Storage Classes
**Stack**: objects created/destroyed in last-in, first-out order

**Heap**: objects created/destroyed in any order; automatic garbage collection optional

**Static**: objects allocated at compile time; persist throughout run
Static Objects

```java
class Example {
    public static final int a = 3;

    public void hello() {
        System.out.println("Hello");
    }
}
```

Examples

- Static class variable
- String constant “Hello”
- Information about the Example class

Advantages

- Zero-cost memory management
- Often faster access (address a constant)
- No out-of-memory danger

Disadvantages

- Size and number must be known beforehand
- Wasteful
The Stack and Activation Records
Idea: some objects persist from when a procedure is called to when it returns.

Naturally implemented with a stack: linear array of memory that grows and shrinks at only one boundary.

Natural for supporting recursion.

Each invocation of a procedure gets its own frame (activation record) where it stores its own local variables and bookkeeping information.
An Activation Record: The State Before Calling \texttt{bar}

\begin{verbatim}
int foo(int a, int b) {
    int c, d;
    bar(1, 2, 3);
}
\end{verbatim}
Recursive Fibonacci

(Real C)
```c
int fib(int n) {
    if (n<2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

(Assembly-like C)
```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```

fib(3)
```
```
fib(2)fib(1)
```
```
fib(1)fib(0)```

Executing fib(3)
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
    L1: tmp1 = n - 1;
        tmp2 = fib(tmp1);
    L2: tmp1 = n - 2;
        tmp3 = fib(tmp1);
    L3: tmp1 = tmp2 + tmp3;
        return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```c
int fib(int n) {
    int tmp1, tmp2, tmp3;
    tmp1 = n < 2;
    if (!tmp1) goto L1;
    return 1;
L1: tmp1 = n - 1;
    tmp2 = fib(tmp1);
L2: tmp1 = n - 2;
    tmp3 = fib(tmp1);
L3: tmp1 = tmp2 + tmp3;
    return tmp1;
}
```
Local arrays with fixed size are easy to stack.

```c
void foo()
{
    int a;
    int b[10];
    int c;
}
```

<table>
<thead>
<tr>
<th>return address</th>
<th>← FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b[9]</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>b[0]</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>← FP − 48</td>
</tr>
</tbody>
</table>
Variable-sized local arrays aren’t as easy.

```c
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```

Doesn’t work: generated code expects a fixed offset for c. Even worse for multi-dimensional arrays.
As always: 
add a level of indirection

```c
void foo(int n) {
    int a;
    int b[n];
    int c;
}
```

Variables remain constant offset from frame pointer.
Implementing Nested Functions with Access Links

```ocaml
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) (* b *)
  in
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

```
(access link)

a:  
x = 5
s = 42
```

let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
    let e q = b (q+1) in
    e (x+1) (* a *)

What does “a 5 42” give?
Implementing Nested Functions with Access Links

```latex
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  in
  let e q = b (q+1) in
  e (x+1) (* a *)
```

What does “a 5 42” give?

<table>
<thead>
<tr>
<th>a:</th>
<th>x = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s = 42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b:</th>
<th>y = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>e:</td>
<td>q = 6</td>
</tr>
</tbody>
</table>
Implementing Nested Functions with Access Links

```
let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
d (y+1) in (* b *)
  let e q = b (q+1) in
e (x+1) (* a *)
```

What does “a 5 42” give?

```
a:
  x = 5
  s = 42
b:
  y = 7
c:
  z = z + s
  w = c (w+1)
d:
  y = 7
  w = 8
e:
  q = 6
```

Implementing Nested Functions with Access Links

let a x s =
  let b y =
    let c z = z + s in
    let d w = c (w+1) in
    d (y+1) in (* b *)
  let e q = b (q+1) in
  e (x+1) (* a *)

What does “a 5 42” give?