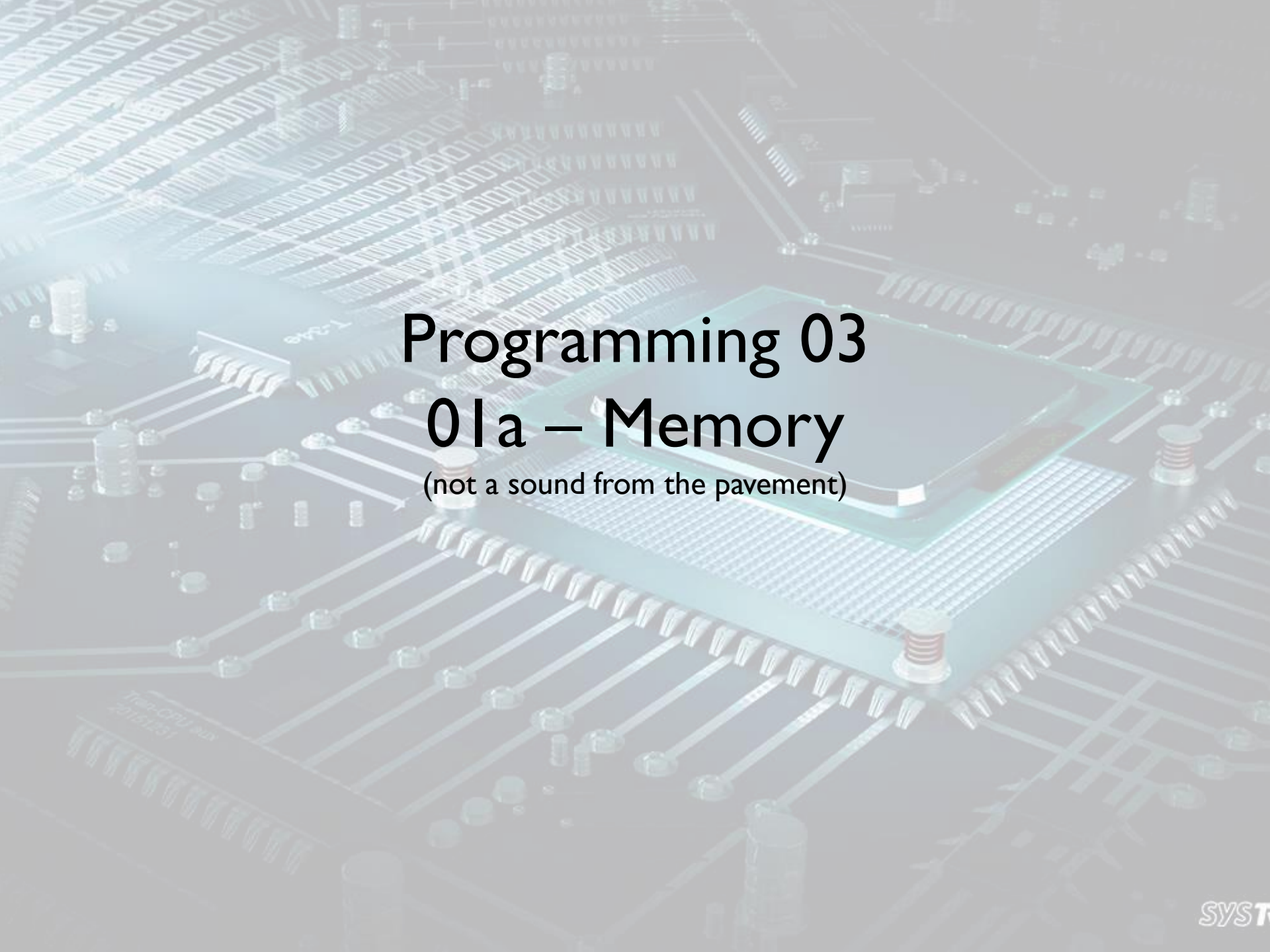
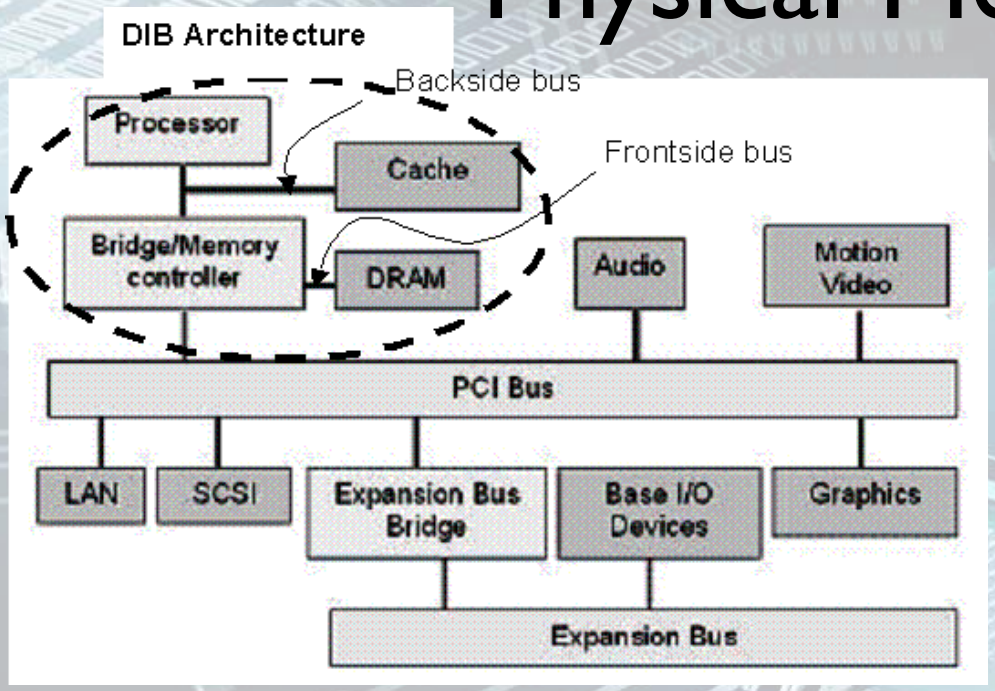
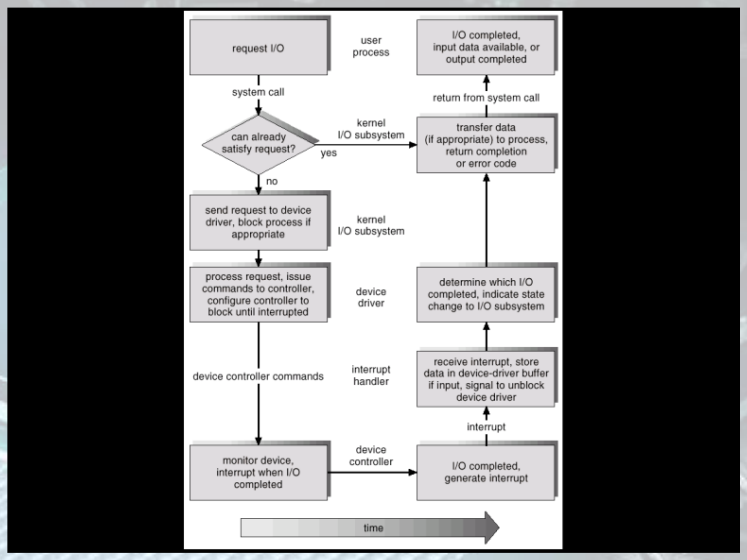
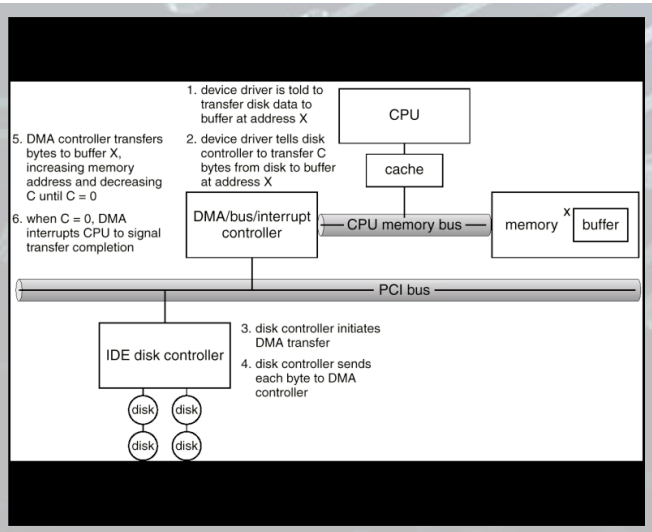


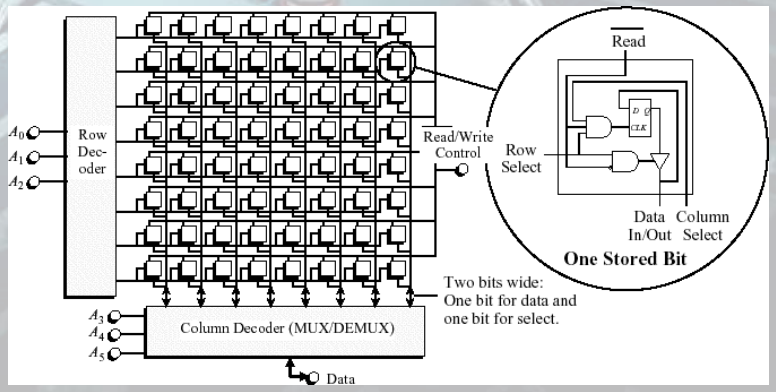
Programming 03
01a – Memory
(not a sound from the pavement)

Physical Memory



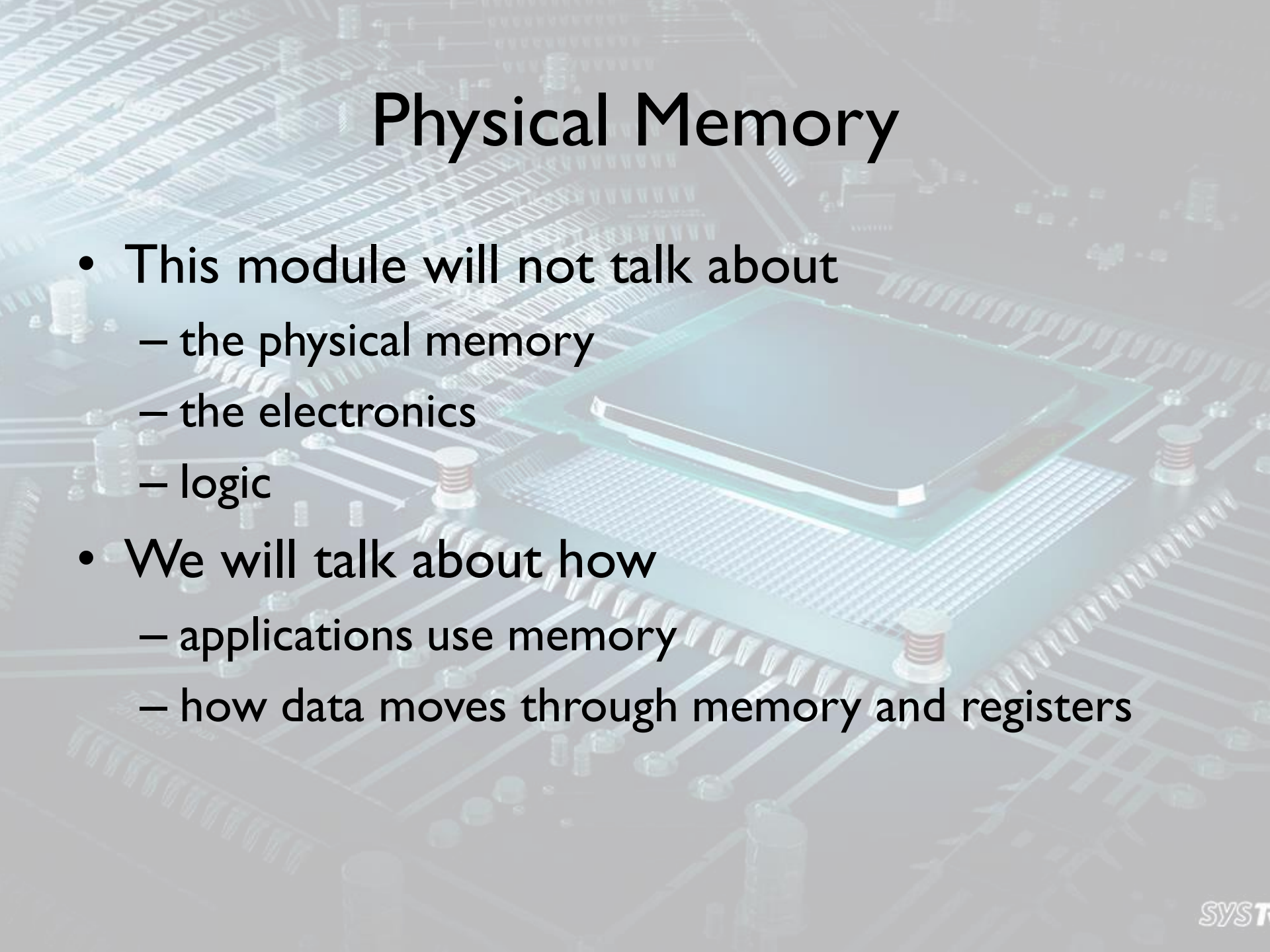






Physical Memory

- This module will not talk about
  - the physical memory
  - the electronics
  - logic
- We will talk about how
  - applications use memory
  - how data moves through memory and registers



Java & Memory

- Java Virtual Machine has four sections
  - The **code** section contains your **bytecode** (the compiled version of your code)
  - The **Stack** section of memory contains **methods, local variables, and reference variables.**
  - The **Heap** section contains **Objects** (may also contain reference variables).
  - The **Static** section contains **Static data/methods**.

Java Primitive Variables

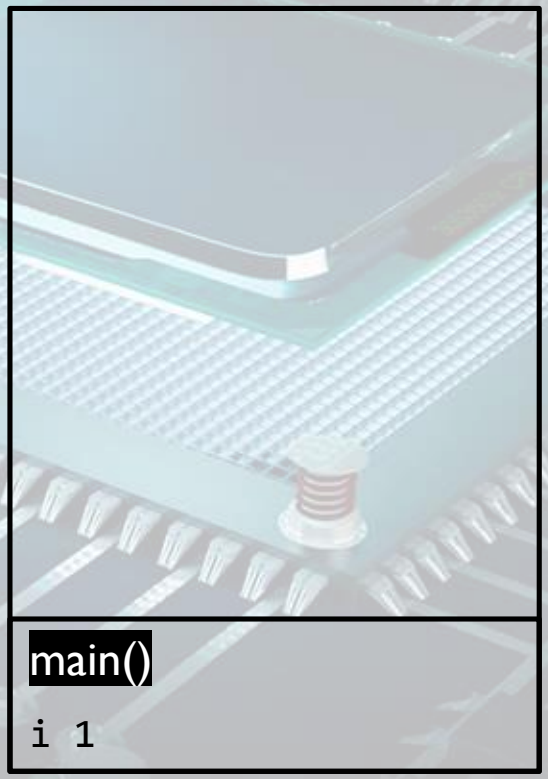
- Primitive variables
  - int, long, float, double, char, boolean, byte, short
- are stored within **frames** on the **stack**
- A new frame is created and added to the stack each time a method is called
- and deleted from the stack when the method ends

A Simple Java Application

CODE

```
public static void main(String[] args)
{
   int i=1;
}
```

STACK



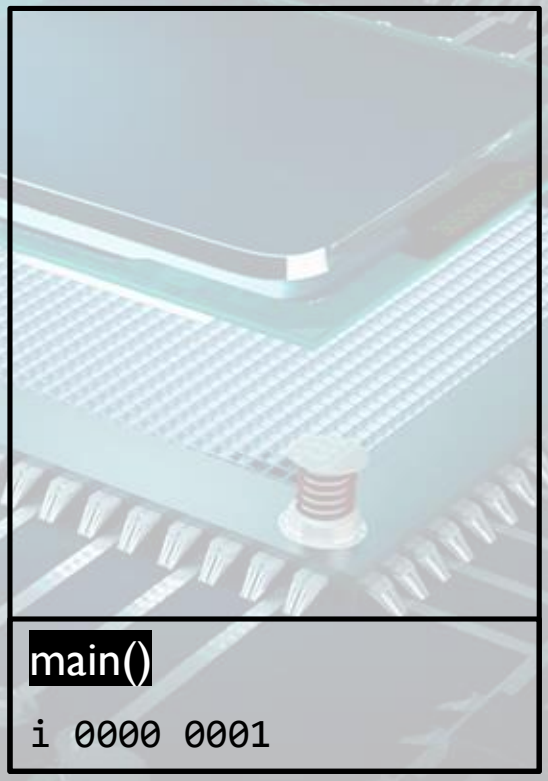
- A **frame** is created on the stack
- The stack is filled from zero upwards
- Within the main() frame all local variables are stored

A Simple Java Application

CODE

```
public static void main(String[] args)
{
   int i=1;
}
```

STACK



- All data is stored in binary
- (demonstrated with 1byte / 8 bits. Actual amount varies by chip)

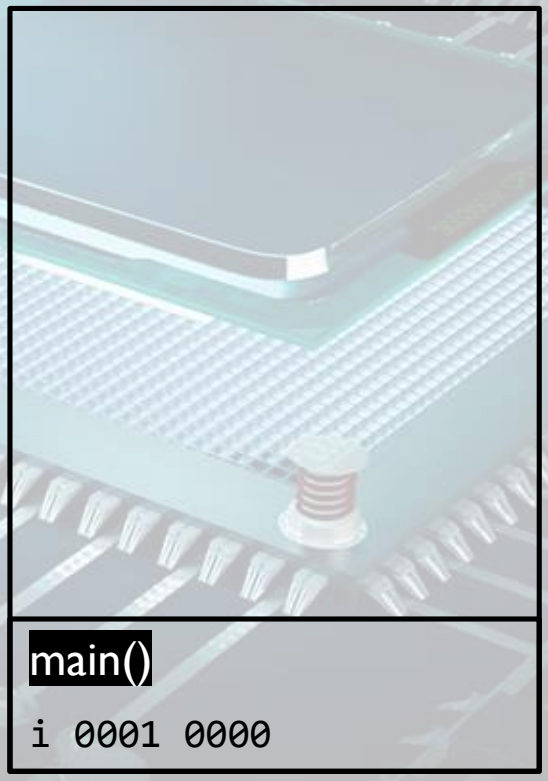
A Simple Java Application

CODE

```
public static void main(String[] args)
{
   int i=1;
}
```

- All data is stored in binary
- It is stored least significant "part" first
  - So an 8 bit number is stored as two four bit numbers with the smallest (least significant) first

STACK



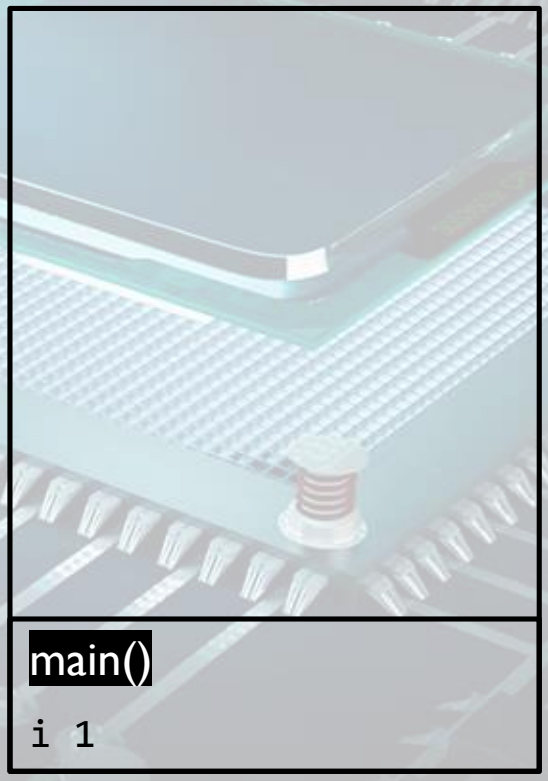
A Simple Java Application

CODE

```
public static void main(String[] args)
{
   int i=1;
}
```

- For simplicity I'll stick to numbers here
- But when we get to Assembly level we'll have to pay attention to this.

STACK



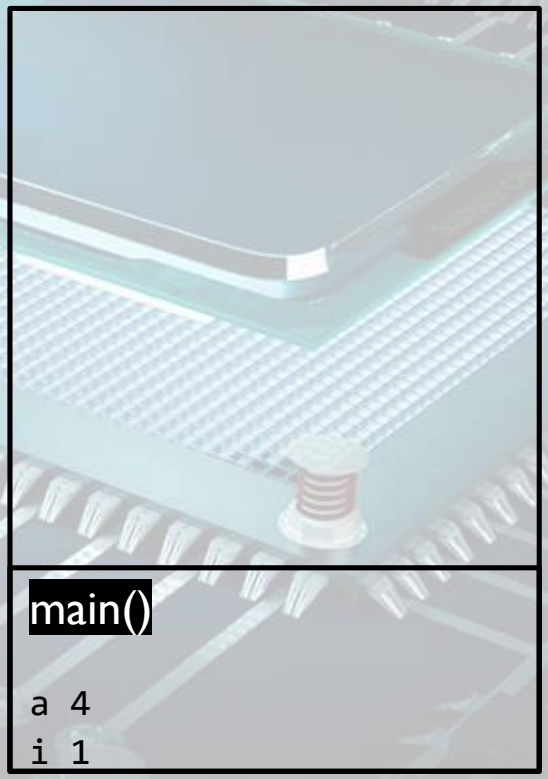
A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;
  int a=4;
}
```

- Each frame fills from the first memory address

STACK

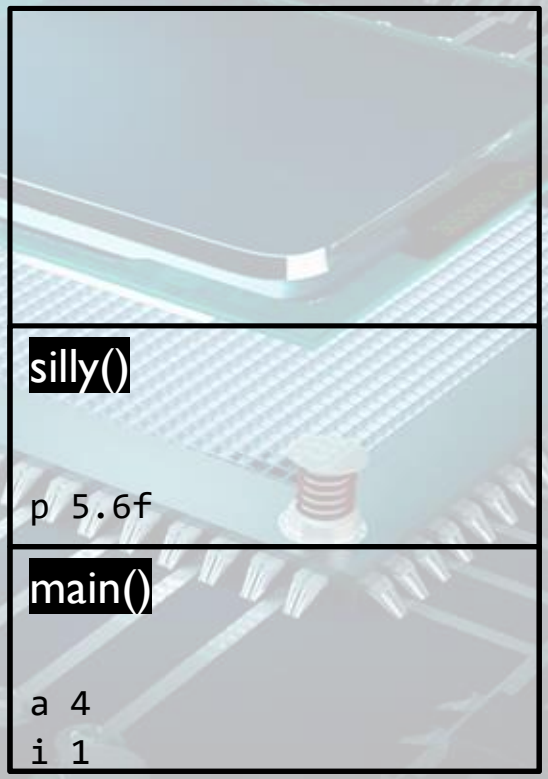


A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;
  int a=4;

  silly();
}

void silly()
{
  float p=5.6f;
}
```

STACK



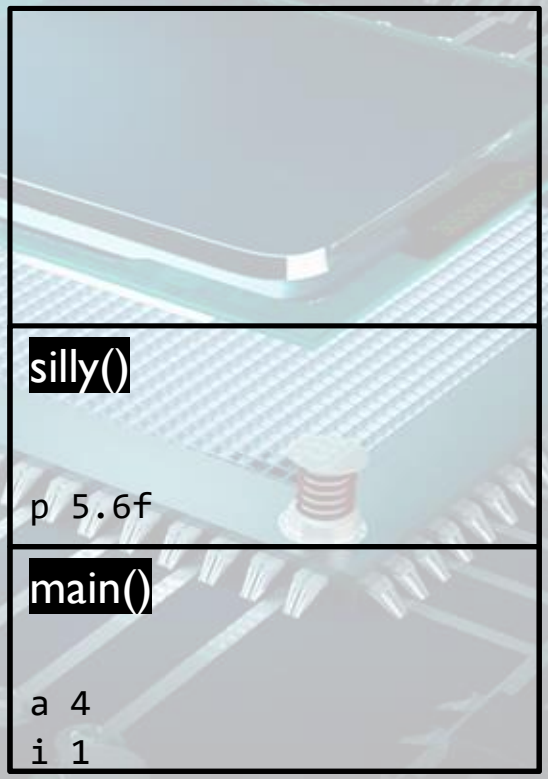
- When a method is **called** a new frame is added to the stack to store the local variables of that method

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;
  int a=4;

  silly();
}

void silly()
{
  float p=5.6f;
}
```

STACK



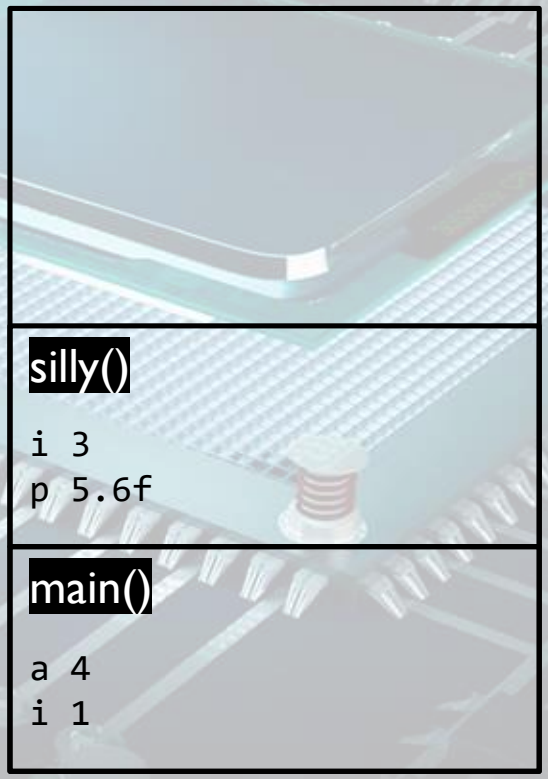
- A method can only access its own frame
- **Silly()** cannot use the i and a variables from main()
- A **frame** limits the **scope** of a variable

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;
  int a=4;

  silly();
}

void silly()
{
  float p=5.6f;
  int i=3;
}
```

STACK



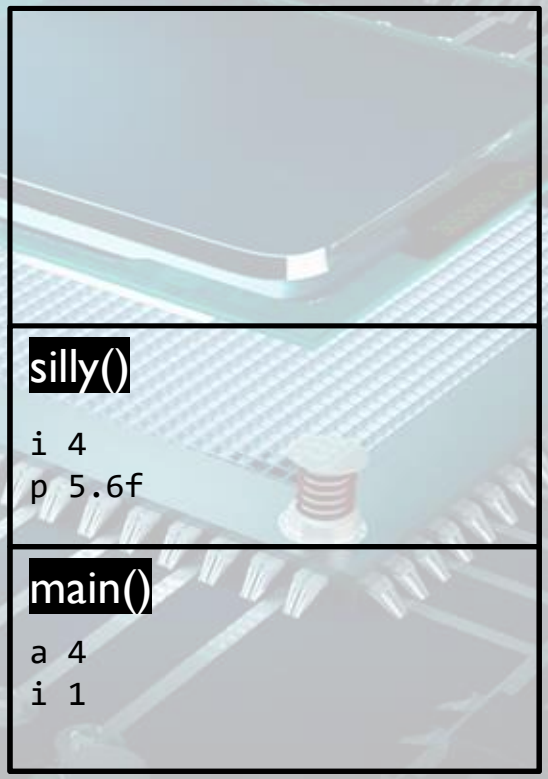
- This is why we can use the same variable name in different methods without changing the value everywhere

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;
  int a=4;

  silly();
}

void silly()
{
  float p=5.6f;
  int i=3;
  i++;
}
```

STACK



- This is why we can use the same variable name in different methods without changing the value everywhere

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  silly(i);
}

void silly(int locali)
{
}
```

STACK

- This is also why we need to pass values as parameters if we want to use them in the method

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  silly(i);
}

void silly(int locali)
{
  locali = 3;
}
```

STACK

- Note that changes to a parameter in a method do not change the value in the calling method

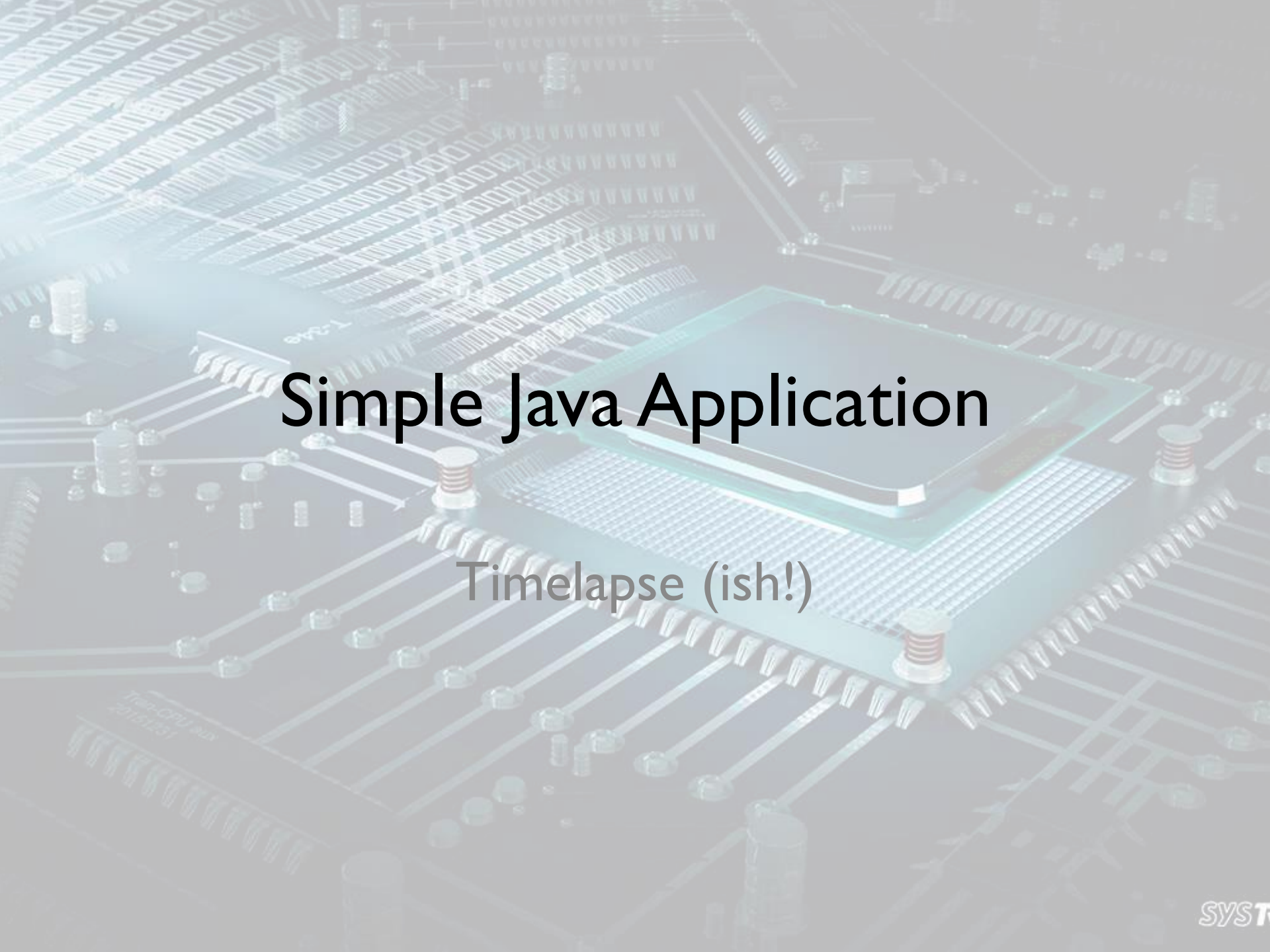
A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK

- Similarly if we want to pass back a value to the calling method we have to use a return to do so



Simple Java Application

Timelapse (ish!)

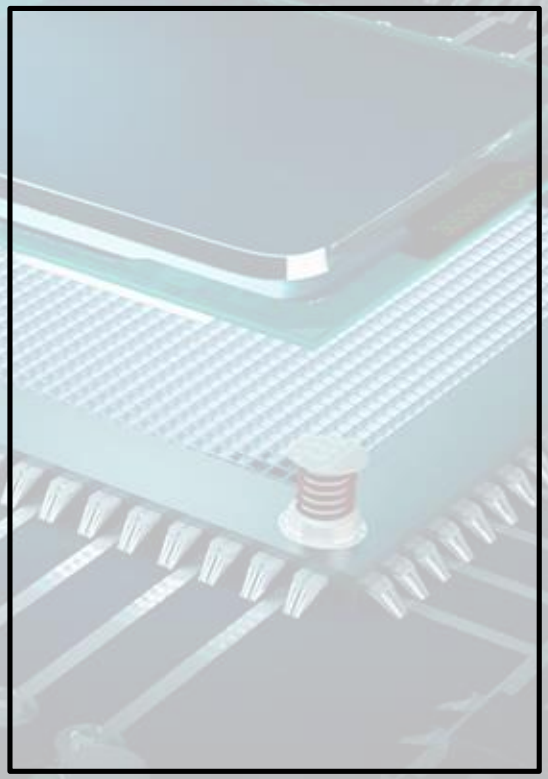
A Simple Java Application

CODE

execute

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK



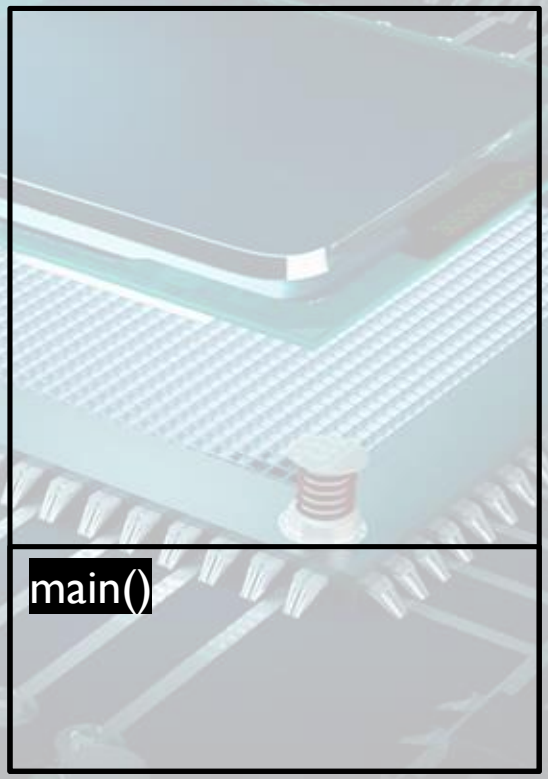
- The stack isn't really empty, other applications are running but as far as this application is concerned there is only an empry stack

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK



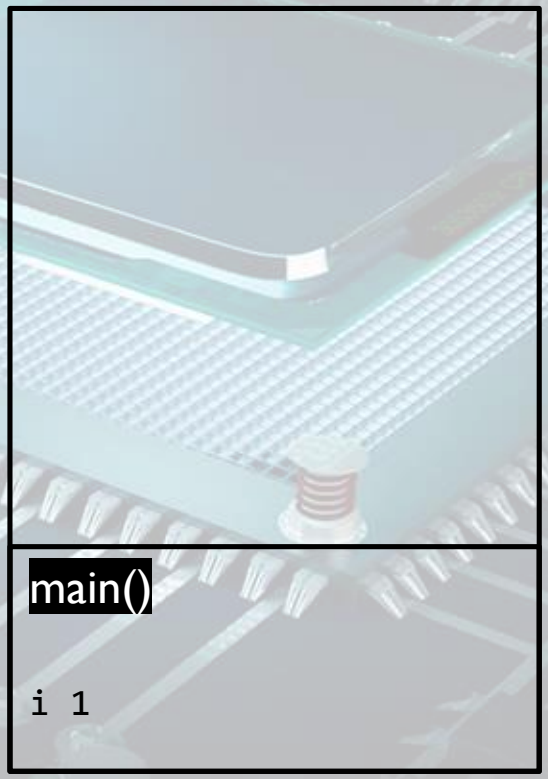
- main() frame added to stack

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK



- i variable created in the main() frame

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK

- (Two steps) silly() frame added to stack, locali variable created with a value passed in from main()

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK

- locali value in silly() changed to 3

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK

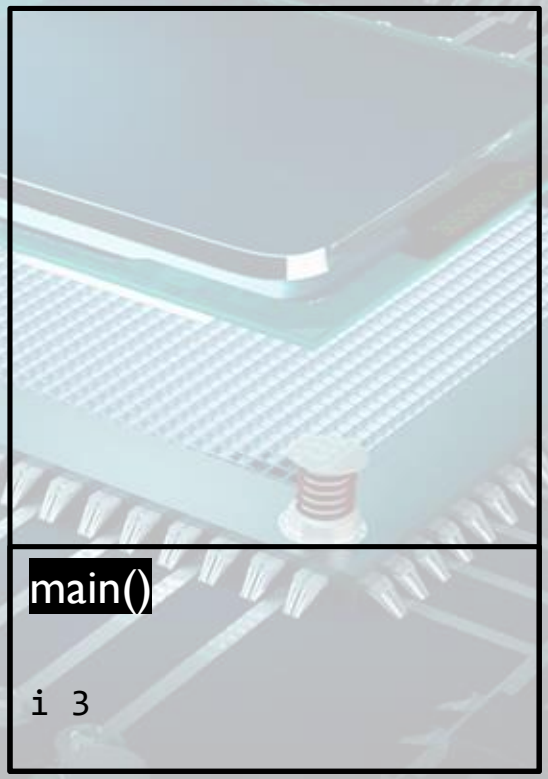
- locali variable in silly() returned to main() and stored in i

A Simple Java Application

CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK



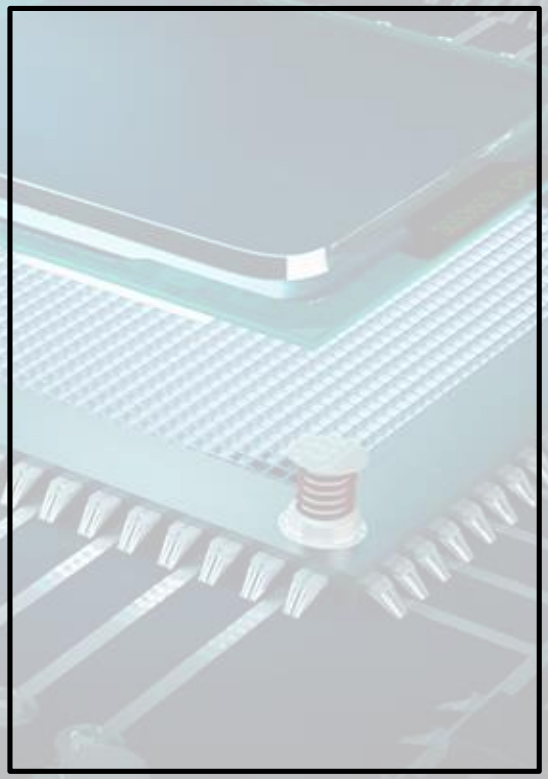
- silly() frame deleted as method exits

A Simple Java Application

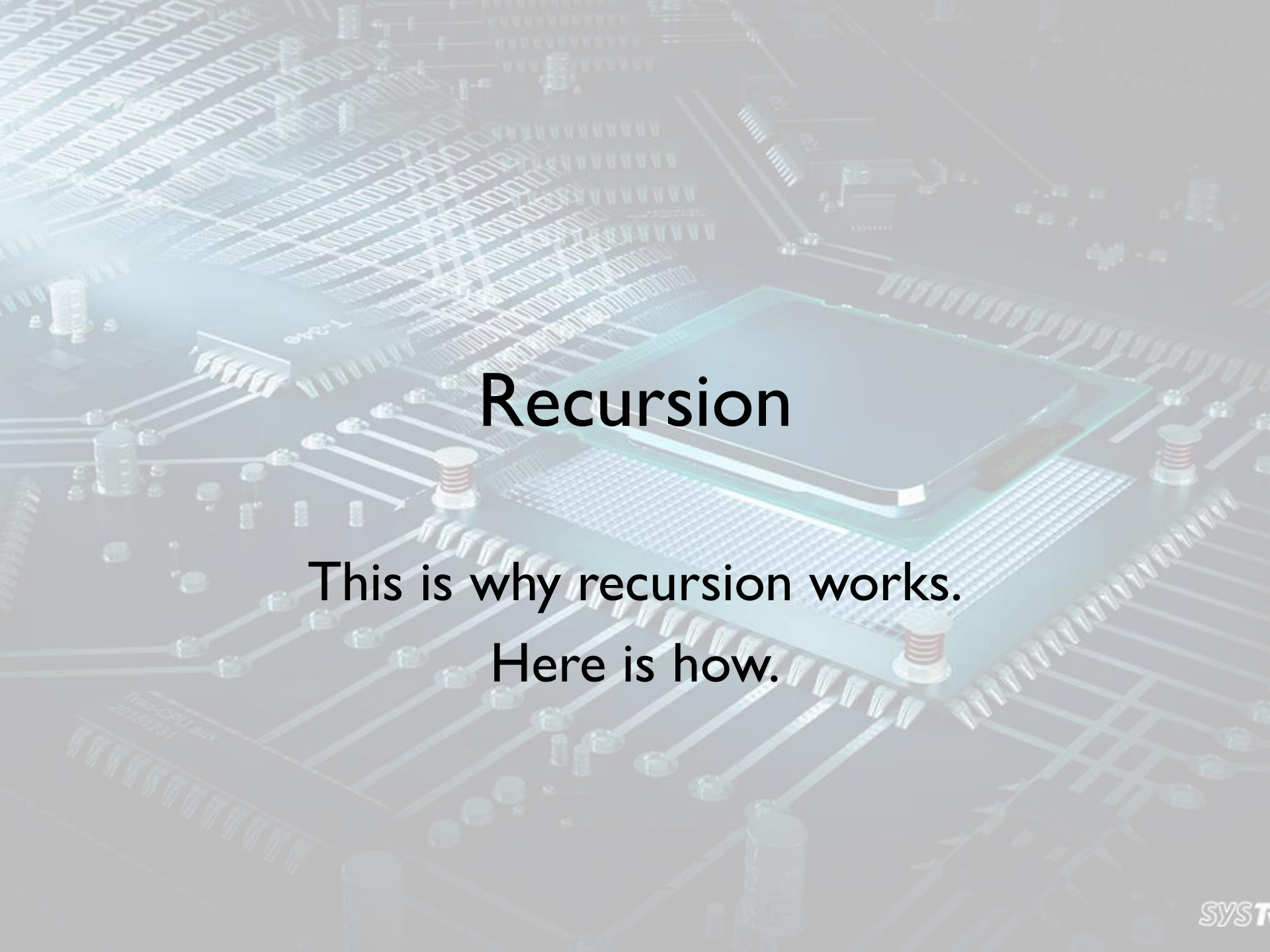
CODE

```
public static void main(String[] args)
{
  int i=1;

  i = silly(i);
}

int silly(int locali)
{
  locali = 3;
  return locali;
}
```

STACK



- main() frame deleted as method exits



Recursion

This is why recursion works.

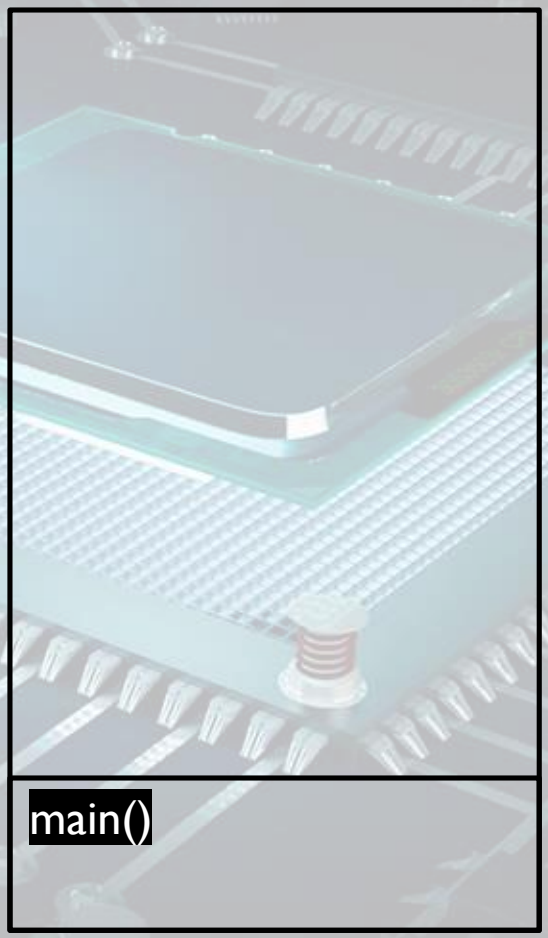
Here is how.

Recursion

CODE

```
int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

- To calculate exponential
- 5! = 5*4*3*2*1
- 4! = 4*3*2*1
- 5! = 5 * 4!
- 1! = 1

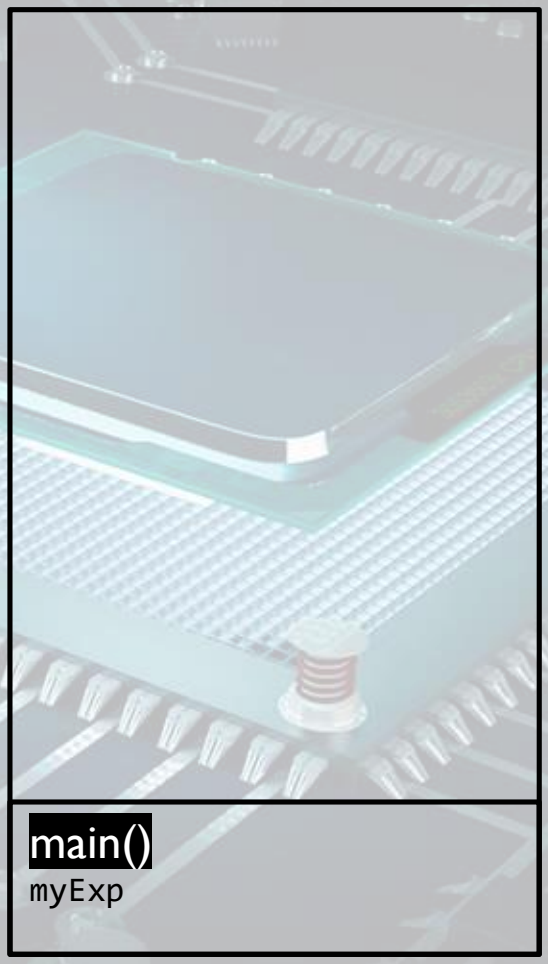


Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

- main() frame created as usual
- myExp created as a variable in main() frame

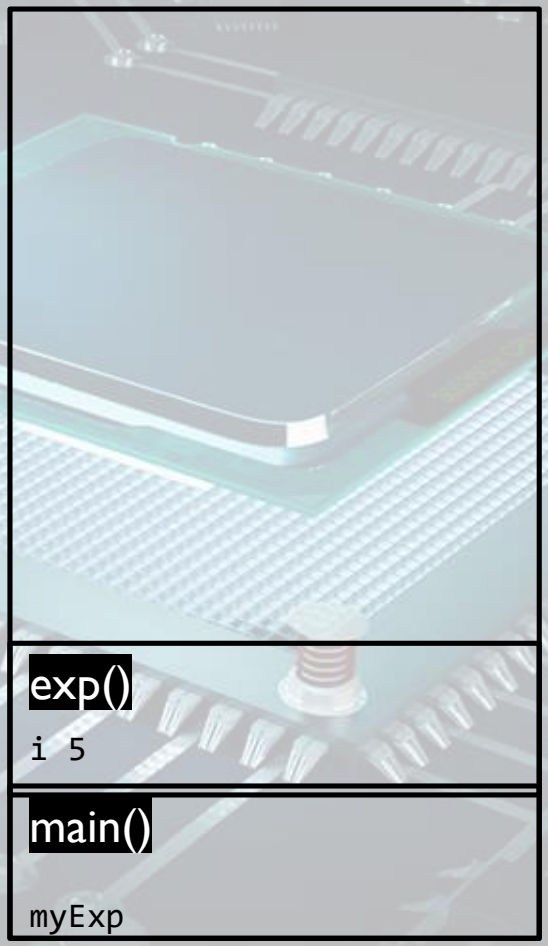


Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

- exp(5) called and a frame is created for exp()
- i is created in exp() and the value 5 is passed in to it

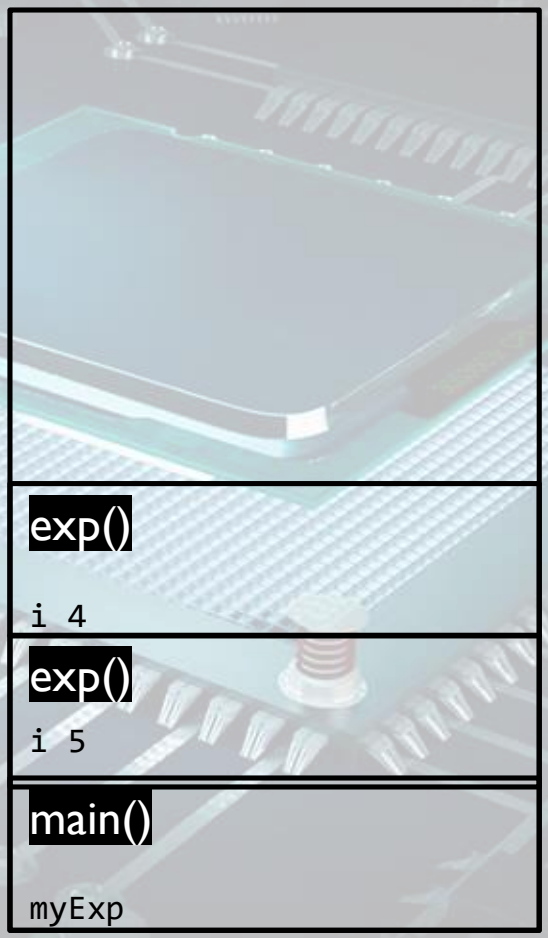


Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

- i is not equal to 1
  - So do the else, have to evaluate exp(i-1) first
- So we call exp(4), which means we need to
  - create a new frame for another exp()
  - create an i in the new exp() frame and set it to 4

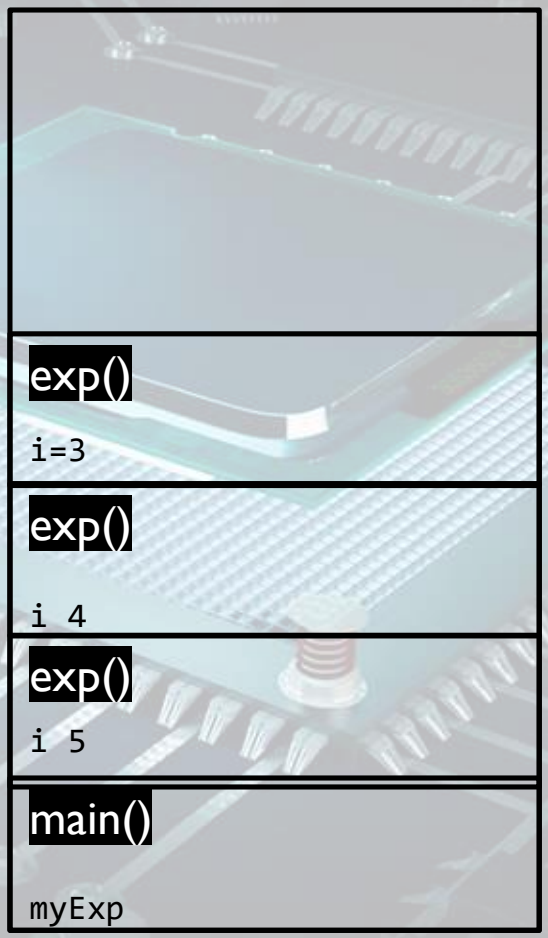


Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

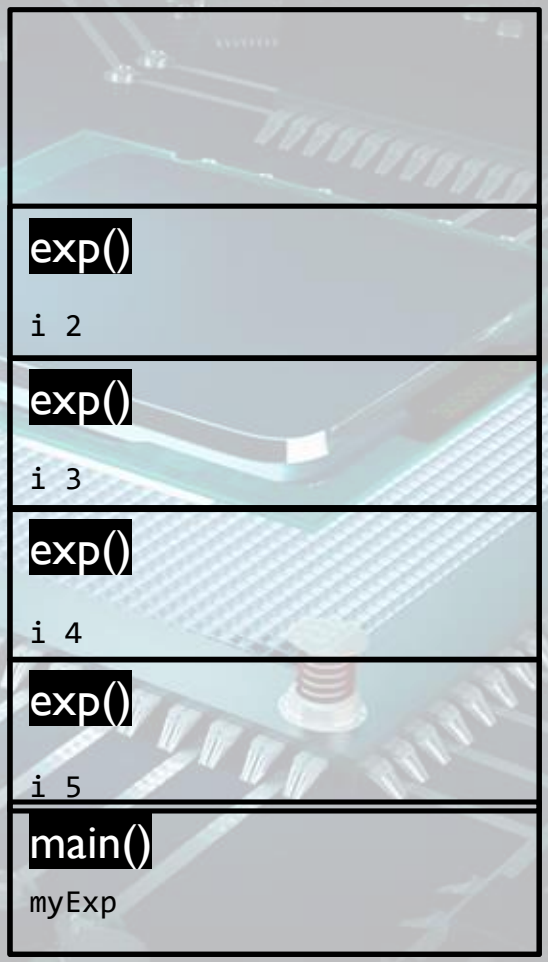
- i is not equal to 1
  - So do the else, have to evaluate exp(i-1) first
- So we call exp(3), which means we need to
  - create a new frame for another exp()
  - create an i in the new exp() frame and set it to 3



Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

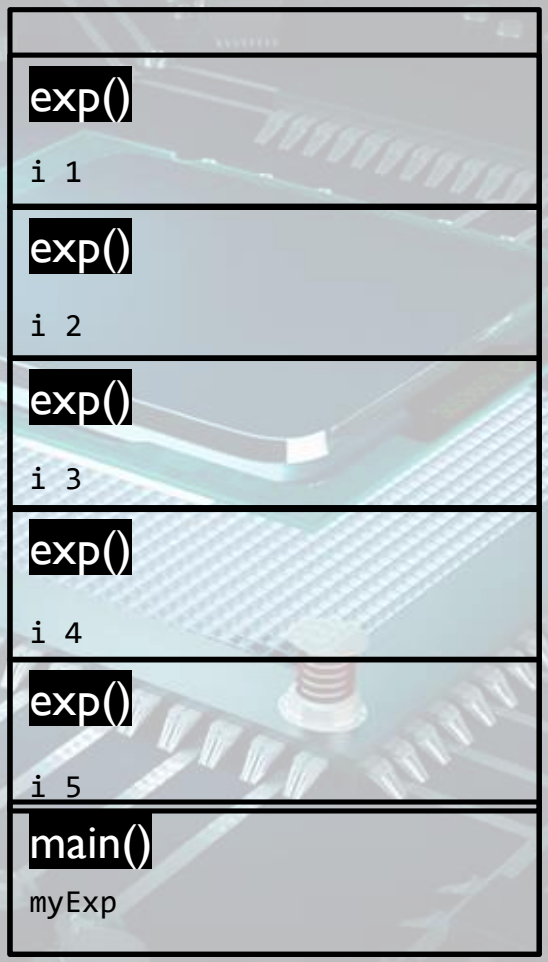


- i is not equal to 1
  - So do the else, have to evaluate exp(i-1) first
- So we call exp(2), which means we need to
  - create a new frame for another exp()
  - create an i in the new exp() frame and set it to 2

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

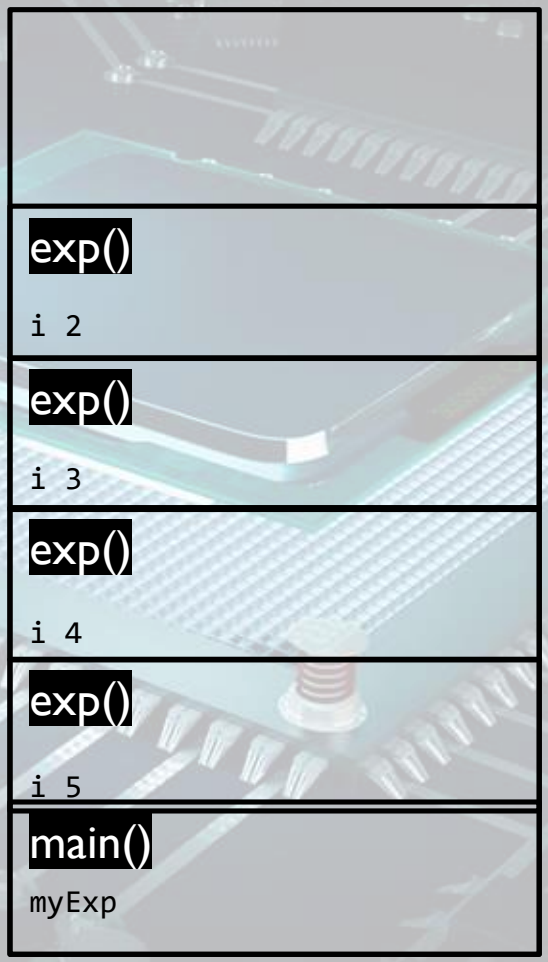


- i is not equal to 1
  - So do the else, have to evaluate exp(i-1) first
- So we call exp(1), which means we need to
  - create a new frame for another exp()
  - create an i in the new exp() frame and set it to 1

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

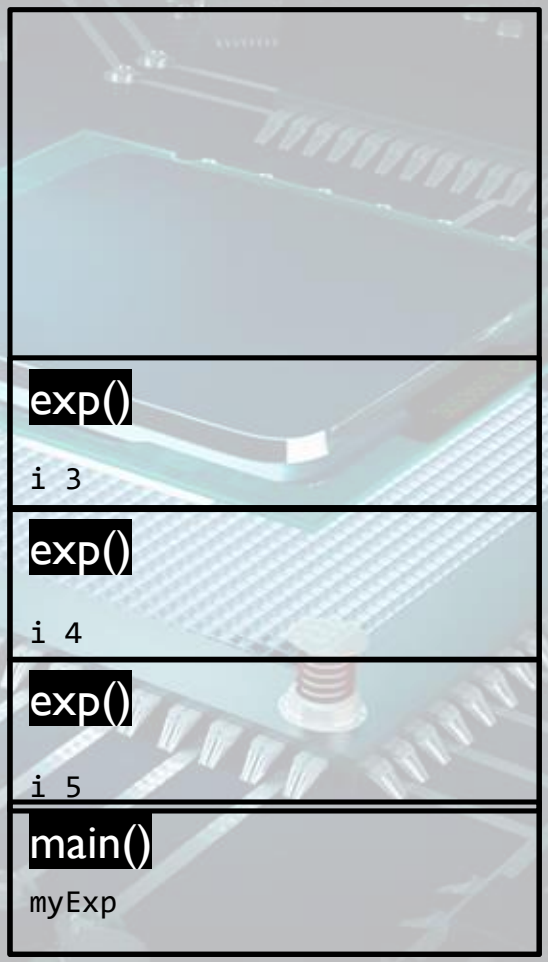


- i is equal to 1
  - So we return 1
  - And delete the current exp() frame

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

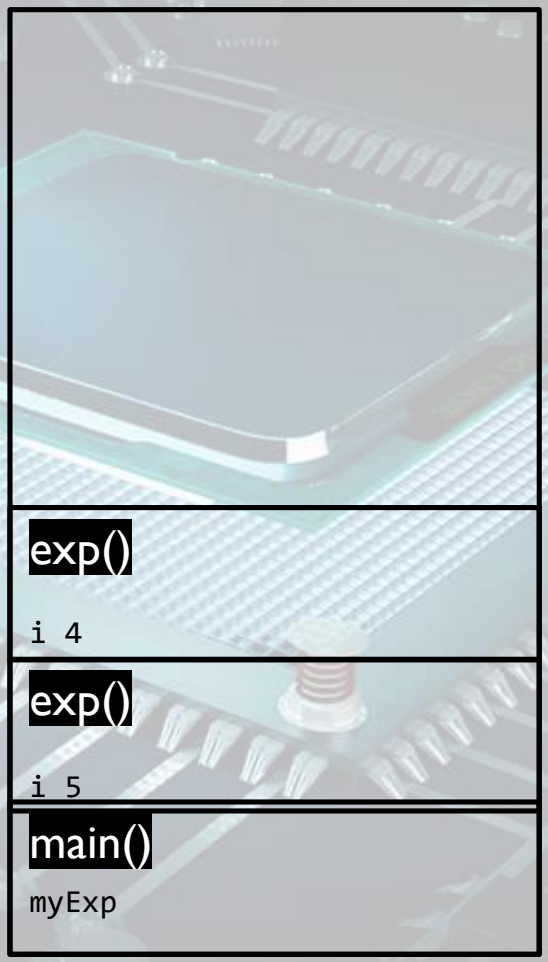


- We return to the previous exp() frame at the point we left
- Here i is 2 and exp(1) is the value we just returned i.e. 1
- So we return 2*1 from here and delete this frame

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

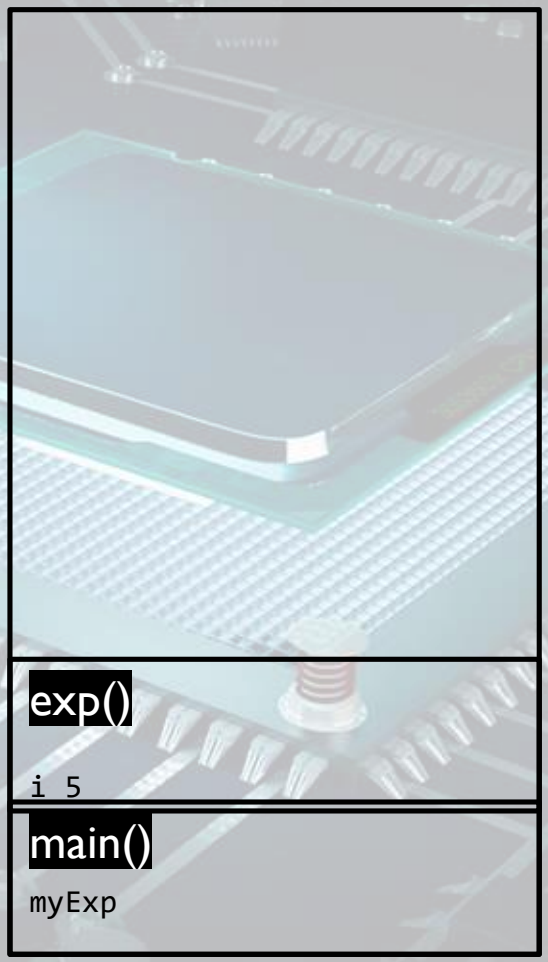


- We return to the previous exp() frame at the point we left
- Here i is 3 and exp(2) is the value we just returned i.e. 2
- So we return 3*2 (6) from here and delete this frame

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```

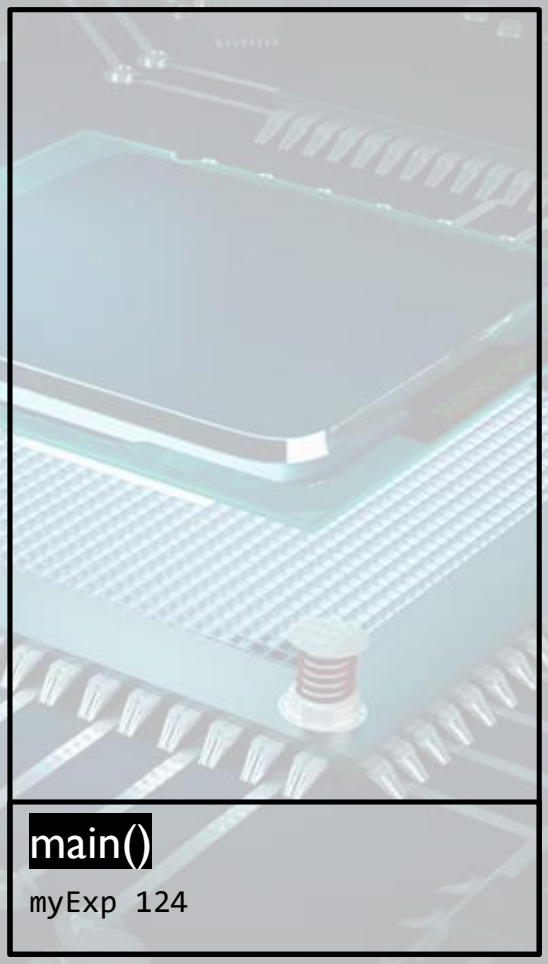


- We return to the previous exp() frame at the point we left
- Here i is 4 and exp(3) is the value we just returned i.e. 6
- So we return 4*6 (24) from here and delete this frame

Recursion

CODE

```
main()
{
  int myExp = exp(5);
}

int exp(int i)
{
  if (i==1)
    return 1;
  else
    return (i * exp(i-1));
}
```



- We return to the previous exp() frame at the point we left
- Here i is 5 and exp(4) is the value we just returned i.e. 24
- So we return 5*24 (120) from here and delete this frame

Summary

- Java Memory
- How the stack is used
  - Frames for method calls
  - Variables within frames
  - Parameters are passed
  - Values are returned
  - Recursion works with the stack

Next Week

- How objects are stored
- The Heap
- Difference between primitive variables and objects
- Direct and Indirect Referencing