Static methods.

Software is never built as a monolithic piece of code – it would make it extremely inflexible, i.e.

- impossible to reuse code without modification
- very difficult to manage the complexity of programming, testing, modifying
- impossible to distribute work between programmers

Instead, software must be developed as a set of reusable blocks.

Methods – Java functional building blocks.

Used to implement a functionally independent task. Examples: methods of the SavitchIn class.

Building an application using simple well designed and documented methods
- makes it easier to write, read, test, and modify software
- allows for distribution of programming tasks between people
- creates reusable components

We’ll start by looking at static methods of Java.

All Java methods are defined within a class.

All Java methods have
- name
- zero or more input parameters and
- one or no return value (output).

Methods may also have
- Local variables and constants

Let’s take a look at an example: practice problem from Handout 4 implemented with static methods.
public class ElectricityBill {

    // Constants denoting base charges and rates for different account types;
    // Defined inside the class but outside of all methods; accessible by all
    // methods of this class.
    public static final double RES_BASE_CHARGE = 6; //residential
    public static final double RES_RATE = .052;
    public static final double COM_BASE_CHARGE = 6; //commercial
    public static final double COM_RATE = .045;
    public static final double IND_PEAK = 76; //industrial
    public static final double IND_OFF_PEAK = 40;
    public static final double IND_RATE = .028;

    public static void main (String args []) {

        String account;
        char accType; // 'R' for residential, 'I'- industrial, 'C'-commercial
        int consumption = 0; // kwhrs consumed
        double amountDue = 0; // to be paid for consumption

        System.out.println("This program calculates the electricity bill");
        System.out.println("Please enter the account number");
        account = SavitchIn.readLine();

        System.out.println("Please enter how many kwhrs were consumed");
        consumption = SavitchIn.readLineInt();

        System.out.println("Please enter the account type (R, C or I)");
        accType = SavitchIn.readLineNonwhiteChar();

        switch (accType){
            case 'R':
                case 'r':
                    amountDue = computeResidential(consumption);
                    break;
            case 'C':
                case 'c':
                    amountDue = computeCommercial(consumption);
                    break;
            case 'I':
                case 'i':
                    System.out.println("Was electricity consumed during peak hours?
(yes/no)");
                    String peakAnswer = SavitchIn.readLine();
                    boolean isPeak = false;
                    if (peakAnswer.equalsIgnoreCase("yes"))
                        isPeak = true;
                    else
                        isPeak = false;
                    amountDue = computeIndustrial(consumption, isPeak);
                    break;
            default:
                System.out.println("Incorrect account type");
                return;
        }
    }

    public static double computeResidential(int consumption) {
        return RES_BASE_CHARGE + (RES_RATE * consumption);
    }

    public static double computeCommercial(int consumption) {
        return COM_BASE_CHARGE + (COM_RATE * consumption);
    }

    public static double computeIndustrial(int consumption, boolean isPeak) {
        return IND_PEAK + (IND_RATE * consumption) + (IND_OFF_PEAK * (1 - isPeak));
    }
}
System.out.println("The amount due account " +account+" is " + amountDue);
}

/* double computeResidential(int consumed)
* computes and returns amount due for using consumed kw-hrs of
* electricity
*/

public static double computeResidential(int consumed)
{
    // local variable
    double cost;
    cost = RES_BASE_CHARGE + RES_RATE*consumed;
    return (cost);
}

/* method double computeCommercial (int consumed)
* computes and returns amount due for using consumed kw-hrs of
* electricity
*/

public static double computeCommercial(int consumed)
{
    // local variables
    double cost;
    int over1000 = consumed - 1000;
    if (consumed <= 1000)
        cost = COM_BASE_CHARGE;
    else
        cost = COM_BASE_CHARGE + COM_RATE * over1000;
    return (cost);
}

/* method double computeIndustrial (int consumed, boolean peak)
* computes and returns amount due for using consumed kw-hrs of
* electricity
*/

public static double computeIndustrial(int consumed, boolean peak)
{
    // local variables
    double cost;
    int over1000 = consumed - 1000;
    if (peak) // same as if (peak == true)
        cost = IND_PEAK;
    else
        cost = IND_OFF_PEAK;
    if (consumed > 1000)
        cost = cost + IND_RATE*over1000;
    return (cost);
}
Method declaration and invocation

- A *method declaration* specifies the code that will be executed when the method is invoked (or called)

- A method declaration begins with a method header:

  ```java
  public static double computeResidential (int consumed)
  ```

  - *return type*, i.e. type of return value
  - *name*
  - type and name of *formal parameter* (aka argument)

- Followed by method *body* in { … }

  ```java
  { 
  double cost; // cost is a *local variable*
  (declared inside method body)
  cost = RES_BASE_CHARGE + RES_RATE*consumed;
  return (cost); // value returned from the method
  }
  ```

- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined. A method that does not return a value has a *void* return type.
- There are two different ways of calling a static method
  1. By name, e.g.
     ```java
     amountDue = computeResidential(consumption);
     ```
     works only from within the same class.
  2. Specifying the class name as in
     ```java
     amountDue = ElectricityBill.computeResidential(consumption);
     ```
     works from within another class as well (e.g. SavitchIn.readLine() – SavitchIn is the name of the class that contains a static methods readLine() )
Parameter passing:

Let’s consider a slightly more complex example in which the method has three parameters of different types:

```java
public static char demo (int num1, int num2, String message) {
    int sum;
    char result;
    sum = num1 + num2;
    result = message.charAt (sum);
    return result;
}
```

Suppose this method is invoked in the following call where myNum = 5:

```java
aChar = demo (myNum, 3, “Hello World “)
```

What happens:

a. Parameters in the call are evaluated and assigned to the method’s arguments, i.e.
   ```
   5   3   “Hello World”
   demo (int num1, int num2, String message)
   ```

b. The control is passed to the first statement of the method.

c. Statements are executed in order until a return statement is encountered:

   ```java
   return result;
   ```

Return statement terminates the execution of the method and passes control back to the point from where the method was invoked;

The returned value (i.e. value of `result`) is substituted in place of the call, i.e.

```java
aChar = demo (myNum, 3, “Hello World “)
```

```
‘r’
```
Notes on the return statement:

- the type of value returned must match the method’s return type, specified in the method header.
- a method may have any number of return statements, but it is a rule of good programming style to have a unique exit point from a method.
- return statement terminates the method execution.
- a method that does not return a value has a `void` return type.

Scoping rules

- Local variables can be declared inside a method.
- Local variables and formal parameters are accessible only from within the method definition and are not accessible from outside of the method.
- The formal parameters of a method create *automatic local variables* when the method is invoked.
- When the method finishes, all local variables are destroyed (including the formal parameters).

Practice Problem:

1. Define a static method that has one integer parameter and returns true if and only if the parameter is even. Show how to call this method to check if a number stored in variable x is even.