CS61B Lecture #36

Trip: Enumeration types.

er 10, *HFJ*, pp. 489–516.

ation between threads

zation

:

Trip into Java: Enumeration Types

ed a type to represent something that has a few, named,
es.

t form, the only necessary operations are == and !=;
erty of a value of the type is that it differs from all

ions of Java, used named integer constants:

```
Pieces {
_PIECE = 0,    // Fields in interfaces are static final.
_KING = 1,
_PIECE = 2,
_KING = 3,
 = 4;
```

vide *enumeration types* as a shorthand, with syntax like

```
{ BLACK_PIECE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY };
```

these values are basically **ints**, accidents can happen.

Enum Types in Java

of Java allows syntax like that of C or C++, but with
tees:

```
 Piece {
CE, BLACK_KING, WHITE_PIECE, WHITE_KING, EMPTY
```

e as a new reference type, a special kind of class type.

ACK_PIECE, etc., are static, final *enumeration constants*
s) of type PIECE.

tomatically initialized, and are the only values of the
type that exist (illegal to use **new** to create an enum

e ==, and also switch statements:

```
ing(Piece p) {
) {
ACK_KING: case WHITE_KING: return true;
: return false;
```


king Enumerals Available Elsewhere

e BLACK_PIECE are static members of a class, not classes.

nlike C or C++, their declarations are not automatically
e the enumeration class definition.

classes, must write Piece.BLACK_PIECE, which can get

th version 1.5, Java has *static imports:* to import all
tions of class checkers.Piece (including enumerals), you

```
atic checkers.Piece.*;
```

port clauses.

use this for enum classes in the anonymous package.

Operations on Enum Types

laration of enumeration constants significant: .ordinal()
ition (numbering from 0) of an enumeration value. Thus,
_KING.ordinal() is 1.

ece.values() gives all the possible values of the type.
 write:

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

nction Piece.valueOf converts a String into a value of
So Piece.valueOf("EMPTY") == EMPTY.

Fancy Enum Types

asses. You can define all the extra fields, methods, and
 you want.

 are used only in creating enumeration constants. The
arguments follow the constant name:

```
 {
CE(BLACK, false, "b"), BLACK_KING(BLACK, true, "B"),
CE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W"),
ll, false, " ");

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

de color, boolean isKing, String textName) {
olor = color; this.isKing = isKing; this.textName = textName;

or() { return color; }
isKing() { return isKing; }
extName() { return textName; }
```

Threads

r programs consist of single sequence of instructions.

equence is called a *thread* (for "thread of control") in

s programs containing *multiple* threads, which (concep-
oncurrently.

a uniprocessor, only one thread at a time actually runs,
wait, but this is largely invisible.

gram access to threads, Java provides the type Thread
. Each Thread contains information about, and controls,

s access to data from two threads can cause chaos, so
structs for controlled communication, allowing threads
cts, to *wait* to be notified of events, and to *interrupt*
ds.

But Why?

programs always have > 1 thread: besides the main
ers clean up garbage objects, receive signals, update
other stuff.

ms deal with asynchronous events, is sometimes conve-
anize into subprograms, one for each independent, re-
ce of events.

w us to insulate one such subprogram from another.

organized like this: application is doing some compu-
O, another thread waits for mouse clicks (like 'Stop'),
attention to updating the screen as needed.

s like search engines may be organized this way, with
er request.

se, sometimes we *do* have a real multiprocessor.

Java Mechanics

e actions "walking" and "chewing gum":

```
1 implements Runnable {
id run()
 (true) ChewGum(); }

1 implements Runnable {
id run()
 (true) Walk(); }
```

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

rnative (uses fact that Thread implements Runnable):

```
 extends Thread {
 run()
true) ChewGum(); }

 extends Thread {
 run()
true) Walk(); }
```

```
Thread chomp = new Chewer2(),
       clomp = new Walker2();
chomp.start();
clomp.start();
```


Avoiding Interference

read has data for another, one must wait for the other

wo threads use the same data structure, generally only
odify it at a time; other must wait.

ould happen if two threads simultaneously inserted an
nked list at the same point in the list?

d conceivably execute

```
ew ListCell(x, p.next);
```

e values of p and p.next; one insertion is lost.

for only one thread at a time to execute a method on a
ject with either of the following equivalent definitions:

```
) {
ized (this) {
f f
```

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


Communicating the Hard Way

ng data is tricky: the faster party must wait for the

oaches for sending data from thread to thread don't

```
hanger {
lue = null;
ceive() {
 r; r = null;
(r == null)
 value; }
 null;
 r;

sit(Object data) {
(value != null) { }
 data;
```

```
DataExchanger exchanger
  = new DataExchanger();

--------------------------------

// thread1 sends to thread2 with
exchanger.deposit("Hello!");

--------------------------------

// thread2 receives from thread1 with
msg = (String) exchanger.receive();
```

read can monopolize machine while waiting; two threads
posit or receive simultaneously cause chaos.

Primitive Java Facilities

on Object makes thread wait (not using processor) un-
y notifyAll, unlocking the Object while it waits.

.util.mailbox has something like this (simplified):

```
Mailbox {
osit(Object msg) throws InterruptedException;
eceive() throws InterruptedException;

edMailbox implements Mailbox {
List<Object> queue = new LinkedList<Object>();

ynchronized void deposit(Object msg) {
add(msg);
otifyAll(); // Wake any waiting receivers

ynchronized Object receive() throws InterruptedException {
(queue.isEmpty()) wait();
 queue.remove(0);
```

More Concurrency

mple can be done other ways, but mechanism is very

you want to think during opponent's move:

```
neOver()) {
ve())
.deposit(computeMyMove(lastMove));


kAheadALittle();
Move = inBox.receiveIfPossible();
e (lastMove == null);
```

ssible (written receive(0) in our actual package) doesn't
; null if no message yet, perhaps like this:

```
chronized Object receiveIfPossible()
InterruptedException {
e.isEmpty())
 null;
ueue.remove(0);
```


Use In GUIs

e library uses a special thread that does nothing but
nts like mouse clicks, pressed keys, mouse movement,

ignate an object of your choice as a *listener;* which
Java's event thread calls a method of that object when-
t occurs.

your program can do work while the GUI continues to
uttons, menus, etc.

cial thread does all the drawing. You don't have to be
this takes place; just ask that the thread wake up when-
nge something.

Interrupts

is an event that disrupts the normal flow of control of

ems, interrupts can be totally *asynchronous,* occurring
points in a program, the Java developers considered
arranged that interrupts would occur only at controlled

grams, one thread can interrupt another to inform it
ing unusual needs attention:

```
d.interrupt();
```

read does not receive the interrupt until it waits: meth-
eep (wait for a period of time), join (wait for thread to
and mailbox deposit and receive.

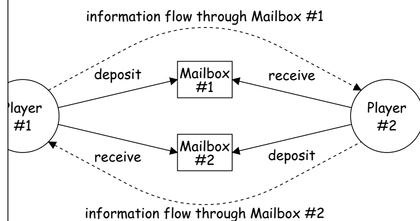
uses these methods to throw InterruptedException,
e is like this:

```
Box.receive();
nterruptedException e) { HandleEmergency(); }
```


Message-Passing Style

primitives very error-prone. Wait until CS162.

e higher-level, and allow the following program struc-



Player is a thread that looks like this:

```
ameOver()) {
ve())
.deposit(computeMyMove(lastMove));

ve = inBox.receive();
```


Coroutines

is a kind of synchronous thread that explicitly hands
o other coroutines so that only one executes at a time,
generators. Can get similar effect with threads and

ursive inorder tree iterator:

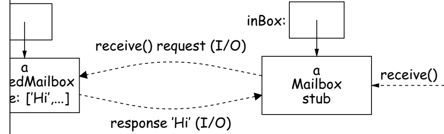
```
or extends Thread {
lbox r;
ree T, Mailbox r) {
T; this.dest = r;    void treeProcessor(Tree T) {
                       Mailbox m = new QueuedMailbox();
n() {                  new TreeIterator(T, m).start();
t);                    while (true) {
d marker);                Object x = m.receive();
                          if (x is end marker)
Tree t) {                    break;
l) return;                do something with x;
eft);                  }
label);             }
ight);
```


Highlights of a GUI Component

```
that draws multi-colored lines indicated by mouse. */
tends JComponent implements MouseListener {
<Point> lines = new ArrayList<Point>();

/ Main thread calls this to create one
edSize(new Dimension(400, 400));
stener(this);

ronized void paintComponent(Graphics g) { // Paint thread
(Color.white);   g.fillRect(0, 0, 400, 400);
  x = y = 200;
Color.black;
 p : lines)
or(c); c = chooseNextColor(c);
ne(x, y, p.x, p.y); x = p.x; y = p.y;


ronized void mouseClicked(MouseEvent e) // Event thread
d(new Point(e.getX(), e.getY())); repaint(); }
```

Remote Objects Under the Hood

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



lbox is an interface, hides fact that on Machine #2 ally have direct access to it.

method calls are relayed by I/O to machine that has

nt or return type OK if it also implements Remote or alized—turned into stream of bytes and back, as can es and String.

involved, expect failures, hence every method can throw ption (subtype of IOException).

mote Mailboxes (A Side Excursion)

e Method Interface allows one program to refer to ob- her program.

allow mailboxes in one program be received from or to in another.

you define an *interface* to the remote object:

```
a.rmi.*;
Mailbox extends Remote {
osit(Object msg)
 InterruptedException, RemoteException;
eceive()
 InterruptedException, RemoteException;
```

that actually will contain the object, you define

```
edMailbox ... implements Mailbox {
mplementation as before, roughly
```