Lecture #37

xcursions into nitty-gritty stuff: Threads, storage man-

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*
ls.

But Why?

programs always have > 1 thread: besides the main
ners 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

roaches for sending data from thread to thread don't

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

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

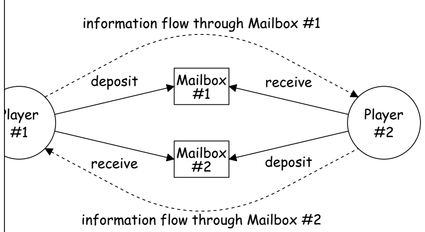
```
DataExchanger exchanger
  = new DataExchanger();

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

// 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.

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;

n() {
t);
d marker);

Tree t) {
l) return;
eft);
label);
ight);
```

```
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;
      do something with x;
   }
}
```


Highlights of a GUI Component

```
hat 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(); }
```


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:

```
meOver()) {
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 nd mailbox deposit and receive.

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

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


mote Mailboxes (A Side Excursion)

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

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

you define an *interface* to the remote object:

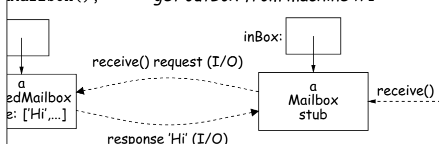
```
a.rmi.*;
Mailbox extends Remote {
osit(Object msg)
 InterruptedException, RemoteException;
eceive()
 InterruptedException, RemoteException;
```

that actually will contain the object, you define

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


Remote Objects Under the Hood

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



lbox is an interface, hides fact that on Machine #2 lly 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 *lized*—turned into stream of bytes and back, as can es and String.

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

Scope and Lifetime

eclaration is portion of program text to which it applies

be contiguous.

s static: independent of data.

extent of storage is portion of program execution dur- exists.

ntiguous

 dynamic: depends on data

xtent:

ntire duration of program

utomatic: duration of call or block execution (local vari-

 From time of allocation statement (**new**) to dealloca- y.

Explicit vs. Automatic Freeing

explicit means to free dynamic storage.

hen no expression in any thread can possibly be influ- change an object, it might as well not exist:

```
steful()

c = new IntList(3, new IntList(4, null));
.tail;
ble c now deallocated, so no way
t to first cell of list
```

t, Java runtime, like Scheme's, recycles the object c *arbage collection*.

Under the Hood: Allocation

s (references) are represented as integer addresses.

 to machine's own practice.

not convert integers ↔ pointers,

parts of Java runtime implemented in C, or sometimes e, where you can.

tor in C:

```
[STORAGE_SIZE];  // Allocated array
inder = STORAGE_SIZE;

ter to a block of at least N bytes of storage */
leAlloc(size_t n) { // void*: pointer to anything
remainder) ERROR();
r = (remainder - n) & ~0x7;  // Make multiple of 8
void*) (store + remainder);
```

Explicit Deallocating

lly require explicit deallocation, because of

ın-time information about what is array

/ of converting pointers to integers.

ın-time information about *unions:*

```
Various {
Int;
* Pntr;
le Double;
// X is either an int, char*, or double
```

ıll three problems; automatic collection possible.

ing can be somewhat faster, but rather error-prone:

orruption

eaks

Free List Strategies

uests generally come in multiple sizes.

ks on the free list are big enough, and one may have to chunk and break it up if too big.

tegies to find a chunk that fits have been used:

l fits:

ocks in LIFO or FIFO order, or sorted by address.

e adjacent blocks.

for *first fit* on list, *best fit* on list, or *next fit* on list ıst-chosen chunk.

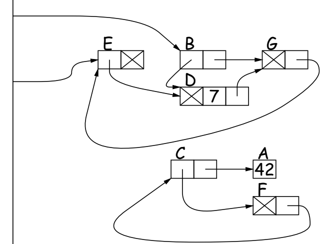
ed fits: separate free lists for different chunk sizes.

stems: A kind of segregated fit where some newly ad- ee blocks of one size are easily detected and combined r chunks.

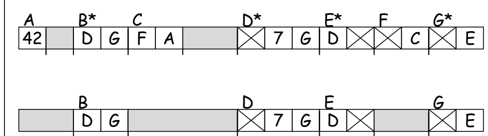
ocks reduces *fragmentation* of memory into lots of lit- d chunks.

rbage Collection: Mark and Sweep

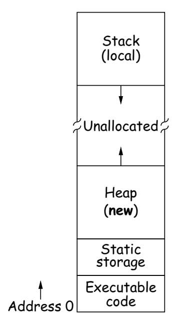
atics)



1. Traverse and mark graph of objects.
2. Sweep through memory, freeing unmarked objects.



xample of Storage Layout: Unix



/ to turn chunks of unallocated region into heap.

omatically for stack.

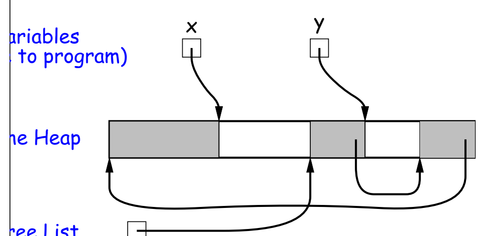
Free Lists

cator grabs chunks of storage from OS and gives to

ycled storage, when available.

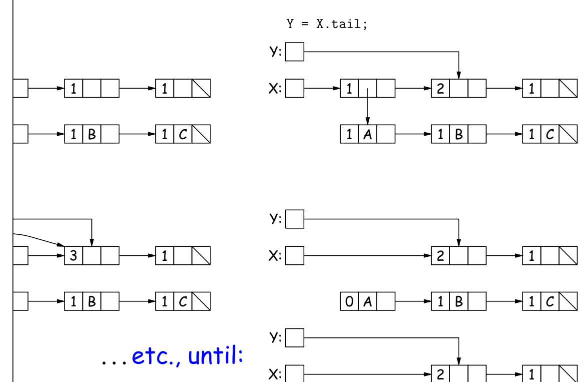
e is freed, added to a *free list* data structure to be

or explicit freeing and some kinds of automatic garbage



age Collection: Reference Counting

count of number of pointers to each object. Release oes to 0.



Cost of Mark-and-Sweep

eep algorithms don't move any exisiting objects—pointers
e.

ount of work depends on the amount of memory swept—
l amount of active (non-garbage) storage + amount of
t necessarily a big hit: the garbage had to be active at
d hence there was always some "good" processing in the
h byte of garbage scanned.

Copying Garbage Collection

roach: *copying garbage collection* takes time propor-
unt of active storage:

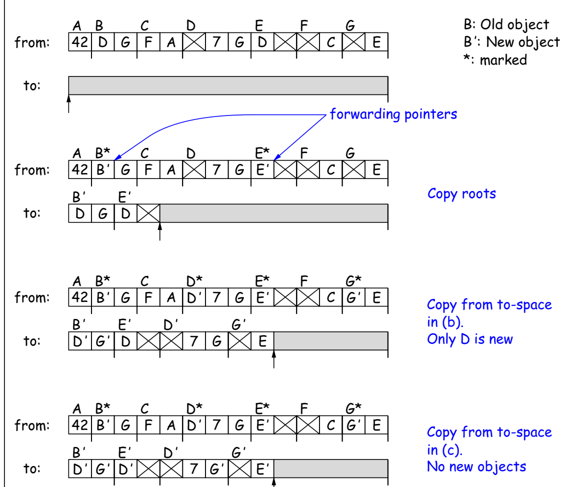
the graph of active objects breadth first, *copying* them
e contiguous area (called "to-space").

py each object, mark it and put a *forwarding pointer*
t points to where you copied it.

time you have to copy an already marked object, just
rwarding pointer instead.

e, the space you copied from ("from-space") becomes
o-space; in effect, all its objects are freed in constant

ying Garbage Collection Illustrated



jects Die Young: Generational Collection

bjects stay active, and need not be collected.

e to avoid copying them over and over.

garbage collection schemes have two (or more) from
for newly created objects (*new space*) and one for
jects that have survived garbage collection (*old space*).

bage collection collects only in new space, ignores point-
to old space, and moves objects to old space.

s usual roots plus pointers in old space that have changed
might be pointing to new space).

ace full, collect all spaces.

h leads to much smaller *pause times* in interactive sys-

There's Much More

st highlights.

on how to implement these ideas efficiently.

garbage collection: What if objects scattered over many

llection: where predictable pause times are important,
emental collection, doing a little at a time.