

Trip into Java: Enumeration Types

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

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

sions of Java, used named integer constants:

```
Pieces {
BLACK_PIECE = 0,    // Fields in interfaces are static final.
BLACK_KING = 1,
WHITE_PIECE = 2,
WHITE_KING = 3,
EMPTY = 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.

18:46 2017

CS61B: Lecture #36 2

CS61B Lecture #36

Trip: Enumeration types.

er 10, *HFJ*, pp. 489-516.

ation between threads

zation

18:46 2017

CS61B: Lecture #36 1

king Enumerals Available Elsewhere

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

nlike C or C++, their declarations are not automatically le 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

```
static checkers.Piece.*;
```

port clauses.

use this for enum classes in the anonymous package.

18:46 2017

CS61B: Lecture #36 4

Enum Types in Java

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

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

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

BLACK_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

se ==, and also switch statements:

```
isKing(Piece p) {
p) {
BLACK_KING: case WHITE_KING: return true;
: return false;
```

18:46 2017

CS61B: Lecture #36 3

Fancy Enum Types

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

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

```
{
BLACK_PIECE(BLACK, false, "b"), BLACK_KING(BLACK, true, "B"),
WHITE_PIECE(WHITE, false, "w"), WHITE_KING(WHITE, true, "W"),
EMPTY, false, " ");

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

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

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

18:46 2017

CS61B: Lecture #36 6

Operations on Enum Types

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

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

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

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

18:46 2017

CS61B: Lecture #36 5

But Why?

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

Programs deal with asynchronous events, is sometimes convenient to organize into subprograms, one for each independent, receive of events.

How do we insulate one such subprogram from another.

Can be organized like this: application is doing some computation, another thread waits for mouse clicks (like 'Stop'), pays attention to updating the screen as needed.

Search engines like search engines may be organized this way, with separate request.

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

Avoiding Interference

If a thread has data for another, one must wait for the other.

If two threads use the same data structure, generally only one can modify it at a time; other must wait.

Could happen if two threads simultaneously inserted an element into a linked list at the same point in the list?

Could conceivably execute

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

using the values of p and p.next; one insertion is lost.

Can be avoided for only one thread at a time to execute a method on an object with either of the following equivalent definitions:

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

Primitive Java Facilities

wait() on Object makes thread wait (not using processor) until notified, notifyAll(), unlocking the Object while it waits.

java.util.concurrent.locks.Lock has something like this (simplified):

```
Mailbox {
    Object receive() throws InterruptedException;
}

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

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

    synchronized Object receive() throws InterruptedException {
        while (queue.isEmpty()) wait();
        return queue.remove(0);
    }
}
```

Threads

Most programs consist of single sequence of instructions. A sequence is called a *thread* (for "thread of control") in multi-threaded programs.

Multi-threaded programs containing *multiple* threads, which (conceptually) run concurrently.

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

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

Simultaneous access to data from two threads can cause chaos, so Java provides constructs for controlled communication, allowing threads to wait to be notified of events, and to interrupt other threads.

Java Mechanics

Two actions "walking" and "chewing gum":

```
class Chewer1 implements Runnable {
    // Walk and chew gum
    Thread chomp;
    public run() {
        (true) ChewGum();
    }
}

class Walker1 implements Runnable {
    Thread clomp;
    public run() {
        (true) Walk();
    }
}
```

Runnable (uses fact that Thread implements Runnable):

```
class Chewer2 extends Thread {
    public run() {
        (true) ChewGum();
    }
}

class Walker2 extends Thread {
    public run() {
        (true) Walk();
    }
}
```

Communicating the Hard Way

Sending data is tricky: the faster party must wait for the slower.

Approaches for sending data from thread to thread don't work.

```
class DataExchanger {
    DataExchanger() {
        // ...
    }
}

class Thread1 {
    DataExchanger exchanger;
    public run() {
        // ...
        exchanger.deposit("Hello!");
    }
}

class Thread2 {
    DataExchanger exchanger;
    public run() {
        // ...
        exchanger.receive();
    }
}
```

Thread can monopolize machine while waiting; two threads can't deposit or receive simultaneously cause chaos.

More Concurrency

Example can be done other ways, but mechanism is very

you want to think during opponent's move:

```
moveOver() {  
    receive()  
    deposit(computeMyMove(lastMove));  
}
```

```
moveAheadALittle();  
Move = inBox.receiveIfPossible();  
if (lastMove == null);
```

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

```
ynchronized Object receiveIfPossible()  
InterruptedException {  
    boolean isEmpty()  
    Object move;  
    queue.remove(0);  
}
```

18:46 2017

CS61B: Lecture #36 14

Use In GUIs

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

designate an object of your choice as a *listener*; which Java's event thread calls a method of that object when it occurs.

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

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

18:46 2017

CS61B: Lecture #36 16

Interrupts

InterruptedException is an event that disrupts the normal flow of control of

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

programs, one thread can interrupt another to inform it of something unusual needs attention:

```
thread.interrupt();
```

thread does not receive the interrupt until it waits: method sleep (wait for a period of time), join (wait for thread to finish) and mailbox deposit and receive.

programs use these methods to throw InterruptedException, InterruptedException is like this:

```
Mailbox.receive();  
InterruptedException e { HandleEmergency(); }
```

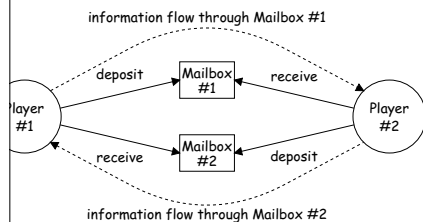
18:46 2017

CS61B: Lecture #36 18

Message-Passing Style

Primitives very error-prone. Wait until CS162.

Higher-level, and allow the following program structure:



Player is a thread that looks like this:

```
moveOver() {  
    receive()  
    deposit(computeMyMove(lastMove));  
}
```

```
move = inBox.receive();
```

18:46 2017

CS61B: Lecture #36 13

Coroutines

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

recursive inorder tree iterator:

```
Coroutine extends Thread {  
    Mailbox r;  
    Tree T, Mailbox r {  
        T; this.dest = r;  
        void treeProcessor(Tree T) {  
            Mailbox m = new QueuedMailbox();  
            new TreeIterator(T, m).start();  
            while (true) {  
                Object x = m.receive();  
                if (x is end marker)  
                    break;  
                Tree t {  
                    t) return;  
                left);  
                label);  
                right);  
            }  
            do something with x;  
        }  
    }  
}
```

18:46 2017

CS61B: Lecture #36 15

Highlights of a GUI Component

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

```
/* Main thread calls this to create one  
Dimension(400, 400));  
MouseListener(this);
```

```
ynchronized void paintComponent(Graphics g) { // Paint thread  
    g.setColor(Color.white); g.fillRect(0, 0, 400, 400);  
    x = y = 200;  
    Color.black;  
    p : lines  
    for (c : lines) {  
        chooseNextColor(c);  
        g.setColor(c); g.fillRect(x, y, p.x, p.y); x = p.x; y = p.y;  
    }  
}
```

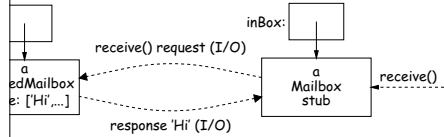
```
ynchronized void mouseClicked(MouseEvent e) // Event thread  
    add(new Point(e.getX(), e.getY())); repaint(); }
```

18:46 2017

CS61B: Lecture #36 17

Remote Objects Under the Hood

```
#1:          // On Machine #2:  
Mailbox mailbox;  
Mailbox mailbox = get outBox from machine #1
```



Mailbox is an interface, hides fact that on Machine #2
client may have direct access to it.

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

object or return type OK if it also implements Remote or
Serialized—turned into stream of bytes and back, as can
be done with Serializable and String.

When using Remote, expect failures, hence every method can throw
IOException (subtype of IOException).

18:46 2017

CS61B: Lecture #36 20

Note Mailboxes (A Side Excursion)

The Remote Method Interface allows one program to refer to ob-
ject in another program.

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

When you define an interface to the remote object:

```
import java.rmi.*;  
Mailbox extends Remote {  
    Object receive()  
    Object receive()  
    Object receive()  
}
```

When that actually will contain the object, you define

```
RemoteMailbox ... implements Mailbox {  
    // implementation as before, roughly
```

18:46 2017

CS61B: Lecture #36 19