Subroutines

The Big Picture

Subroutines provide a way to associate a name with a set of statements.

Subroutines are an organizational tool.

- used to manage complexity
- facilitate reuse of code

The idea of a black box is key.

- subroutine contract makes it possible to separate what the subroutine does and how to use it from how it does its job

The advantage(s) of subroutines are that... (choose all that apply)

- ★ Code can be used at many places in the program, but it only has to be written once.
- ★ Programs are easier to develop because complex tasks can be broken into simpler components.
- ★ Programs are easier to understand because a name can be used instead of a list of instructions.
- ★ You only have to understand the contract of the subroutine in order to use it.
- ☐ Subroutines must be defined inside classes.

true, but it's not really an advantage, just a fact

- are there any cases where subroutines are necessary rather than just convenient?
  - in theory, no, but really anything but the smallest programs would be too cumbersome to create without subroutines

Syntax

- declaration – to define the subroutine name

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

  - access modifiers
    - `public`, `private`, `protected`, [none] define where the subroutine name can be used
      - `private` can only be used within the same class, `public` can be used anywhere
  - other modifiers
    - `static`
  - return type – allows the subroutine to be used to compute values instead of just doing something
    - `void` if nothing to be returned
  - parameter list – allows information to be passed into the subroutine by the caller
    - empty if nothing to be passed in

```java
/**
 * Print a bingo card. Only the uncrossed off numbers are printed.
 *
 * @param card the bingo card to print
 */
public static void printBingoCard(int[] card) {
    for (int i = 0; i < card.length; i++) {
        if (card[i] != -1) {
            System.out.printf(" %2d", card[i]);
        }
    }
    System.out.println();
}
```

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


Syntax

- call – to carry out the instructions in the subroutine body

```
subroutine-name(parameter-values);
…subroutine-name(parameter-values)…
```

  - number and type of parameter values must match declaration
  - can be used as an expression only if the return type is not `void`

```java
for (int player = 0; player < cards.length; player++) {
    System.out.print("player " + player + ":");
    printBingoCard(cards[player]);
}
```


The Big Picture

Subroutines are useful for managing complexity and promoting reuse.

However, it is common to want to do the same task but with different values.

- e.g. convert Fahrenheit temperatures to Celsius (or vice versa)
- e.g. print an array
- e.g. present a math quiz problem with a particular difficulty

In `main`, we achieve this with variables instead of using literal values.

In subroutines, we achieve this with *parameters*.

- parameters are essentially variables local to a particular subroutine whose values are set when we call the subroutine instead of through assignment statements

The purpose of formal parameters in a subroutine is:

- ★ to pass information from the outside world into the subroutine body
- ○ to pass information from the subroutine body to the outside world
- ○ to store values used locally inside the subroutine body

Syntax

- declaration

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

  - parameter list – allows information to be passed into the subroutine by the caller
    - comma-separated list of *parameter declarations*, each of which has the form `type param-name`
    - parameter names are only visible/known inside the subroutine body

---

- call

```
subroutine-name(parameter-values);
```

…`subroutine-name(parameter-values)`…

  - parameter values – specifies the values for the parameters
    - comma-separated list of values
    - number and type of parameter values must match declaration


```
/**
 * Print a bingo card. Only the uncrossed off numbers are printed.
 *
 * @param card the bingo card to print
 */
public static void printBingoCard(int[] card) {
    for (int i = 0; i < card.length; i++) {
        if (card[i] != -1) {
            System.out.printf(" %2d", card[i]);
        }
    }
    System.out.println();
}
```

```
for (int player = 0; player < cards.length; player++) {
    System.out.print("player " + player + ":");
    printBingoCard(cards[player]);
}
```


Semantics

When a subroutine is called –

- boxes are created for each parameter
- the values being passed are computed and copied into the respective box
- the body of the subroutine is executed
- when the subroutine was called as a statement, control continues with the next statement after the call

```
public class Demo {
  public static void foo ( int a, String b ) {
    System.out.println("a: "+a);
    System.out.println("length of b: "+b.length());
  }
  public static void main ( String[] args ) {
    System.out.println("line 1");
    foo(10+args.length,"hello");
    System.out.println("last line");
  }
}
```


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 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