CS61B Lecture #36

Today:

- A Brief Side Trip: Enumeration types.

- *DSIJ,* Chapter 10, *HFJ,* pp. 489–516.

  – Threads

  – Communication between threads

  – Synchronization

  – Mailboxes

Side Trip into Java: Enumeration Types

- Problem: Need a type to represent something that has a few discrete values.

- In the purest form, the only necessary operations are == the only property of a value of the type is that it differs others.

- In older versions of Java, used named integer constants:

```
interface Pieces {
   int BLACK_PIECE = 0,      // Fields in interfaces are static
       BLACK_KING = 1,
       WHITE_PIECE = 2,
       WHITE_KING = 3,
       EMPTY = 4;
}
```

- C and C++ provide *enumeration types* as a shorthand, with syr this:

```
enum Piece { BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING
```

- But since all these values are basically **ints**, accidents can ha

Enum Types in Java

- New version of Java allows syntax like that of C or C++, b
  more guarantees:

  ```
  public enum Piece {
      BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
  }
  ```

- Defines `Piece` as a new reference type, a special kind of cla

- The names `BLACK_PIECE`, etc., are static, final *enumeration co*
  (or *enumerals*) of type `PIECE`.

- They are automatically initialized, and are the only values
  enumeration type that exist (illegal to use **new** to create
  value.)

- Can safely use ==, and also `switch` statements:

  ```
  boolean isKing(Piece p) {
    switch (p) {
      case BLACK_KING: case WHITE_KING: return true;
      default: return false;
    }
  }
  ```

Making Enumerals Available Elsewhere

- Enumerals like `BLACK_PIECE` are static members of a class, no

- Therefore, unlike C or C++, their declarations are not autom
  visible outside the enumeration class definition.

- So, in other classes, must write `Piece.BLACK_PIECE`, which
  annoying.

- However, with version 1.5, Java has *static imports:* to im
  static definitions of class `checkers.Piece` (including enumer
  write

      import static checkers.Piece.*;

  among the import clauses.

- Alas, *cannot* use this for enum classes in the anonymous pac

Operations on Enum Types

- Order of declaration of enumeration constants significant: .
  gives the position (numbering from 0) of an enumeration valu
  `Piece.BLACK_KING.ordinal()` is 1.

- The array `Piece.values()` gives all the possible values of th
  Thus, you can write:

  ```
  for (Piece p : Piece.values())
      System.out.printf("Piece value #%d is %s%n", p.ordinal(
  ```

- The static function `Piece.valueOf` converts a String into a
  type `Piece`. So `Piece.valueOf("EMPTY") == EMPTY`.

Fancy Enum Types

- Enums are classes. You can define all the extra fields, meth[
constructors you want.

- Constructors are used only in creating enumeration constar
constructor arguments follow the constant name:

```java
enum Piece {
    BLACK_PIECE(BLACK, false, "b"), BLACK_KING(BLACK, true, "E
    WHITE_PIECE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W
    EMPTY(null, false, " ");

    private final Side color;
    private final boolean isKing;
    private final String textName;

    Piece(Side color, boolean isKing, String textName) {
        this.color = color; this.isKing = isKing; this.textName
    }

    Side color() { return color; }
    boolean isKing() { return isKing; }
    String textName() { return textName; }
}
```

Threads

- So far, all our programs consist of single sequence of instru

- Each such sequence is called a *thread* (for "thread of con
  Java.

- Java supports programs containing *multiple* threads, which (
  tually) run concurrently.

- Actually, on a uniprocessor, only one thread at a time actua
  while others wait, but this is largely invisible.

- To allow program access to threads, Java provides the type
  in `java.lang`. Each `Thread` contains information about, and c
  one thread.

- Simultaneous access to data from two threads can cause c
  are also constructs for controlled communication, allowing
  to *lock* objects, to *wait* to be notified of events, and to *in*
  other threads.

But Why?

- Typical Java programs always have > 1 thread: besides t
  program, others clean up garbage objects, receive signals,
  the display, other stuff.

- When programs deal with asynchronous events, is sometime
  nient to organize into subprograms, one for each independ
  lated sequence of events.

- Threads allow us to insulate one such subprogram from anot

- GUIs often organized like this: application is doing some
  tation or I/O, another thread waits for mouse clicks (like
  another pays attention to updating the screen as needed.

- Large servers like search engines may be organized this w
  one thread per request.

- And, of course, sometimes we *do* have a real multiprocessor

Java Mechanics

- To specify the actions "walking" and "chewing gum":

```
class Chewer1 implements Runnable {
    public void run()
        { while (true) ChewGum(); }
}
class Walker1 implements Runnable {
    public void run()
        { while (true) Walk(); }
}
```

```
// Walk and chew gum
Thread chomp
   = new Thread(new
Chewer1());
Thread clomp
   = new Thread(new
Walker1());
chomp.start(); clomp.st
```

- Concise Alternative (uses fact that Thread implements Runn

```
class Chewer2 extends Thread {
    public void run()
        { while (true) ChewGum(); }
}
class Walker2 extends Thread {
    public void run()
        { while (true) Walk(); }
}
```

```
Thread chomp = new Chew
       clomp = new Walk
chomp.start();
clomp.start();
```

Avoiding Interference

- When one thread has data for another, one must wait for th[e data]
  to be ready.

- Likewise, if two threads use the same data structure, gener[ally]
  one should modify it at a time; other must wait.

- E.g., what would happen if two threads simultaneously inse[rt an]
  item into a linked list at the same point in the list?

- A: Both could conceivably execute

  ```
  p.next = new ListCell(x, p.next);
  ```

  with the *same* values of `p` and `p.next`; one insertion is lost.

- Can arrange for only one thread at a time to execute a meth[od on a]
  particular object with either of the following equivalent def[initions:]

```
    void f(...) {                        synchronized void f
      synchronized (this) {                body of f
        body of f                        }
      }
    }
```

Communicating the Hard Way

- Communicating data is tricky: the faster party must wait
  slower.

- Obvious approaches for sending data from thread to thre
  work:

```
class DataExchanger {                      DataExchanger exchanger
    Object value = null;                     = new DataExchanger()
    Object receive() {
        Object r; r = null;                -----------------------
        while (r == null)
          { r = value; }                   // thread1 sends to thr
        value = null;                      exchanger.deposit("Hell
        return r;
    }                                      -----------------------
    void deposit(Object data) {
        while (value != null) { }          // thread2 receives fro
        value = data;                      msg = (String) exchange
    }
}
```

- BAD: One thread can monopolize machine while waiting; two
  executing `deposit` or `receive` simultaneously cause chaos.

Primitive Java Facilities

- `wait` method on Object makes thread wait (not using proces
  til notified by `notifyAll`, unlocking the Object while it wait

- Example, `ucb.util.mailbox` has something like this (simplif

```java
interface Mailbox {
  void deposit(Object msg) throws InterruptedException;
  Object receive() throws InterruptedException;
}

class QueuedMailbox implements Mailbox {
  private List<Object> queue = new LinkedList<Object>();

  public synchronized void deposit(Object msg) {
    queue.add(msg);
    this.notifyAll(); // Wake any waiting receivers
  }

  public synchronized Object receive() throws InterruptedEx
    while (queue.isEmpty()) wait();
    return queue.remove(0);
  }
}
```
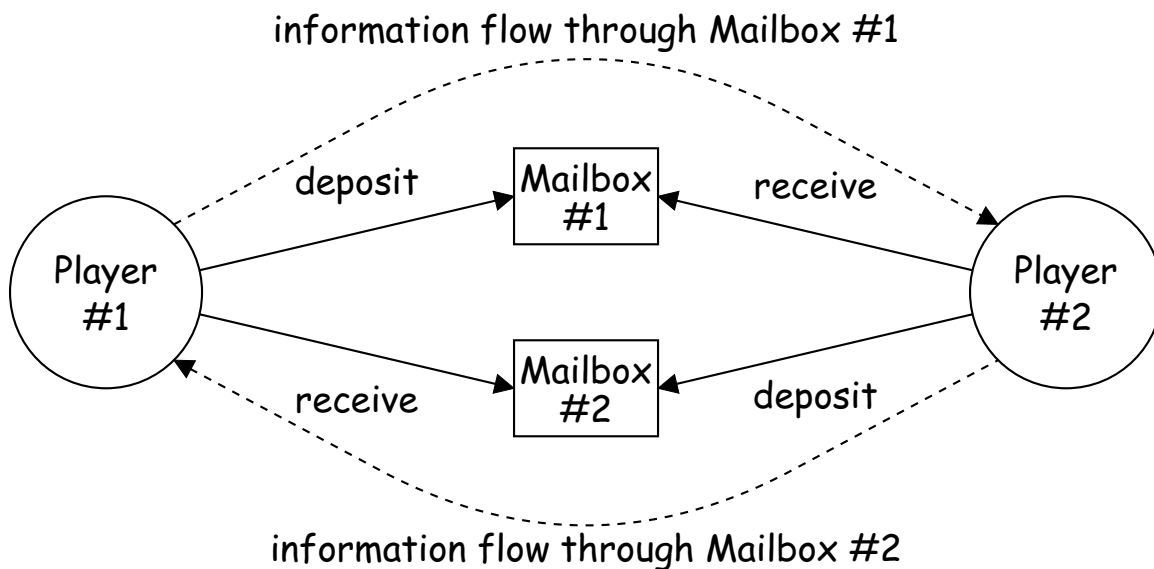
Message-Passing Style

- Use of Java primitives very error-prone. Wait until CS162.

- Mailboxes are higher-level, and allow the following program
  ture:

information flow through Mailbox #1

deposit   Mailbox   receive
          #1

Player   Player
#1   #2

receive   Mailbox   deposit
          #2

information flow through Mailbox #2

- Where each Player is a thread that looks like this:

```
while (! gameOver()) {
  if (myMove())
    outBox.deposit(computeMyMove(lastMove));
  else
    lastMove = inBox.receive();
}
```

More Concurrency

- Previous example can be done other ways, but mechanism
  flexible.

- E.g., suppose you want to think during opponent's move:

```
while (!gameOver()) {
  if (myMove())
    outBox.deposit(computeMyMove(lastMove));
  else {
    do {
      thinkAheadALittle();
      lastMove = inBox.receiveIfPossible();
    } while (lastMove == null);
  }
}
```

- receiveIfPossible (written receive(0) in our actual packag
  wait; returns null if no message yet, perhaps like this:

```
public synchronized Object receiveIfPossible()
    throws InterruptedException {
  if (queue.isEmpty())
    return null;
  return queue.remove(0);
}
```

Coroutines

- A *coroutine* is a kind of synchronous thread that explicitl[y] off control to other coroutines so that only one executes a[t] like Python generators. Can get similar effect with thre[e] mailboxes.

- Example: recursive inorder tree iterator:

```java
class TreeIterator extends Thread {
  Tree root; Mailbox r;
  TreeIterator(Tree T, Mailbox r) {
    this.root = T; this.dest = r;     void treeProcessor(Tree T) {
  }                                      Mailbox m = new QueuedMailb
  public void run() {                    new TreeIterator(T, m).star
    traverse(root);                      while (true) {
    r.deposit(End marker);                 Object x = m.receive();
  }                                        if (x is end marker)
  void traverse(Tree t) {                    break;
    if (t == null) return;                 do something with x;
    traverse(t.left);                    }
    r.deposit(t.label);               }
    traverse(t.right);
  }
}
```

Use In GUIs

- Jave runtime library uses a special thread that does noth
  wait for *events* like mouse clicks, pressed keys, mouse mo
  etc.

- You can designate an object of your choice as a *listener*
  means that Java's event thread calls a method of that objec
  ever an event occurs.

- As a result, your program can do work while the GUI cont
  respond to buttons, menus, etc.

- Another special thread does all the drawing. You don't hav
  aware when this takes place; just ask that the thread wake u
  ever you change something.

Highlights of a GUI Component

```java
/** A widget that draws multi-colored lines indicated by mouse
class Lines extends JComponent implements MouseListener {
  private List<Point> lines = new ArrayList<Point>();

  Lines() {  // Main thread calls this to create one
    setPreferredSize(new Dimension(400, 400));
    addMouseListener(this);
  }
  public synchronized void paintComponent(Graphics g) { // Pai
    g.setColor(Color.white);   g.fillRect(0, 0, 400, 400);
    int x, y;   x = y = 200;
    Color c = Color.black;
    for (Point p : lines)
      g.setColor(c); c = chooseNextColor(c);
      g.drawLine(x, y, p.x, p.y); x = p.x; y = p.y;
    }
  }
  public synchronized void mouseClicked(MouseEvent e) // Event
    { lines.add(new Point(e.getX(), e.getY())); repaint(); }
  ...
}
```

Interrupts

- An *interrupt* is an event that disrupts the normal flow of co
  a program.

- In many systems, interrupts can be totally *asynchronous, o*
  at arbitrary points in a program, the Java developers con
  this unwise; arranged that interrupts would occur only at co
  points.

- In Java programs, one thread can interrupt another to in
  that something unusual needs attention:

  ```
  otherThread.interrupt();
  ```

- But `otherThread` does not receive the interrupt until it waits
  ods `wait`, `sleep` (wait for a period of time), `join` (wait for th
  terminate), and mailbox deposit and receive.

- Interrupt causes these methods to throw `InterruptedEx`
  so typical use is like this:

  ```
  try {
    msg = inBox.receive();
  } catch (InterruptedException e) { HandleEmergency(); }
  ```

Remote Mailboxes (A Side Excursion)

- RMI: Remote Method Interface allows one program to refe
  jects in another program.

- We use it to allow mailboxes in one program be received
  deposited into in another.

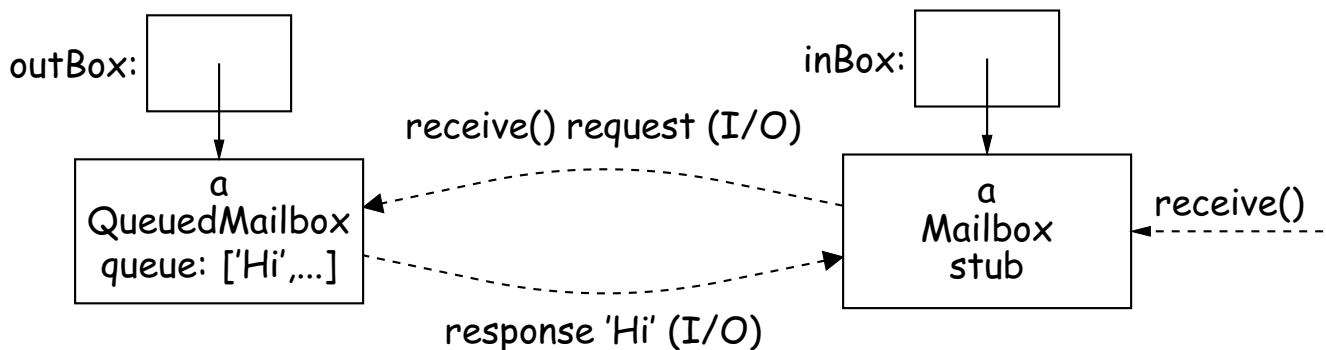- To use this, you define an *interface* to the remote object:

```java
import java.rmi.*;
interface Mailbox extends Remote {
  void deposit(Object msg)
    throws InterruptedException, RemoteException;
  Object receive()
    throws InterruptedException, RemoteException;
  ...
}
```

- On machine that actually will contain the object, you define

```java
class QueuedMailbox ... implements Mailbox {
    Same implementation as before, roughly
}
```

Remote Objects Under the Hood

```
// On machine #1:          // On Machine #2:
Mailbox outBox             Mailbox inBox
   = new QueuedMailbox();     = get outBox from machine #1
```

outBox: ☐

a
QueuedMailbox
queue: ['Hi',...]

receive() request (I/O)

response 'Hi' (I/O)

inBox: ☐

a
Mailbox
stub

receive()

- Because `Mailbox` is an interface, hides fact that on Mach doesn't actually have direct access to it.

- Requests for method calls are relayed by I/O to machine t real object.

- Any argument or return type OK if it also implements Re can be *serialized*—turned into stream of bytes and back, primitive types and String.

- Because I/O involved, expect failures, hence every method c RemoteException (subtype of IOException).