When we declare variables, it is important to initialize them before they are used. Who is responsible for making sure that a subroutine's formal parameters have values?

- No one - the formal parameters are not required to have values.
- [✓ The caller - the formal parameters get their values from outside the subroutine. (i.e. when the subroutine is called)]
- The subroutine - there must be assignment statements at the beginning of the subroutine's body to give values to the formal parameters.
- The subroutine - there must be assignment statements somewhere in the subroutine body (before the parameters are used), though they don't necessarily have to be at the beginning.

```
    public static void foo ( int a, String b ) {
      System.out.println(“a: “+a);
      System.out.println(“length of b: “+b.length());
    }
  }
```

```
public static void printGreeting ( String name ) {
   System.out.println("Hello "+name+"!");
}
```

If you wanted to call this subroutine in order to print "Hello arthur!", what would you write?

- `printGreeting();`
- `System.out.println(printGreeting());`
- [✓ `printGreeting("arthur");`]
- `System.out.println(printGreeting("arthur"));`

-
  ```
  message = printGreeting();
  System.out.println(message);
  ```

-
  ```
  message = printGreeting("arthur");
  System.out.println(message);
  ```
- none of the above

Subroutine Contracts

- the contract allows for the separation of interface and implementation
  - defines how to use the subroutine and what it accomplishes (but not how)
  - part of the declaration

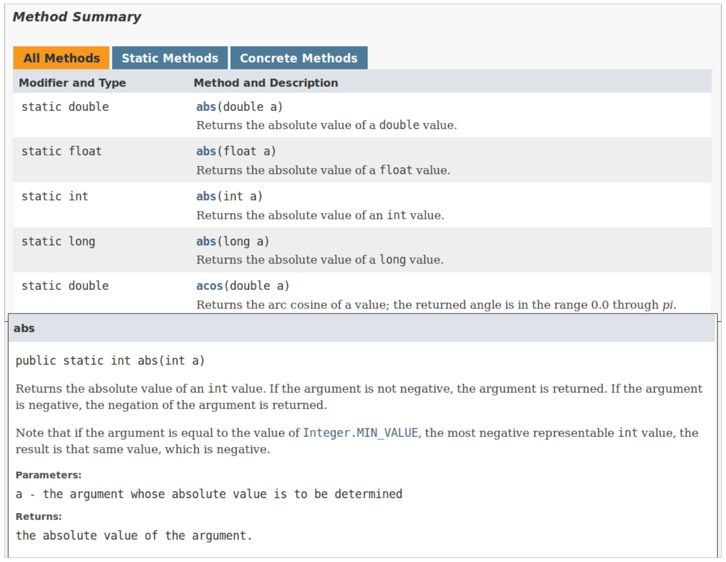
Contracts and Javadoc

A subroutine's contract tells you everything you need to know in order to use the subroutine.

- header
  - syntax of how to call the subroutine
  - types of parameters
- comment
  - what the subroutine does
  - what the parameters are for
  - what the return value is
  - preconditions

Javadoc is a tool that can generate documentation from specially-formatted comments.

- from now on, we will use javadoc style for public comments



Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method and Description
static double	abs(double a) Returns the absolute value of a double value.
static float	abs(float a) Returns the absolute value of a float value.
static int	abs(int a) Returns the absolute value of an int value.
static long	abs(long a) Returns the absolute value of a long value.
static double	acos(double a) Returns the arc cosine of a value; the returned angle is in the range 0.0 through *pi*.

abs

`public static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative.

Parameters:

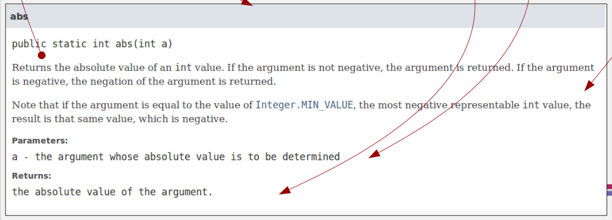
a - the argument whose absolute value is to be determined

Returns:

the absolute value of the argument.

Javadoc Format – Subroutines

```
/**
 * The first sentence should summarize what the
 * subroutine does.  Any other sentences can
 * provide more information.
 *
 * @param param1name describe param1
 * @param param2name describe param2
 * @return describe return value
 */
```



abs

`public static int abs(int a)`

Returns the absolute value of an int value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of Integer.MIN_VALUE, the most negative representable int value, the result is that same value, which is negative.

Parameters:

a - the argument whose absolute value is to be determined

Returns:

the absolute value of the argument.

Javadoc Format – Classes

```
/**
 * The first sentence should summarize the
 * purpose of the class.  Any other sentences can
 * provide more information.
 *
 * @author who wrote the class
 */
```

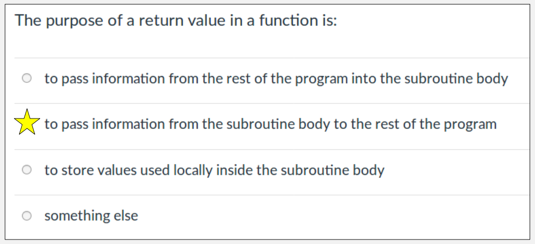
Preconditions

- *preconditions* are assumptions made in order for the subroutine to work correctly
  - e.g. specific requirements for parameter values (other than type)
  - must be stated as part of the subroutine's contract
- *robust* programs check preconditions whenever possible
  - want to fail fast if there is a problem
- convention is to throw an `IllegalArgumentException` if a precondition is violated

```
if ( precondition is violated ) {
  throw new IllegalArgumentException("detail message");
}
```

The Big Picture

- subroutines are a self-contained unit
  - variables declared inside one subroutine are not visible inside another
- *parameters* allow the caller to pass values into a subroutine
- *return values* allow the subroutine to hand one value back to the caller
  - the term *function* is often used for a subroutine that returns a value, though the terminology can be used sloppily (e.g. “function” may be used interchangeably with “subroutine”)



The purpose of a return value in a function is:

- ○ to pass information from the rest of the program into the subroutine body
- [✓ to pass information from the subroutine body to the rest of the program]
- ○ to store values used locally inside the subroutine body
- ○ something else

Syntax and Semantics

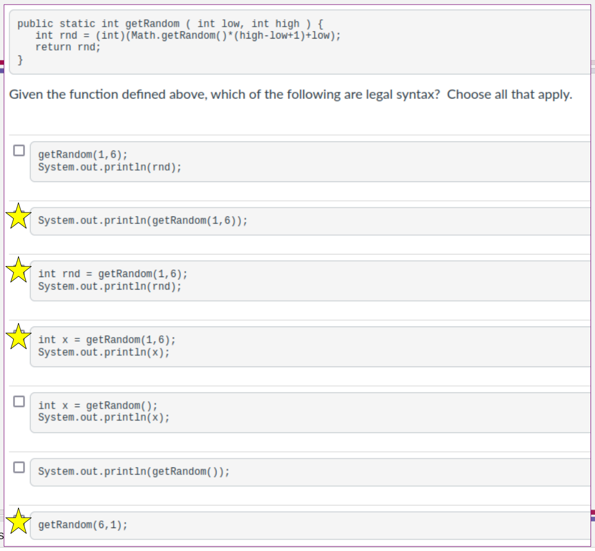
- declaration

```
modifiers return-type subroutine-name ( parameter-list ) {
   statements
}
```

  - non-`void` return type indicates that this is a function, and defines the type of the value handed back to the caller
    - can only return one thing
  - body must contain a single `return` statement for every path
    - can have multiple `return` statements, but `return` exits the function immediately so only one per path of execution
- call

  …*subroutine-name*(*parameter-values*)…

  - the statement form is also legal, but generally function calls should occur in expressions
    - otherwise the return value is ignored, which is generally not what you want



```
public static int getRandom ( int low, int high ) {
   int rnd = (int)(Math.getRandom()*(high-low+1)+low);
   return rnd;
}
```

Given the function defined above, which of the following are legal syntax? Choose all that apply.

☐
```
getRandom(1,6);
System.out.println(rnd);
```

[✓]
```
System.out.println(getRandom(1,6));
```

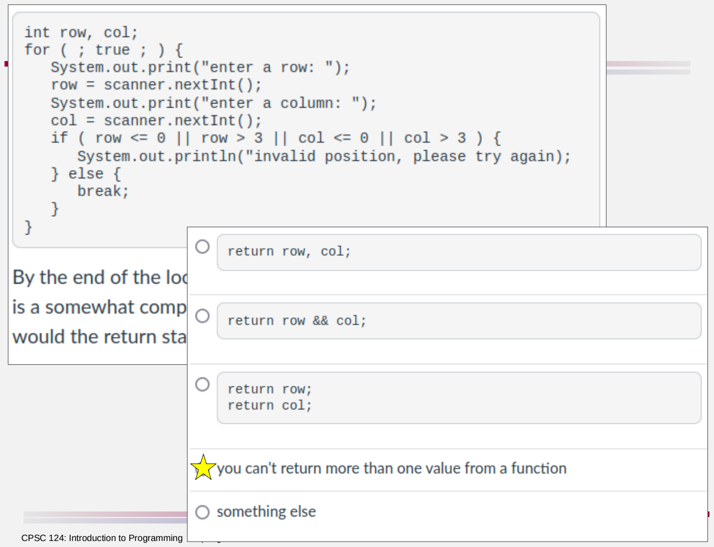
[✓]
```
int rnd = getRandom(1,6);
System.out.println(rnd);
```

[✓]
```
int x = getRandom(1,6);
System.out.println(x);
```

☐
```
int x = getRandom();
System.out.println(x);
```

☐
```
System.out.println(getRandom());
```

[✓]
```
getRandom(6,1);
```



```
int row, col;
for ( ; true ; ) {
   System.out.print("enter a row: ");
   row = scanner.nextInt();
   System.out.print("enter a column: ");
   col = scanner.nextInt();
   if ( row <= 0 || row > 3 || col <= 0 || col > 3 ) {
      System.out.println("invalid position, please try again);
   } else {
      break;
   }
}
```

By the end of the loo
is a somewhat comp
would the return sta

- ○ `return row, col;`
- ○ `return row && col;`
- ○ `return row;`
  `return col;`
- [✓ you can't return more than one value from a function]
- ○ something else

Scope

- the body of the subroutine is the cook in the kitchen
- the caller is the waiter in the dining room
- kitchen and dining room are separated – waiter can't see what is going on in the kitchen, cook can't see what is going on in the dining room
  - subroutine cannot use the caller's local variables
  - caller cannot use the subroutine's local variables
- waiter hands order slips to the cook through the pass-through
- cook hands plates of food back to the waiter
  - only one plate of food per order
- only values – the order slip, the plate of food – go through the pass through
  - named parameters allow the cook to access values passed through
  - caller must store or use the values they get back

https://www.annarbor.com/entertainment/food-drink/touring-ann-arbors-best-breakfast-joints/