions). Has DST!

Java: java.util.Calendar (but watch out, it does more)

from MySQL docs: “MySQL converts TIMESTAMP values from the current time zone to UTC for storage, and back from UTC to the current time zone for retrieval. (This does not occur for other types such as DATETIME.)”

But sometimes this is used to store a Physical time!
DBs are again often culprits
need well-defined global state to untangle

Datatypes

ISO 8601 Strings

`“2022-05-27T16:01:29Z”`

`“2022-05-27T11:01:29-05:00”`

`“2022-05-27T17:01:29+01:00”`

What do I have?

its components are not independently meaningful!

don’t know what zone it was generated in, offset is not zone
don’t know DST
so:
An ISO8601 string is a physical time!

Conversion

- Get a handle on your egocentric state!
  - make Zone and maybe Calendar System explicit in your domain
  - user Zone is a must; user Calendar is a delightful feature
- Avoid old deprecated methods
- Never pretend you have something that you don’t

How to get from one to the other?

explicit state needs to be part of your domain. May fix on Gregorian internally, but allowing user cal for format/IO is a gracious move

Don’t try to fake a TZ with a lat+lon or an offset. Store User/TZ as olson name not offset

“what day does this shift start” (have phys time?) don’t know!
“when does this shift start” (have cal/clock?) don’t know!

Mitigation Strategies

Oops, someone modeled a calendar date as a timestamp!

```
User.birthday = "1971-05-13T05:00:00Z"
```

or worse

```
User.birthday = 42958800
```

and we can't fix it...

schemas and apis are forever

Mitigation Strategies

Oops, someone modeled a calendar date as a timestamp!

Real fix:

```
User.birthday = {year: 1971, month: 5, day: 13}
```

Hack fix, if we can’t change the schema/types:

```
User.birthday = “1971-05-13T11:00:00Z” (42984000)
```

- Store as a GMT 11am
- Convert to components using a GMT Calendar

Best: store what you mean

Hack that might work: store GMT 11ams

then going forward, new code renders into GMT and treats h:m:s as irrelevant

old bad code that renders into the local zone only misses +14 zone. might be small enough?

Noonish rather than midnight is good practice whenever doing Calendar math with absolute times

Mitigation Strategies

Oops, someone assumed there are *n* somethings in a something else!

```
for hour in 0...23 {...}

let tomorrow = today + 86400

for hourtime in stride(from: today.timeIntervalSince1970,
                       through: tomorrow.timeIntervalSince1970,
                       by: 3600) {...}
```

(or something even worse like counting days in a month)

some antipatterns

btw, numbers like this in a pull request should flunk code review: there oughta be a lint rule for ‘86400’ and maybe ‘3600’

Mitigation Strategies

Oops, someone assumed there are *n* somethings in a something else!

```
var cal = Calendar(identifier: .iso8601)

cal.timeZone = TimeZone(secondsFromGMT: 0)!

var birthdaycomponents = DateComponents(year: 1971,
                                        month: 5,
                                        day: 13,
                                        hour: 12)

let birthday = cal.date(from: birthdaycomponents)!

let dayafterbirthday = cal.date(byAdding: .day,
                                value: 1,
                                to: birthday)
```

The platform is there to help you, use it

unfortunately this one forces you to round-trip

Mitigation Strategies

Oops, someone assumed there are *n* somethings in a something else!

```
for day in cal.range(of: .day, in: .month, for: birthday)!
{
  var components = birthdaycomponents
  components.day = day
  let date = cal.date(from: components)
  /* do something interesting for that day */
}
```

The platform is there to help you, use it

enumerating units within larger units should always use the OS

Mitigation Strategies

Oops, someone assumed time is a series of universal days!

Your events on a timeline,

grouped by Day

fitness app. just look at shape of graph

Left: in app, grouped by local day when created

Middle: same data, apple health

Right: same... but grouped by different days (Mumbai)

problem? app should match AH, no?

and we can't fix it, because we attached metadata to those calendar days (goal values)

problem: glitch on travel days
extra problem: now the system is stuck storing that metadata for every single day for every single user

Mitigation Strategies

Oops, someone assumed time is a series of universal days!

No quick fix, but a hard choice:

- Lean into it: Events are now “Calendar”-only
- Repaginate: Page by local days
- Re-engineer

This is a toughie and goes all the way to requirements

Lean into it:
accept that events may no longer be on their correct “day” (chosen strategy)

Repaginate:
accept that daily cycles in the past may no longer line up with days. lose metadata correspondence

Re-engineer:
take it all the way back to requirements and fix your fundamental domain model. What does a “day” mean when a user travels?

Summary

- Points in time and Calendar Dates are **not the same thing**
  - A round-trip between them is an “**egocentric**” operation
- Use your platform’s facilities to do calendar math on Calendar Dates
  - ...but use them with a correct mental model of time and calendars
- Never assume there are exactly *n* somethings in a something else
- Almost isn’t good enough: tests may pass on code that will eventually explode
- The world is a very big place, and it is not centered on you

1. so don’t try to model them as such
    1. model or specify your full context. Zone (not offset), calendar system
2. bearing in mind that they may be badly designed

3. that’s the platform’s job. 23/25 hours in a DST day, all kinds of wacky months, weird leap years (100/400 rule)

5. egocentric tests will pass! watch out.

7. your audience is not you. understand calendar math vs time math in your domain

Mitigation Strategies

Oops, someone else here has a conundrum!

?

war stories: wing it

Solar Seconds and leap seconds digression here

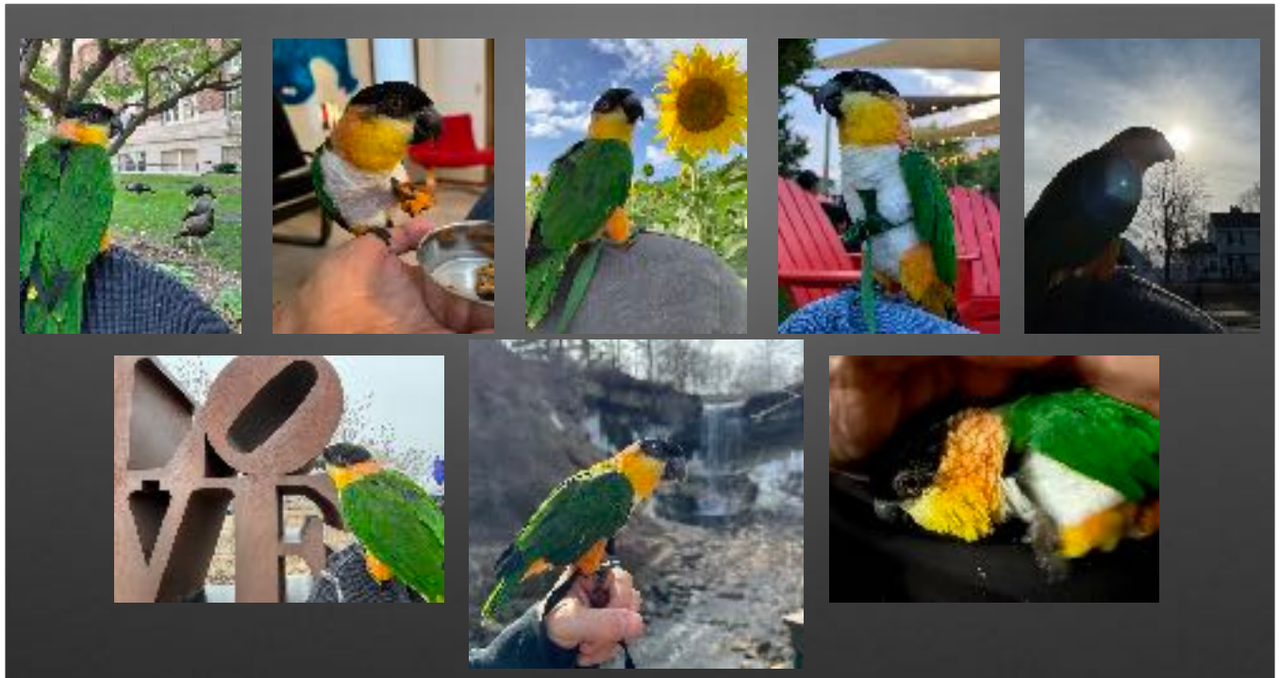


image acknowledgements:
Gene Ray (dec.)
Creative Commons/Wikipedia
Own Work