CSCI 1103: Static Methods

Chris Kauffman

Last Updated:
Wed Oct 25 10:41:57 CDT 2017

Logistics

Reading from Eck

- Ch 4 on Methods (today)
- Ch 5 on Objects/Classes (later)

Goals

Finish methods, call stack

Lab07: Method Practice

Project 3

- Due Wednesday
- Questions?

Exercise: Project 3 Practice Conversion

Perform the following conversions

Binary to Decimal

Convert 1101001 base 2 to base 10

Decimal to Binary

Convert 86 base 10 to base 2

Answers: Project 3 Practice Conversion

Binary to Decimal: 1101101 from base 2 to base 10

Increasing powers of to right to left

```
   1101101 base 2
=  1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0
     64  +   32  + 0     +   8   +   4   + 0     +   1
= 109 base 10
```

Decimal to Binary: 86 from base 10 to base 2

Repeated division by 2, remainders in revers order

```
        Quotient          Remainder
 86 / 2 = 43       86 % 2 = 0 last digit
 43 / 2 = 21       43 % 2 = 1
 21 / 2 = 10       21 % 2 = 1
 10 / 2 =  5       10 % 2 = 0 middle digit
  5 / 2 =  2        5 % 2 = 1
  2 / 2 =  1        2 % 2 = 0
  1 / 2 =  0        1 % 2 = 1 first digit
Quotient = 0; terminate                  86 base 10 = 1010110 base 2
```

Finish Strings:

Recall

- What is a `String`?
- Is a `String` a reference or a primitive type?
- Describe similarities of `String` to arrays?
- Describe a key limitation of `String` compared to standard arrays and the special word associated with that limitation

String Concatenation

Previous slides

Functions, Subroutines, Methods

- Many names for the idea of *parameterized code*
- Given some inputs, do some stuff, produce some outputs
- We are acquainted with this via
  - Ask user for input
  - Do stuff
  - Print things on the screen
- All programming languages have a way for this to happen without user typing or printing to screen
- In Java, such things are called methods
- We've been writing them since the beginning as `main()` is a method with some special properties
- Terminology:
  - Run / Invoke / Call a method (function, subroutine)
    All names to make use of some parameterized code
  - Caller: whoever is running a method, usually a method
  - Callee: the method that is being run

Exercise: `FirstMethod.java`

```
 1  public class FirstMethod{                  // Class demonstrating a non-main() method
 2
 3    public static void printCatchphrase(){  // a method which returns nothing (void)
 4      String msg = "Wubalubadubdub!!";      // variable that is local to method
 5      System.out.println(msg);              // print something
 6      return;                               // optional return, no needed due to void type
 7    }
 8
 9    public static void main(String args[]){ // main() method: execution starts here
10      System.out.println("Insert catchphrase");
11      printCatchphrase();                   // run the printCatchphrase() method
12      System.out.println("and again...");   // print something additional
13      printCatchphrase();                   // run printCatchphrase() again
14    }
15  }
```

Sample Run

```
> javac FirstMethod.java
> java FirstMethod
Insert catchphrase
Wubalubadubdub!!
and again...
Wubalubadubdub!!
>
```

Questions

- Where does execution begin based on the sample run
- What new keyword for returning from a method is introduced
- Identify the caller and callee of methods and the location where a method is being invoked

Scope of Variables in Methods

- Methods each have their own scope (?)
- Variables in one method are not accessible to another method
- Attempts to reference local vars in another method will lead to errors

```
 1  // Methods have their own scope for variables and can't share variable
 2  // declarations
 3  public class MethodScoping{
 4
 5    public static void printCatchphrase(){  // a method which returns nothing (void)
 6      String msg = "Wubalubadubdub!!";      // variable that is local to method
 7      System.out.println(msg);              // print something
 8      return;                               // optional return, no needed due to void type
 9    }
10
11    public static void main(String args[]){ // main() method: execution starts here
12      System.out.println(msg);              // msg not found: local to printCatchphrase()
13                                            // so not visible in main()
14    }
15  }
```

```
> javac MethodScoping.java
MethodScoping.java:12: error: cannot find symbol
    System.out.println(msg);
                       ^
  symbol:   variable msg
  location: class MethodScoping
1 error
```

Communication between Methods

- Parameters or arguments allow information to be given to methods
- Return values allow a method to give an answer back
- Signature of a method shows what parameters it needs and what return value will be given back

Signatures have the following form

```
public static retType methodName(ArgType1 arg1, ArgType2 arg2,
                                  ArgType3 arg3)
```

- `retType` is the type returned such as `int`, `double`, `String`
- `methodName` is the name of the method
- `ArgType1` is the type of the first argument such as `int`, `double`, `String`
- `arg1` is the name of the first argument
- Likewise for `ArgType2 arg2` and `ArgType3 arg3`

Exercise: Analyze Method Signatures

Identify the name, return type, and arguments to the following methods based on their signatures

```
public static void printCatchphrase(){ .. }
```

```
public static void printString(String str){ .. }
```

```
public static int magicNum(){ .. }
```

```
public static int countLargerThan(double arr[], double thresh){..}
```

```
public static char [] asCharArray(String str){ .. }
```

```
public static double [][] specialMat(int rows, int cols, String options){..}
```

Answers: Analyze Method Signatures

```
public static void printCatchphrase(){ .. }
// name printCatchphrase
// no parameters, no return value

public static void printString(String str){ .. }
// name printString
// one parameter, str a string, no return value

public static int magicNum(){ .. }
// name magicNum
// no parameters, returns an int

public static int countLargerThan(double arr[], double thresh){..}
// name countLargerThan
// two parameters, arr : double array, and thresh : a double
// returns an integer

public static char [] asCharArray(String str){ .. }
// name asCharArray
// one parameter, str : a String
// returns an array of char

public static double [][] specialMat(int rows, int cols, String options){..}
// name specialMat
// 3 parameters: rows int, cols int, and options String
// returns a 2D double array
```

Calling Existing Methods

- We have been doing this since early
  - `System.out.println("hello");`
  - `double sqrt2 = Math.sqrt(2);`
  - `double twoToPow = Math.pow(2.0, 7.4);`
- Appropriate number of arguments must be given
- Answer produced by the method must be stored in a variable
- Uses class names like `Math` and `System` to specify where a method lives
- In `FirstMethod`, `printCathphrase()` did not need this

```
public class FirstMethod{

  public static void printCatchphrase(){..}

  public static void main(String args[]){
    System.out.println("Insert catchphrase");
    printCatchphrase();
    ...
```

- Methods can be directly called within the class they are defined; outside that class must use class name

Exercise: Method Behavior and `return`

- Methods begin running from their start with parameters given
- When a `return` statement is encountered, go back to caller
- If the method must return an answer (return type is not `void`), must `return answer;`
- Trace the execution of the code below and show output

```
1  public class SampleMethods{                 // Class demonstrating a non-main() method
2
3    public static void printCatchphrase(){  // a method which returns nothing (void)
4      String msg = "Wubalubadubdub!!";      // variable that is local to method
5      System.out.println(msg);              // print something
6      return;                               // optional return, no needed due to void type
7    }
8
9    public static int multiplyAdd(int a, int b, int c){
10     int mult = a * b;
11     int sum = mult + c;
12     return sum;                           // must return an integer
13   }
14
15   public static void main(String args[]){ // main() method: execution starts here
16     int ans = multiplyAdd(3,4,5);         // call the multiplyAdd() method
17     System.out.println(ans);              // print the answer produced
18     printCatchphrase();                   // run the printCatchphrase() method
19     int x=2, y=1, z=7;                    // local variables
20     int ans2 = multiplyAdd(x,y,z);        // call the multiplyAdd() method again
21     System.out.println(ans2);             // print the answer produced
22   }
23 }
```

Answer: Method Behavior and `return`

```
 1  public class SampleMethods{
 2
 3    public static void printCatchphrase(){
 4      String msg = "Wubalubadubdub!!";
 5      System.out.println(msg);
 6      return;
 7    }
 8
 9    public static int multiplyAdd(int a, int b, int c){
10      int mult = a * b;
11      int sum = mult + c;
12      return sum;
13    }
14
15    public static void main(String args[]){
16      int ans = multiplyAdd(3,4,5);
17      System.out.println(ans);
18      printCatchphrase();
19      int x=2, y=1, z=7;
20      int ans2 = multiplyAdd(x,y,z);
21      System.out.println(ans2);
22    }
23  }
```

```
> javac SampleMethods.java
> java SampleMethods
17
Wubalubadubdub!!
9
>
```

Line	Method
	main
16	callMultiplyAdd
10	multiplyAdd
11	
12	`return`
17	main
18	call printCatchphrase
4	printCatchPhrase
5	
6	`return`
19	main
20	call multiplyAdd
10	multiplyAdd
11	
12	`return`
21	main

Why Methods?

- New programmers often wonder: *Why use methods at all?*
- True that one can just copy and paste the same code
- An appeal to your aesthetic may be in order. Consider:

```
 1  // Simple program that makes use of method invocations to calculate a
 2  // square root. Contrast with ComplexSqrt.java where full-ish method
 3  // bodies are substituted in.
 4  public class SimpleSqrt{
 5    public static void main(String args[]){
 6      String prompt = "Enter a number:";
 7      System.out.println(prompt);
 8      double x = TextIO.getDouble();
 9      double rootX = Math.sqrt(x);
10      String result = "Square root is: "+rootX;
11      System.out.println(result);
12    }
13  }
```

A fairly simple program to understand
Examine its alternative where the (rough) text of some methods is copied and pasted into their corresponding locations.

`ComplexSqrt.java`

Method Specification

- A method *specification* summarizes a method, info like...
- Signature: name, parameters, return type (and visibility)
- General description of behavior
- Errors / Exceptions that can occur

Examples of Method Specs

```
public class MethodSpecs{

  public static boolean compareIntArrays(int x[], int y[]);
  // Compare two integer arrays. Return true if all elements are equal
  // and false if any elements are unequal.  Return false if the
  // arrays are differnt length.

  public static int indexOf(String s, char q);
  // Determine if character q is in String s. Return the lowest index
  // of s that has q if it is present.  Return -1 if q is not
  // present. Throw an exception if s is null.

}
```

Developing Specs, Writing Methods

- A key skill in our course and in life is breaking large, *ambiguous* problems into smaller, manageable sub-problems which are *clearly defined*
- Without a clear description of the problem, solutions are nearly impossible

Writing Specifications

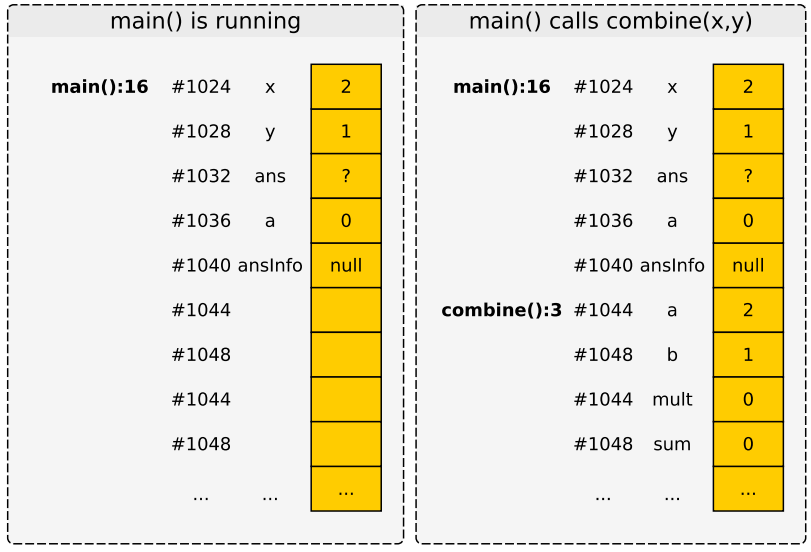
- Defines how a problem is subdivided
- Comes under the general heading of design
- Most projects will provide a specifications, trains you on what to strive for
- We will likely have at least one free-form project with more design freedom

Implementing Methods

- Given a specification, write code that "does" it
- Will comprise a lot of the work associated with subsequent projects
- Not as creative as designing your own but good training
- Allows us to provide test cases for your code

The Call Stack: Where Method Data Lives

- Methods have local data
  - Parameters passed in
  - Variables within the method
- Each active method has a stack frame on the call stack
- Contains enough memory for parameters and local variables of a running method
- Frames also contain a record of the current instruction
- Invoking a method pushes a new frame on the stack

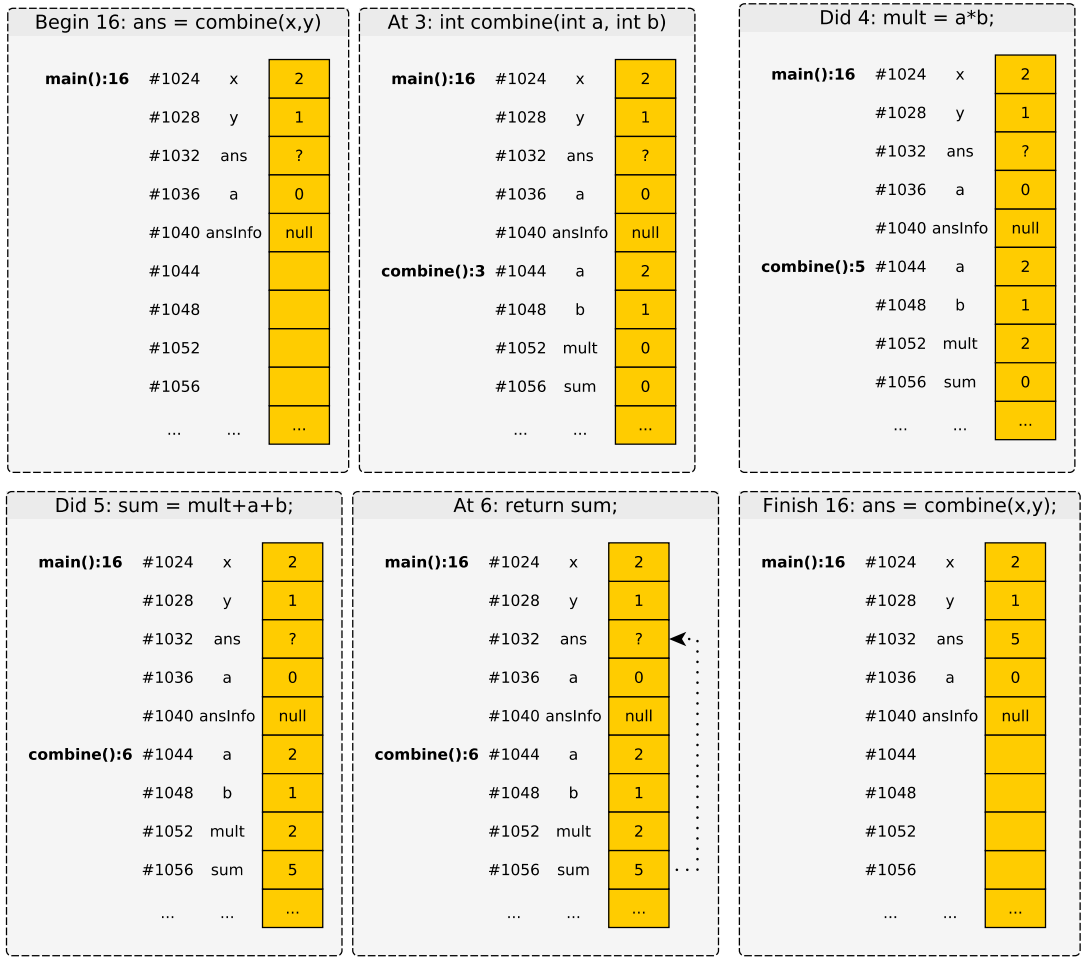


- Stack frames are tightly packed in memory: adjacent to each other
- When a method `returns`, a frame pops off the stack destroying all local variables

Demonstration of Stack Frames 1

- Control goes from `main()` to `combine()` at line 13
- `Main()` remembers where it left off, resumes line 13 after receiving return value

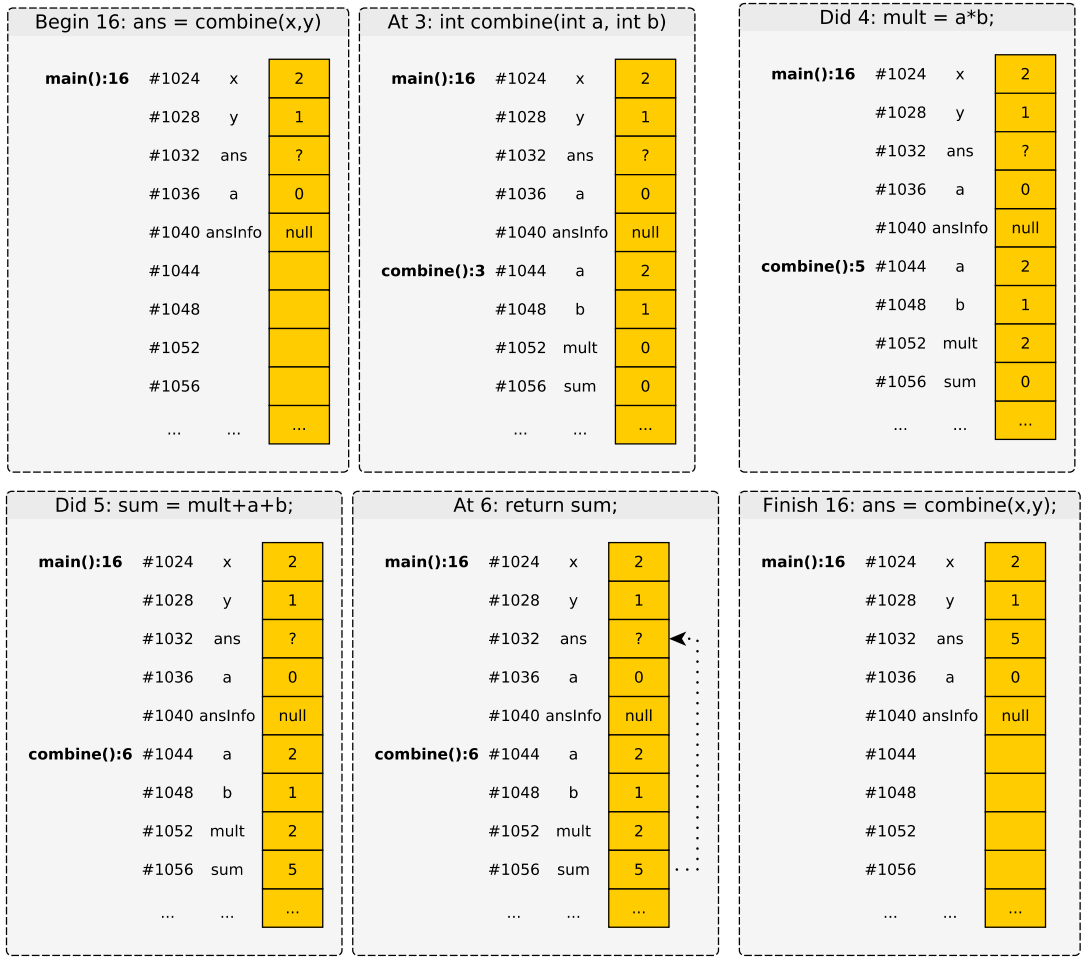
```
 1 public class CallStackDemo{
 2
 3   public static int combine(int a, int b){
 4     int mult = a * b;
 5     int sum = mult + a+b;
 6     return sum;
 7   }
 8
 9   public static String info(int x){
10     String xInfo = "Number is "+x;
11     return xInfo;
12   }
13
14   public static void main(String args[]){
15     int x=2, y=1;
16 >>  int ans = combine(x,y);
17
18     int a=6;
19     ans = combine(a,a);
20
21     String ansInfo = info(ans);
22     ansInfo = info( combine(5,ans) );
23
24   }
25 }
```



Exercise: Demonstration of Stack Frames 2

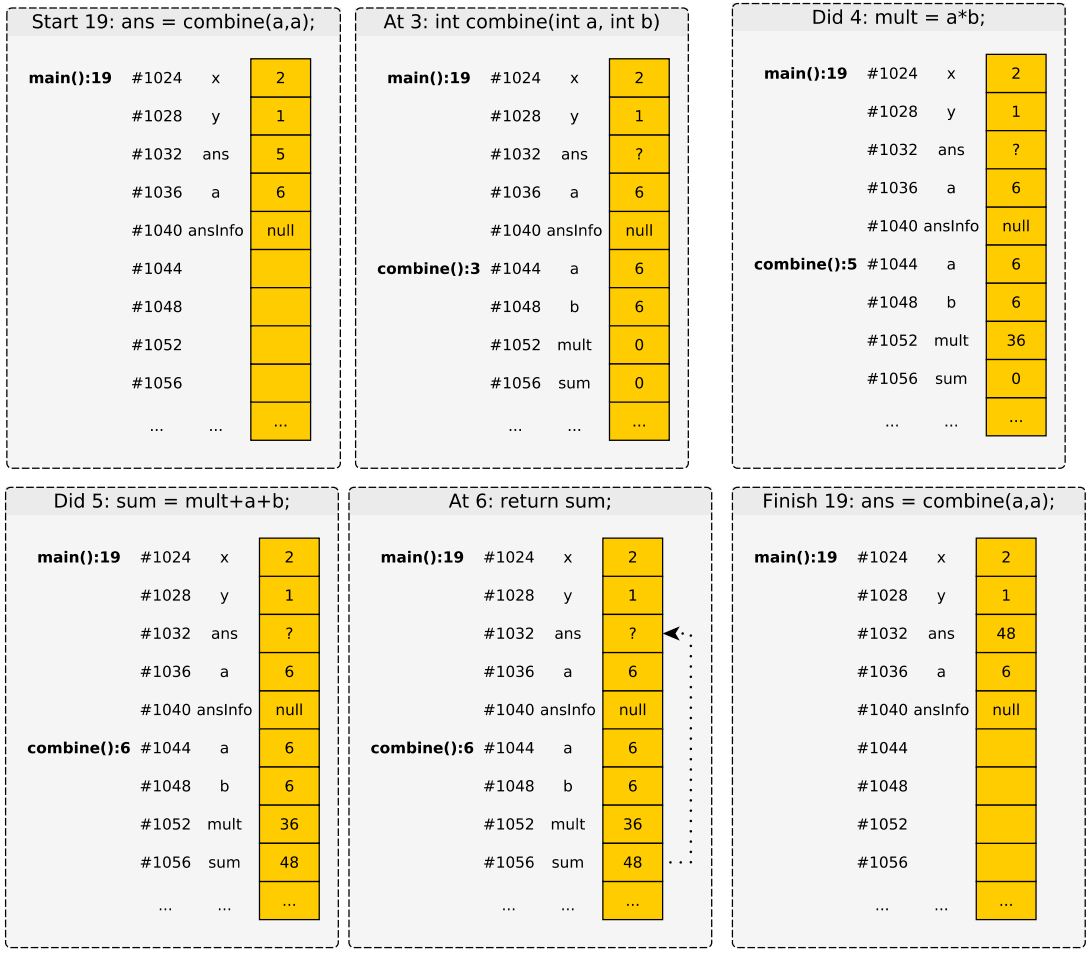
- Based on these pictures draw the next steps for lines 18 and 19 of `main()`
- Make sure to show how a frame for `combine()` gets pushed on then popped off

```
 1 public class CallStackDemo{
 2
 3   public static int combine(int a, int b){
 4     int mult = a * b;
 5     int sum = mult + a+b;
 6     return sum;
 7   }
 8
 9   public static String info(int x){
10     String xInfo = "Number is "+x;
11     return xInfo;
12   }
13
14   public static void main(String args[]){
15     int x=2, y=1;
16 >>  int ans = combine(x,y);
17
18     int a=6;
19     ans = combine(a,a);
20
21     String ansInfo = info(ans);
22     ansInfo = info( combine(5,ans) );
23
24   }
25 }
```



Answer: Demonstration of Stack Frames 2

```
 1 public class CallStackDemo{
 2
 3   public static int combine(int a, int b){
 4     int mult = a * b;
 5     int sum = mult + a+b;
 6     return sum;
 7   }
 8
 9   public static String info(int x){
10     String xInfo = "Number is "+x;
11     return xInfo;
12   }
13
14   public static void main(String args[]){
15     int x=2, y=1;
16     int ans = combine(x,y);
17
18     int a=6;
19 >>  ans = combine(a,a);
20
21     String ansInfo = info(ans);
22     ansInfo = info( combine(5,ans) );
23
24   }
25 }
```



Note that there are multiple variables named a active but this is not a problem

Stack and Heap

Stack / Call Stack / Method

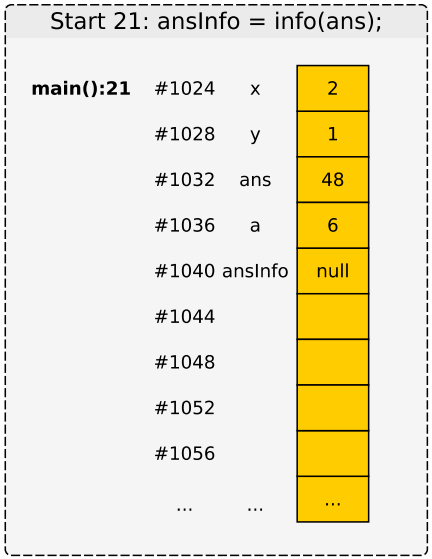
- Where data associated with running methods lives
- Whenever a method is called, more space is allocated on the stack
- Whenever a method finishes, memory is removed from the stack
- The method that is actually running is at the *top of the stack*
- Lower stack frames are waiting to get control back

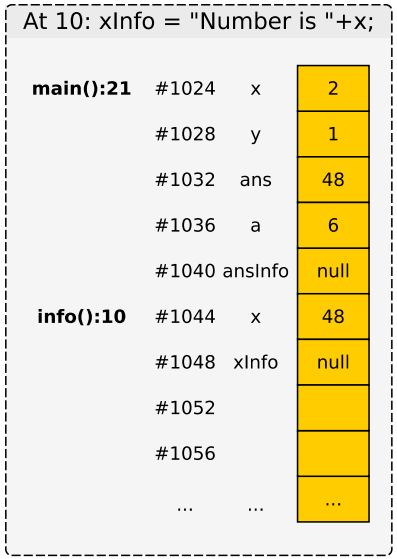
Heap / Dynamic Memory

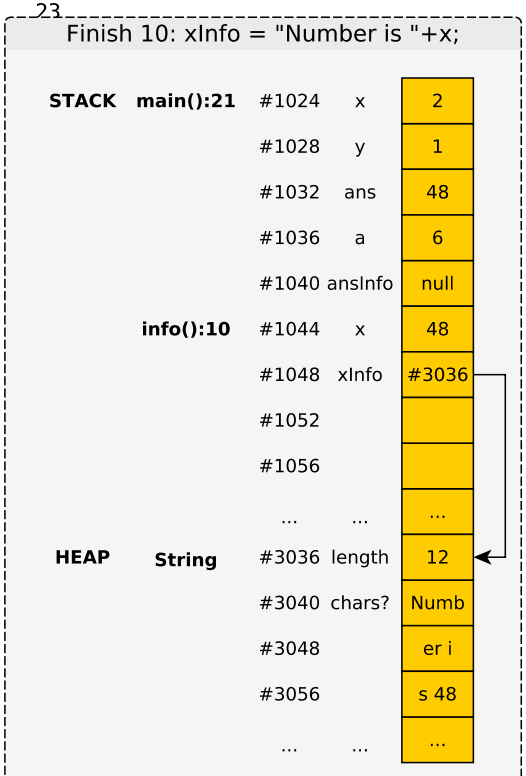
- All reference types allocated with `new` live in the heap
- Multiple stack variables can refer to locations in the heap
- Heap data can be primitives or references to other heap locations

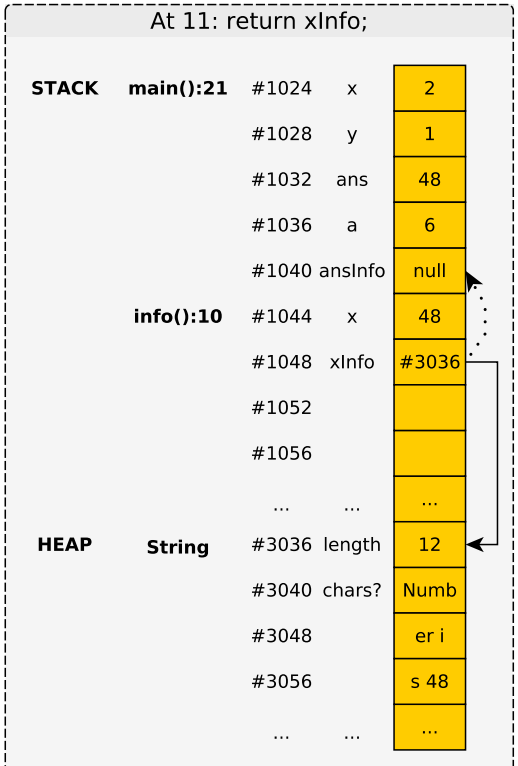
Stack and Heap in Pictures

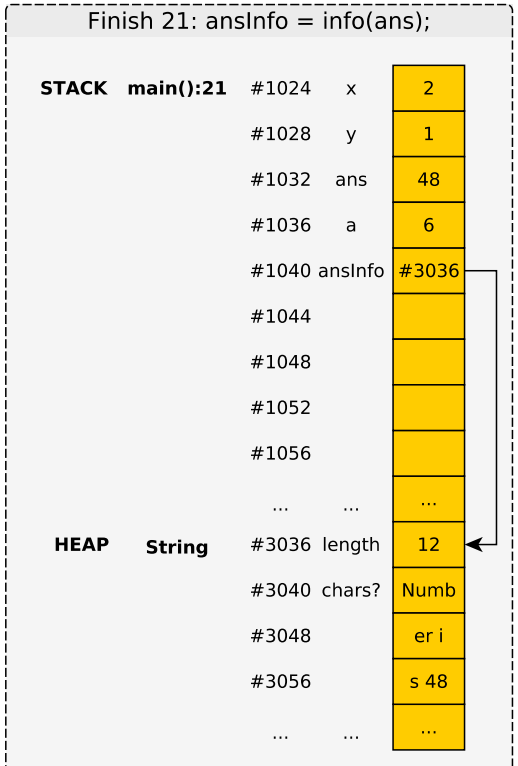
```
 1 public class CallStackDemo{
 2
 3   public static int combine(int a, int b){
 4     int mult = a * b;
 5     int sum = mult + a+b;
 6     return sum;
 7   }
 8
 9   public static String info(int x){
10     String xInfo = "Number is "+x;
11     return xInfo;
12   }
13
14   public static void main(String args[]){
15     int x=2, y=1;
16     int ans = combine(x,y);
17
18     int a=6;
19     ans = combine(a,a);
20
21 >>  String ansInfo = info(ans);
22     ansInfo = info( combine(5,ans) );
23
```











Stack Stacks Up

- Stack builds up with each call
- When a method is called, current method suspends
- Stack frame pushed on for new method
- Execution begins
- Can happen to very deep stack depths
- Stack variables may refer to same heap locations
- Demonstrate with `DeepStack.java`

Stack Stacks Up: 3 Frames deep

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4 3>  if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13     return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17 2>  boolean nameOK = checkLength(name);
18     boolean xOK = inRange(x);
19     return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25 1>  boolean validA = doChecks(candA,iqA);
26
27     String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- Stack contains 3 frames
- checkLength() active method
- doChecks() watining for it
- main() waiting for doChecks()

STACK

Method:Line	Box#	VarName	Value
main:25	#1024	candA	#3036
	#1028	iqA	47.3
Doing line 25	#1036	validA	?
	#1038	candB	null
	#1042	iqB	0.0
	#1050	validB	false
doChecks:17	#1052	name	#3036
	#1056	x	47.3
Doing line 17	#1064	nameOK	?
	#1066	xOK	false
checkLength:4	#1068	word	#3036

HEAP

Box#	Name	Value
#3036	length	5
	chars	"Morty"
#4048	length	11
	chars	"Jerry Smith"

Stack Stacks Up: Finish 1 method

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4     if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13     return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17     boolean nameOK = checkLength(name);
18 2>  boolean xOK = inRange(x);
19     return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25 1>  boolean validA = doChecks(candA,iqA);
26
27     String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- checkLength() finishes
- true returned to doChecks()
- frame for checkLength() pops off
- Stack contains 2 frames

STACK

Method:Line	Box#	VarName	Value
main:25	#1024	candA	#3036
	#1028	iqA	47.3
Doing line 25	#1036	validA	?
	#1038	candB	null
	#1042	iqB	0.0
	#1050	validB	false
doChecks:18	#1052	name	#3036
	#1056	x	47.3
Finished 17	#1064	nameOK	true
	#1066	xOK	false

HEAP

Box#	Name	Value
#3036	length	5
	chars	"Morty"
#4048	length	11
	chars	"Jerry Smith"

Stack Stacks Up: Start another method

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4     if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13 3>  return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17     boolean nameOK = checkLength(name);
18 2>  boolean xOK = inRange(x);
19     return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25 1>  boolean validA = doChecks(candA,iqA);
26
27     String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- push on inRange()
- copy x to parameter z
- Stack contains 3 frames

STACK

Method:Line	Box#	VarName	Value
main:25	#1024	candA	#3036
	#1028	iqA	47.3
Doing line 25	#1036	validA	?
	#1038	candB	null
	#1042	iqB	0.0
	#1050	validB	false
doChecks:18	#1052	name	#3036
	#1056	x	47.3
	#1064	nameOK	true
Doing 18	#1066	xOK	?
inRange:13	#1068	z	47.3

HEAP

Box#	Name	Value
#3036	length	5
	chars	"Morty"
#4048	length	11
	chars	"Jerry Smith"

Stack Stacks Up: Finish another method

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4     if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13     return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17     boolean nameOK = checkLength(name);
18     boolean xOK = inRange(x);
19 2>  return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25 1>  boolean validA = doChecks(candA,iqA);
26
27     String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- inRange() finishes
- true returned to doChecks()
- frame for inRange() pops off
- Stack contains 2 frames

STACK

Method:Line	Box#	VarName	Value
main:25	#1024	candA	#3036
	#1028	iqA	47.3
Doing line 25	#1036	validA	?
	#1038	candB	null
	#1042	iqB	0.0
	#1050	validB	false
doChecks:18	#1052	name	#3036
	#1056	x	47.3
	#1064	nameOK	true
Finished 18	#1066	xOK	true

HEAP

Box#	Name	Value
#3036	length	5
	chars	"Morty"
#4048	length	11
	chars	"Jerry Smith"

Stack Stacks Up: Back to main()

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4     if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13     return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17     boolean nameOK = checkLength(name);
18     boolean xOK = inRange(x);
19     return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25     boolean validA = doChecks(candA,iqA);
26
27 1>  String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- checkLength() finishes
- true returned to doChecks()
- frame for checkLength() pops off
- Stack contains 2 frames

STACK

Method:Line	Box#	VarName	Value
main:27	#1024	candA	#3036
	#1028	iqA	47.3
Finished 25	#1036	validA	true
	#1038	candB	null
	#1042	iqB	0.0
	#1050	validB	false

HEAP

Box#	Name	Value
#3036	length	5
	chars	"Morty"
#4048	length	11
	chars	"Jerry Smith"

Stack Stacks Up: Offline Exercise

```
 1 public class DeepStack{
 2
 3   public static boolean checkLength(String word){
 4     if(word.length() < 10){
 5       return true;
 6     }
 7     else{
 8       return false;
 9     }
10   }
11
12   public static boolean inRange(double z){
13     return 25.5 <= z && z < 65.0;
14   }
15
16   public static boolean doChecks(String name, double x){
17     boolean nameOK = checkLength(name);
18     boolean xOK = inRange(x);
19     return nameOK || xOK;
20   }
21
22   public static void main(String args[]){
23     String candA = "Morty";
24     double iqA = 47.3;
25     boolean validA = doChecks(candA,iqA);
26
27 1>  String candB = "Jerry Smith";
28     double iqB = 27.8;
29     boolean validB = doChecks(candB,iqA);
30   }
31 }
```

- A good exercise to complete running the above line by line
- Push frames on as methods start
- Pop frames them off as methods finis

Exercise: Swapping and Methods

```
1  public class SwapMethods{
2
3    // Swap two ints?
4    public static void swapIntsP(int x, int y){
5      int tmp = x;
6      x = y;
7      y = tmp;
8    }
9
10   // Swap 0th element of two int arrays?
11   public static void swapIntsR(int x[], int y[]){
12     int tmp = x[0];
13     x[0] = y[0];
14     y[0] = tmp;
15   }
16
17   public static void main(String args[]){
18     int a=3, b=5;
19     swapIntsP(a,b);
20     System.out.printf("a: %d  b: %d\n",
21                       a,b);
22
23     int aarr[] = {4}, barr[] = {6};
24     swapIntsR(aarr,barr);
25     System.out.printf("aarr[0]: %d  barr[0]: %d\n",
26                       aarr[0],barr[0]);
27
28
29   }
30 }
```

- Predict the output of running `main()`
- Calls two "swapping" methods
- Draw a memory diagram to support your predictions
- Do numbers *actually swap* in both cases?

Answer: Swapping and Methods

```
1  public class SwapMethods{
2
3    // Swap two ints?
4    public static void swapIntsP(int x, int y){
5      int tmp = x;
6      x = y;
7      y = tmp;
8    }
9
10   // Swap 0th element of two int arrays?
11   public static void swapIntsR(int x[], int y[]){
12     int tmp = x[0];
13     x[0] = y[0];
14     y[0] = tmp;
15   }
16
17   public static void main(String args[]){
18     int a=3, b=5;
19     swapIntsP(a,b);
20     System.out.printf("a: %d  b: %d\n",
21                       a,b);
22
23     int aarr[] = {4}, barr[] = {6};
24     swapIntsR(aarr,barr);
25     System.out.printf("aarr[0]: %d  barr[0]: %d\n",
26                       aarr[0],barr[0]);
27
28
29   }
30 }
```

Altering Primitive Args: Fail

- `swapIntsP()` does not swap arguments passed in
- Copies of args made on the call stack
- Changes to copies are lost when frame pops

Altering Reference Args: Yes

- `swapIntsR()` does swap
- Copies of references made on the call stack
- Changes to memory area referred to are seen by `main()`

Exercise: Increase

```
1  public class Increase{
2
3    // Increase?
4    public static int increaseP(int x){
5      x++;
6      return x;
7    }
8
9    // Increase?
10   public static int increaseR(int x[]){
11     x[0]++;
12     return x[0];
13   }
14
15   public static void main(String args[]){
16     int a=2, b=4;
17     increaseP(a);
18     b = increaseP(b);
19
20     System.out.printf("a: %d  b: %d\n",
21                       a,b);
22
23     int aar[] = {6}, bar[]={8};
24     increaseR(aar);
25     bar[0] = increaseR(bar);
26
27     System.out.printf("aar[0]: %d  bar[0]: %d\n",
28                       aar[0],bar[0]);
29   }
30 }
```

- Similar to swapping activity
- Predict the output of running `main()`
- Calls two "increase" methods
- Draw a memory diagram to support your predictions
- Do numbers *actually swap* in both cases?

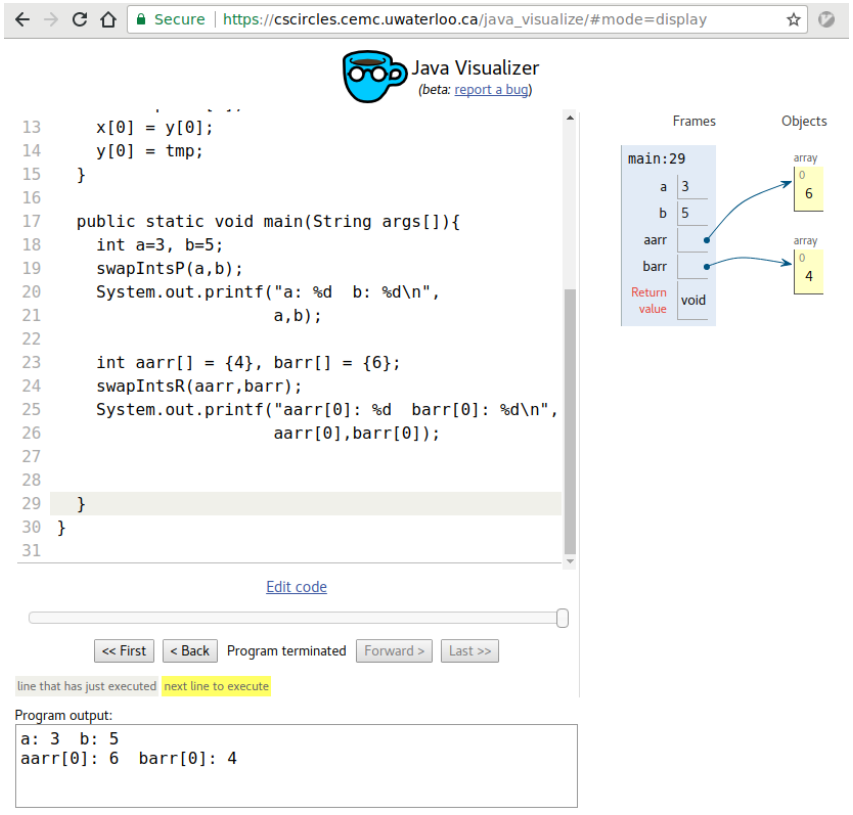
Closing Notes on Stack/Heap

Stack and Heap Models

- Accurately modeling how the stack and heap work with method calls resolves many mysteries about why code behaves certain ways
- Having a good model like this will serve you for the an entire career in computer science
- Tools like the debuggers and the Java Visualizer can help build your model (but aren't allowed on exams)

Java Visualizer

Upload code and run to "see" memory as program runs



Common Method patterns

Checking parameters for errors

Common to check parameters for consistency prior to further action

```
public static int sumRange(int arr[],
                           int start,
                           int stop)
// Sum elements in arr[] between start
// and stop-1. Returns 0 if either start
// or stop are beyond array bounds.
{
  // Check for range of start and stop
  if(start < 0 || stop > arr.length){
    return 0;
  }
  // Range okay, sum elements
  int sum = 0;
  for(int i=start; i<stop; i++){
    sum += arr[i];
  }
  return sum;
}
```

Return during loop

Common in "search" type problems to return when answer known, even mid-loop.

```
public static int containsChar(String s,
                               char c)
// Return first index of c in String
// s. If c is not present, return -1.
{
  // Search for character c
  for(int i=0; i<s.length(); i++){
    if(s.charAt(i) == c){
      return i;        // found, return
    }
  }
  // all chars checked, none matched c
  return -1;
}
```