Iterators

(* This type specifies the interface for iterators *)

type 'a iterator = {
    hasnext: unit -> bool;
    next: unit -> 'a
}

• An iterator object lets us enumerate the elements of some underlying collection.
  – i.e. “Iterator” is an abstraction for interacting with collections of data.

• An object is a record of functions that share some common (encapsulated) state
  – e.g. The state of an iterator remembers a “current pointer” into the collection datatype; the pointer is shared between hasnext and next.

Making a Queue Iterator

let new_queue_iterator (q: 'a queue) : 'a iterator =
(* create a fresh reference with same contents as q.tail *)
let current = ref !q.tail in
{
    next = (fun () ->
        begin match !current with
        | None ->
            failwith "next called on exhausted iterator"
        | Some qn ->
            current := !(qn.prev);
            qn.v
        end)
    ;
    hasnext = (fun () ->
        !current <> None)
}
Making a List Iterator

```ocaml
class new_list_iterator (l: 'a list) : 'a iterator =
    (* create a fresh reference to the list *)
let current = ref l in
{ 
    next = (fun () ->
        begin match !current with
        | [] ->
            failwith "next called on exhausted iterator"
        | h::t ->
            current := t;
            h
        end);
    hasnext = (fun () ->
        !current <> [])
}
```

And similarly for trees, sets, maps, etc.

Taking Stock: CIS120 so far...

- General Design Strategy
  - Understand the problem / Formulate the interface / Generate Tests / Write the code (refine; apply the design strategy recursively)
- Recursive, generic datatypes
  - lists, trees, etc.
  - Design pattern: recursion
- Impérative data structures
  - Mutable reference cells in the heap
  - Queues
- Several styles of abstraction:
  - Generic functions (map, fold)
  - Abstract datatypes via modules and interfaces (set, cmap)
  - Objects (counter, iterator)

The CIS120 Trajectory

- This week: putting it all together
  - Build a GUI library and client application from scratch in OCaml
- Several purposes:
  - Show you that you have enough knowledge to do some pretty serious programming (just more of the same).
  - Work through a more interesting design process
  - Illustrate the event-driven programming model
  - Motivate the features of object-oriented languages like Java
  - Give you a feel for how real GUI libraries (like Java’s Swing) work

- Next week: transition to Java

Demo: GUI Paint Application
Designing a GUI

OCaml’s Graphics library* provides very simple primitives for:
- Creating a window
- Drawing various shapes: points, lines, text, rectangles, circles, etc.
- Getting the mouse position, whether the mouse button is pressed, what key is pressed, etc.

- See gdemo.ml

- How do we go from that to a functioning GUI?

*Pragmatic note: when compiling a program that uses the Graphics module, add graphics.cmxa (for native compilation) or graphics.cma (for bytecode compilation) to OCaml Build Flags under the Projects->Properties dialog in Eclipse.

Step #1: Understand the Problem

- What are the concepts involved and how do they relate to each other?

  - Class answers:
    - layout?
    - colors
    - buttons, menus, text, handles, lines, tabs
    - “clickable” things, stuff that the user can interact with
    - mouse cursor

Design Challenge #1: Abstracting Layout

- How can we make it so that we can re-use the functions that draw different widgets (buttons, check boxes, text, etc.) in different places on the window?
A graphics context `Gctx.t` represents a position within the window, relative to which the widget-local coordinates should be interpreted. We can add additional context information that should be “inherited” by children widgets (e.g., current pen color).

---

## Simple Widgets

(* An interface for simple GUI widgets *)

type t = {
  repaint : Gctx.t -> unit;
  size : Gctx.t -> (int * int)
}

- You can ask a simple widget to repaint itself.
- You can ask a simple widget to tell you its size.
- Both operations are relative to a graphics context

---

**Module: Gctx**

Graphics Contexts

---

**Module: SimpleWidget**

See swdemo.ml for an example of how to use the library